EDUSHARK TRAINING

Self-paced programmes in data and AI

Machine Learning

A 16-Week Practitioner Workbook

Build models, measure them honestly, and ship one.

edusharktraining.com

Machine Learning: A 16-Week Practitioner Workbook

Publisher: EDUSHARK TRAINING
Web: https://edusharktraining.com/machine-learning
Contact: info@edusharktraining.com

Typeset with LaTeX.

How to Use This Workbook

This book covers 16 weeks, 112 lessons and 128 self-assessment questions. Assumes some Python. No prior machine learning.

What is different about it

Every result in this book came out of code that was run, and the printed output is what the machine actually returned. Where a technique lost to something simpler, the number that says so is on the page next to it.

How each chapter is laid out

One chapter per week, seven days per chapter. Each day states what it is for, works through the material, and ends with a takeaway worth keeping. Code appears as it was run, and program output is set apart in grey so you can tell at a glance which is which:

```
print(sum(x * x for x in range(4)))
```

```
14
```

Every chapter ends with a self-assessment quiz. The answers are on the same page, set small, because a question you cannot check is not worth attempting. Anything you get wrong points at the day to re-read.

Working alongside the site

Every chapter here matches a week published at https://edusharktraining.com/machine-learning. The website version carries the same material with the code laid out for copying. Use whichever suits where you are reading.

Contents

Chapter 1

Python for Machine Learning

Week 1

OBJECTIVES

Set up the environment, learn the array and the DataFrame, and build a working churn model before the week is out.

Week 1 does two jobs. It gives you the Python you need, arrays, DataFrames, plots, and the three-method contract every scikit-learn object obeys, and it gets you to a real, measured model by day 7 rather than making you wait a month for one. You will also generate the customer dataset the rest of the course is taught on, deliberate defects and all.

Day 1 What Machine Learning Actually Is

Supervised, unsupervised, and the one idea the whole field rests on

Most programming you have done so far is *instructional*. You knew the rule, add tax at 20 percent, reject an order under £5, and you wrote it down. Machine learning inverts that. You have the answers and you want the rule, and the rule is one nobody can state precisely.

Nobody can write the condition for "this customer is about to leave". But a company has thousands of customers who did leave and thousands who stayed, and the difference is in the data somewhere. A learning algorithm searches for it.

The three families you will meet

Almost everything in this course sits in one of three boxes. Knowing which box a problem belongs to decides which algorithms are even candidates.

Family	You give it	It learns to	Example
Supervised	Inputs *and* known answers	Predict the answer for new inputs	Will this customer churn?
Unsupervised	Inputs only	Find structure, groups, axes, oddities	Which customers behave alike?
Reinforcement	An environment and a reward	Choose actions that pay off over time	Which offer to show, learning as you go

Weeks 4 to 7 are supervised learning, weeks 9 and 10 are unsupervised. Reinforcement learning is a course of its own and we only sketch it.

Supervised splits again, by what you are predicting

Inside supervised learning the type of the answer decides the metric, the loss function and often the algorithm:

- **Regression**: the answer is a number on a continuous scale. How much will this customer spend next quarter?
- **Classification**: the answer is one of a fixed set of labels. Churn or stay. Fraud or legitimate. Which of nine defect types.

The same dataset can pose both. `monthly_charges` is a regression target; `churned` is a classification target. You will use both this course.

The one idea that separates ML from statistics

A statistician asks whether a relationship is real. A machine learning practitioner asks whether a model will work on data it has not seen. That second question has a name, and it is the discipline the whole field is built on.

Watch Out

The mistake that ruins most first projects

You fit a model, score it on the same rows you fitted it on, and get 99 percent. It feels like success. It is the machine learning equivalent of marking your own exam with the answer sheet open. Any model complex enough can memorise its training set. Week 7 is devoted to measuring this honestly; until then, treat every score computed on training data as meaningless.

What you need installed

Python 3.10 or newer, and a handful of libraries. Install Python from python.org and tick *Add Python to PATH* on Windows, or use your package manager on Linux and macOS. Check it took:

Work in a virtual environment, always

A virtual environment is a private copy of Python's package folder for one project. Without one, every project on your machine shares a single set of library versions, and the day a project needs an older scikit-learn you break the other five. Create one per project:

Your prompt gains a `(.venv)` prefix. That prefix is the whole point: it tells you which Python you are about to run. Now install the stack:

Pin what you installed

Run `pip freeze > requirements.txt` once the environment works. That file records exact versions, so `pip install -r requirements.txt` reproduces it on another machine, or on yours in six months, when the newest scikit-learn has renamed the argument your code passes.

How you will actually work

Two tools, two jobs. Use a notebook while you are exploring, because you want to see each intermediate result. Move to a `.py` file the moment the code matters, because notebooks hide

execution order and a notebook that only runs if you press the cells in the right sequence is not reproducible.

> **Day 1 takeaway**

Day 2 NumPy and Vectorised Thinking

Arrays, shapes, broadcasting and why loops are the wrong habit

Every machine learning library in Python stands on NumPy. A scikit-learn model does not really take a DataFrame; it takes an array of numbers, and pandas hands one over on request. TensorFlow tensors follow NumPy's rules deliberately. Learn the array and the rest of the stack stops feeling arbitrary.

Why not a plain Python list

A Python list holds pointers to objects scattered across memory, each one carrying its own type information. A NumPy array holds raw numbers of one type packed end to end. That single difference buys both compactness and speed, because the loop that adds them runs in C instead of in the interpreter.

```
import numpy as np

py_list = list(range(1_000_000))
arr = np.arange(1_000_000)

print(type(arr), arr.dtype, arr.shape)
print('list  bytes:', py_list.__sizeof__())
print('array bytes:', arr.nbytes)
```

```
<class 'numpy.ndarray'> int64 (1000000,)
list  bytes: 8000040
array bytes: 8000000
```

The sizes look similar, but the list figure counts only the pointers. Each of the million integers is a separate object of roughly 28 bytes on top of that. The array figure is the whole story.

Vectorised thinking

The habit to build this week: stop writing loops over rows. An operation written against a whole array is shorter, faster and less likely to be wrong.

```
import numpy as np
import time

charges = np.random.default_rng(0).uniform(20, 120, 1_000_000)

t = time.perf_counter()
loop = [c * 1.2 for c in charges]
loop_time = time.perf_counter() - t

```

```
t = time.perf_counter()
vec = charges * 1.2
vec_time = time.perf_counter() - t

print(f'loop       {loop_time:4f}s')
print(f'vectorised {vec_time:4f}s')
print(f'speedup    {loop_time / vec_time:0f}x')
```

```
loop       0.083917s
vectorised 0.002325s
speedup    36.093464x
```

Forty-something times, for arithmetic. On the operations that matter in model fitting the gap is wider still, because those call into optimised linear algebra libraries.

Shape is the thing you will get wrong

Nearly every confusing error in scikit-learn and TensorFlow is a shape error. Build the habit now of printing `.shape` whenever something surprises you.

```
import numpy as np

a = np.array([1, 2, 3])              # 1-D, shape (3,)
b = np.array([[1, 2, 3]])            # 2-D, one row
c = np.array([[1], [2], [3]])        # 2-D, one column

for name, x in [('a', a), ('b', b), ('c', c)]:
    print(name, x.shape, x.ndim)
```

```
a (3,) 1
b (1, 3) 2
c (3, 1) 2
```

Watch Out

(3,) is not (3, 1)

A one-dimensional array of three numbers and a column of three numbers are different objects, and scikit-learn cares. Fitting on a 1-D feature array raises *Expected 2D array, got 1D array instead*. The fix is `x.reshape(-1, 1)`, where `-1` means *work out this dimension from the length*.

Broadcasting

When you combine arrays of different shapes, NumPy stretches the smaller one to fit, if the shapes are compatible. Compatible means: compare dimensions from the right, and each pair must be equal or one of them must be 1.

```
import numpy as np

X = np.array([[10.0, 200.0],
              [12.0, 240.0],
```

```
              [11.0, 100.0]])

col_mean = X.mean(axis=0)      # shape (2,) -- one mean per column
centred = X - col_mean         # (3, 2) - (2,) broadcasts down the rows

print('means  ', col_mean)
print('centred\n', centred)
print('new means', centred.mean(axis=0).round(10))
```

```
means   [ 11. 180.]
centred
 [[ -1.  20.]
 [  1.  60.]
 [  0. -80.]]
new means [0. 0.]
```

That is standardisation, in one line, and it is exactly what `StandardScaler` does in week 8. `axis=0` means "collapse the rows, keep the columns", which is the direction you almost always want, because columns are features.

Boolean masks

Selecting rows by condition is the operation you will reach for constantly.

```
import numpy as np

charges = np.array([25.0, 88.5, 104.0, 19.0, 71.2])
high = charges > 70

print('mask   ', high)
print('values ', charges[high])
print('count  ', high.sum())      # True counts as 1
print('share  ', high.mean())     # so the mean is the proportion
```

```
mask    [False  True  True False  True]
values  [ 88.5 104.   71.2]
count   3
share   0.6
```

mask.mean() is a proportion

Because `True` is 1 and `False` is 0, the mean of a boolean array is the fraction of elements that satisfy the condition. `(y == 1).mean()` is your churn rate. You will use this constantly and it saves defining a counter.

> **Day 2 takeaway**

Day 3 pandas and the Course Dataset

Series, DataFrames, group-by, and generating the data you will use

NumPy gives you a grid of one type. Real data is a table with a name and a type per column, missing values, and dates. That is pandas.

Series and DataFrame

A `Series` is one column: values plus an index. A `DataFrame` is a dictionary of Series sharing that index.

```
import pandas as pd

s = pd.Series([25.0, 88.5, 104.0], name='monthly_charges')
print(s)
print('dtype:', s.dtype)

df = pd.DataFrame({
    'customer_id': ['C1', 'C2', 'C3'],
    'monthly_charges': [25.0, 88.5, 104.0],
    'contract': ['Month-to-month', 'Two year', 'Month-to-month'],
})
print(df)
print(df.dtypes)
```

```
0     25.0
1     88.5
2    104.0
Name: monthly_charges, dtype: float64
dtype: float64
  customer_id  monthly_charges        contract
0          C1             25.0  Month-to-month
1          C2             88.5        Two year
2          C3            104.0  Month-to-month
customer_id         object
monthly_charges    float64
contract            object
dtype: object
```

Watch Out

object means pandas gave up

A dtype of `object` means the column holds Python objects, usually strings. That is right for `contract`. It is a bug for anything you intend to do arithmetic on. When a column you expected to be numeric shows as `object`, one value in it is not a number, and you will meet exactly that tomorrow.

Selecting: loc and iloc

Two accessors, and mixing them up is the most common pandas error. `.loc` selects by *label*, `.iloc` by *integer position*.

```
import pandas as pd

df = pd.DataFrame({
```

```
    'tenure': [3, 40, 12, 65],
    'charges': [70.0, 55.5, 99.9, 21.0],
    'churned': [1, 0, 1, 0],
}, index=['a', 'b', 'c', 'd'])

print(df.loc['b', 'charges'])          # label row, label column
print(df.iloc[1, 1])                   # second row, second column
print(df.loc[df['churned'] == 1, ['tenure', 'charges']])
```

```
55.5
55.5
   tenure  charges
a       3     70.0
c      12     99.9
```

That last line is the pattern you will use every day: a boolean condition picks the rows, a list of names picks the columns.

Grouping answers most questions

"Does churn differ by contract type?" is a group-by.

```
import pandas as pd

df = pd.DataFrame({
    'contract': ['Month-to-month'] * 4 + ['Two year'] * 4,
    'churned':  [1, 1, 0, 1,          0, 0, 0, 1],
    'charges':  [70, 82, 65, 90,      55, 40, 61, 58],
})

summary = df.groupby('contract').agg(
    customers=('churned', 'size'),
    churn_rate=('churned', 'mean'),
    avg_charge=('charges', 'mean'),
)
print(summary.round(3))
```

```
               customers  churn_rate  avg_charge
contract
Month-to-month          4        0.75       76.75
Two year                4        0.25       53.50
```

Named aggregation, `new_name=('column', 'function')`, is worth learning properly. It produces flat, readable column names instead of the nested mess the older syntax gives you.

Make pandas print the whole frame

By default pandas folds wide tables and replaces the middle columns with an ellipsis, so the output you see depends on how wide your terminal happens to be. Set these two options once at the top of every notebook. Every printed result in this course was produced with them set, so your output will match what these pages show.

```
import pandas as pd

```

```
pd.set_option('display.width', 100)
pd.set_option('display.max_columns', 25)
print('pandas', pd.__version__)
```

```
pandas 2.2.3
```

Build the dataset for the rest of the course

Public churn datasets move, vanish behind logins, or change shape. So you will generate yours. Save this as `make_dataset.py` next to your notebook and run it once. It is deterministic: the same seed gives everyone the same file, so the numbers in these pages are the numbers you will see.

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(42)
n = 3000

contract = rng.choice(['Month-to-month', 'One year', 'Two year'],
                      size=n, p=[0.55, 0.25, 0.20])
internet = rng.choice(['Fibre optic', 'DSL', 'No internet'], size=n,
                      p=[0.44, 0.34, 0.22])
payment = rng.choice(['Electronic check', 'Mailed check',
                      'Bank transfer', 'Credit card'], size=n,
                     p=[0.34, 0.23, 0.22, 0.21])
dependents = rng.choice(['Yes', 'No'], size=n, p=[0.3, 0.7])

# Two-year customers have by definition been around longer.
base = np.where(contract == 'Two year', 42,
                np.where(contract == 'One year', 28, 16))
tenure = np.clip(rng.gamma(2.2, base / 2.2), 1, 72).round().astype(int)

price = np.where(internet == 'Fibre optic', 79,
                 np.where(internet == 'DSL', 55, 21))
monthly = np.clip(rng.normal(price, 9), 15, 125).round(2)
support = rng.poisson(np.where(internet == 'Fibre optic', 1.6, 0.9))

# The rule a model is meant to recover.
logit = (-2.8
         + 1.55 * (contract == 'Month-to-month')
         - 0.85 * (contract == 'Two year')
         - 0.055 * tenure
         + 0.021 * monthly
         + 0.30 * support
         + 0.42 * (payment == 'Electronic check')
         - 0.25 * (dependents == 'Yes')
         + rng.normal(0, 0.45, n))
churned = (1 / (1 + np.exp(-logit)) > rng.uniform(size=n)).astype(int)

total = (monthly * tenure * rng.uniform(0.94, 1.06, n)).round(2)
start = np.datetime64('2019-01', 'M')
signup = (start + (72 - tenure).astype('timedelta64[M]')).astype('datetime64[D]')
```

```

df = pd.DataFrame({
    'customer_id': ['C%05d' % i for i in range(1, n + 1)],
    'signup_date': pd.to_datetime(signup).astype(str),
    'tenure_months': tenure,
    'contract': contract,
    'internet_service': internet,
    'payment_method': payment,
    'has_dependents': dependents,
    'support_calls': support,
    'monthly_charges': monthly,
    'total_charges': total,
    'churned': churned,
})

# --- and now spoil it, the way collection genuinely spoils data ---
fresh = df.index[df['tenure_months'] <= 1]
df.loc[fresh, 'total_charges'] = np.nan
# A space, not an empty string: pandas reads '' as missing, which would
# hide the very defect this is meant to reproduce.
df['total_charges'] = df['total_charges'].map(
    lambda v: ' ' if pd.isna(v) else ('%.2f' % v))

drift = rng.choice(df.index, size=int(n * 0.07), replace=False)
df.loc[drift, 'contract'] = df.loc[drift, 'contract'].map(
    lambda s: rng.choice([s.upper(), s.lower(), ' ' + s, s + ' ']))

for col, frac in (('has_dependents', 0.06), ('monthly_charges', 0.055)):
    gaps = rng.choice(df.index, size=int(n * frac), replace=False)
    df.loc[gaps, col] = np.nan

df.loc[df.index[7], 'support_calls'] = 97
df = pd.concat([df, df.iloc[500:638]], ignore_index=True)
df = df.sample(frac=1, random_state=42).reset_index(drop=True)

df.to_csv('customers.csv', index=False)
print('customers.csv:', df.shape, 'churn rate %.1f%%' % (df['churned'].mean() *
    ↪ 100))
```

```
customers.csv: (3138, 11) churn rate 26.8%
```

The mess is on purpose

Blank totals, four spellings of one contract type, 138 duplicated rows, one customer with 97 support calls and gaps in two columns. Every one of those is a defect you will meet in real extracts, and every one teaches something specific in the weeks ahead. Cleaning it is tomorrow.

> **Day 3 takeaway**

Day 4 First Contact With Messy Data

Dtypes that lie, categories that multiply, duplicates and gaps

You have a CSV. Before any modelling, you need to know what is actually in it, and the answer is never what the column names promise.

First contact

```python
import pandas as pd

df = pd.read_csv('customers.csv')
print(df.shape)
print(df.dtypes)
```

```
(3138, 11)
customer_id          object
signup_date          object
tenure_months         int64
contract             object
internet_service     object
payment_method       object
has_dependents       object
support_calls         int64
monthly_charges     float64
total_charges        object
churned               int64
dtype: object
```

Two things are already wrong. `signup_date` is text, not a date, expected, since CSV has no date type. But `total_charges` is text too, and that is a defect: it should be a number.

Why a numeric column arrives as text

```python
import pandas as pd
df = pd.read_csv('customers.csv')

numeric = pd.to_numeric(df['total_charges'], errors='coerce')
bad = df.loc[numeric.isna(), ['customer_id', 'tenure_months', 'total_charges']]
print('unconvertible rows:', len(bad))
print(bad.head())
```

```
unconvertible rows: 15
    customer_id  tenure_months total_charges
220      C00175              1
531      C02127              1
624      C02779              1
749      C02411              1
870      C00436              1
```

Fifteen rows, every one of them a customer whose tenure is one month. They have not been billed yet, so the field holds a single space, and one non-numeric value among three thousand numbers forces the whole column to text. `errors='coerce'` turns what cannot be parsed into `NaN` instead of raising, which is what you want here.

Watch Out

Do not fill those with zero

Zero means "billed nothing". These customers mean "not billed yet". Writing zero invents a fact and drags the column mean down. Leave them missing and decide deliberately in week 8: missingness that carries meaning is a feature, not a nuisance.

Categories that are not as tidy as they look

```
import pandas as pd
df = pd.read_csv('customers.csv')

print('distinct contract values:', df['contract'].nunique())
print(df['contract'].value_counts())
```

```
distinct contract values: 15
contract
Month-to-month     1611
One year            710
Two year            592
 Month-to-month      43
MONTH-TO-MONTH       31
Month-to-month       25
month-to-month       22
TWO YEAR             18
 One year            18
One year             17
ONE YEAR             14
Two year             12
 Two year            10
two year              8
one year              7
Name: count, dtype: int64
```

Fifteen levels of a three-level variable. Group by this column and you get fifteen groups; one-hot encode it and you get fifteen columns, twelve of which are near-empty noise. Normalising is two calls:

```
import pandas as pd
df = pd.read_csv('customers.csv')

df['contract'] = df['contract'].str.strip().str.title()
print(df['contract'].value_counts())
```

```
contract
Month-To-Month    1732
One Year           766
Two Year           640
Name: count, dtype: int64
```

`.str.strip()` removes the padding, `.str.title()` puts casing back in one form. Three levels, as intended.

Duplicates

```
import pandas as pd
df = pd.read_csv('customers.csv')

print('fully duplicated rows:', df.duplicated().sum())
print('repeated customer ids:', df['customer_id'].duplicated().sum())

clean = df.drop_duplicates()
print('rows before', len(df), '→ after', len(clean))
```

```
fully duplicated rows: 138
repeated customer ids: 138
rows before 3138 → after 3000
```

Watch Out

Duplicates leak across a train/test split

138 rows appear twice. Split the data afterwards and some customers land in *both* halves. The model then gets tested on rows it trained on, and the test score reports memorisation as skill. De-duplicate before you split, never after.

Missing values

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

missing = df.isna().sum()
print(missing[missing > 0])
print('\nrows with at least one gap:', df.isna().any(axis=1).sum())
```

```
has_dependents     180
monthly_charges    165
total_charges       15
dtype: int64

rows with at least one gap: 346
```

346 rows of 3000. Dropping them costs you eleven percent of the data and, worse, assumes the gaps are random. If customers who decline to state dependents churn differently, dropping them biases the model. Week 8 handles this properly with imputation inside a pipeline.

The target

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates()

print(df['churned'].value_counts())
print('churn rate: %.1f%%' % (df['churned'].mean() * 100))
```

```
churned
0    2196
1     804
Name: count, dtype: int64
churn rate: 26.8%
```

Write that number down

26.8 percent churn means a model that predicts "nobody churns" and never gets anything right is **73.2 percent accurate**. That is your floor. Any accuracy you report from now on has to be compared against it, and week 7 will show you why accuracy is the wrong headline metric here at all.

> **Day 4 takeaway**

Day 5 Seeing the Data Before Modelling It

matplotlib, seaborn, and the charts that actually decide things

Summary statistics hide shape. Two columns with the same mean and standard deviation can look nothing alike, and the difference decides which model will work. Plot before you model.

matplotlib, and the one thing to learn about it

matplotlib has two interfaces. The `plt.plot()` style keeps hidden state about which figure is current and breaks the moment you draw two things. Use the explicit style: ask for a figure and axes, then draw on the axes.

```
import matplotlib
matplotlib.use('Agg')          # no window needed; write straight to a file
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates()

fig, ax = plt.subplots(figsize=(7, 4))
ax.hist(df['tenure_months'], bins=36, color='#2563eb', edgecolor='white')
ax.set_xlabel('Tenure (months)')
ax.set_ylabel('Customers')
ax.set_title('How long customers have been with us')
fig.tight_layout()
fig.savefig('tenure.png', dpi=120)
plt.close(fig)
print('written')
```

```
written
```

Agg means headless

`matplotlib.use('Agg')` selects a backend that renders to a file with no display attached. In a notebook you do not need it. In a script, on a server, or in a scheduled job you do. Without it, matplotlib tries to open a window and the job hangs or crashes.

Comparing a distribution across groups

The question that matters is never "what does tenure look like", it is "does tenure look different for the people who left". Overlay the two.

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates()
stayed = df.loc[df['churned'] == 0, 'tenure_months']
left = df.loc[df['churned'] == 1, 'tenure_months']

fig, ax = plt.subplots(figsize=(7, 4))
ax.hist([stayed, left], bins=24, density=True,
        label=['Stayed', 'Churned'], color=['#2563eb', '#dc2626'])
ax.set_xlabel('Tenure (months)')
ax.set_ylabel('Share of group')
ax.legend()
fig.tight_layout()
fig.savefig('tenure_by_churn.png', dpi=120)
plt.close(fig)

print(df.groupby('churned')['tenure_months'].describe().round(1))
```

```
          count  mean   std  min   25%   50%   75%   max
churned
0        2196.0  26.9  18.7  1.0  13.0  22.0  37.0  72.0
1         804.0  14.1   9.1  1.0   8.0  12.0  18.0  72.0
```

density=True, not counts

2196 customers stayed and 804 left. Plot raw counts and the churned bars are dwarfed regardless of shape. `density=True` scales each group to its own area, so you compare shapes rather than group sizes. Getting this wrong is the most common misleading chart in churn analysis.

The medians are 22 months against 12. That gap is the single strongest signal in the dataset, and any model that fails to find it is broken.

seaborn for the statistical plots

seaborn sits on matplotlib and knows about DataFrames, so the plots you want during exploration are one line each.

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

```python
import pandas as pd
import seaborn as sns

df = pd.read_csv('customers.csv').drop_duplicates()
df['contract'] = df['contract'].str.strip().str.title()

fig, ax = plt.subplots(figsize=(7, 4))
sns.barplot(data=df, x='contract', y='churned', errorbar=('ci', 95), ax=ax)
ax.set_ylabel('Churn rate')
ax.set_xlabel('')
fig.tight_layout()
fig.savefig('churn_by_contract.png', dpi=120)
plt.close(fig)

print(df.groupby('contract')['churned'].agg(['size', 'mean']).round(3))
```

```
               size   mean
contract
Month-To-Month  1660  0.414
One Year         731  0.137
Two Year         609  0.028
```

Because `churned` is 0 or 1, its mean *is* the churn rate, so a bar chart of the mean is a bar chart of the rate. 41 percent against under 3 percent: contract type matters enormously.

Correlation, and what it will not tell you

```python
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

num = ['tenure_months', 'monthly_charges', 'total_charges',
       'support_calls', 'churned']
print(df[num].corr().round(2))
```

```
                tenure_months  monthly_charges  total_charges  support_calls  churned
tenure_months            1.00             0.00           0.83           0.03    -0.32
monthly_charges          0.00             1.00           0.45           0.15     0.21
total_charges            0.83             0.45           1.00           0.10    -0.21
support_calls            0.03             0.15           0.10           1.00     0.08
churned                 -0.32             0.21          -0.21           0.08     1.00
```

Watch Out

0.83 between tenure and total charges

Of course: total charges is roughly monthly charges multiplied by tenure. The column is nearly a restatement of two others. Correlations that high between features cause unstable coefficients in linear models, a problem called multicollinearity that week 4 addresses. Note it now; do not delete anything yet.

Also note that correlation only sees straight-line relationships between numbers. It says nothing

about `contract`, which the bar chart just showed is the strongest predictor you have. A correlation matrix is a starting point, never a feature-selection method.

Day 5 takeaway

Day 6 The scikit-learn Contract

fit, predict, transform, and the rule about fitting on test data

scikit-learn is large, but it is not complicated, because nearly every object in it obeys the same three-method contract. Learn the contract once and you can use a model you have never heard of.

The contract

Method	Who has it	What it does
`fit(X, y)`	Everything	Learn from data. Returns the object itself.
`predict(X)`	Models	Produce an answer for new rows.
`transform(X)`	Preprocessors	Produce a modified version of the input.
`fit_transform(X)`	Preprocessors	Both, in one call. Training data only.
`predict_proba(X)`	Most classifiers	Class probabilities rather than a hard label.

The contract in action

```python
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=200, n_features=3, noise=12, random_state=0)

# Identical calls, entirely different algorithms.
for model in [LinearRegression(), RandomForestRegressor(random_state=0)]:
    model.fit(X, y)
    pred = model.predict(X[:3])
    print(f'{type(model).__name__:24s} {pred.round(1)}')
```

```
LinearRegression         [ 21.3 -94.4 204.5]
RandomForestRegressor    [ 17.4 -87.2 195.7]
```

Learned state lives on the object, with a trailing underscore

A convention worth knowing: attributes that exist only after `fit` end in an underscore. If you see `coef_` or `n_features_in_`, that value was learned from data. Anything without the underscore is a setting you chose.

```python
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=200, n_features=3, noise=12, random_state=0)
model = LinearRegression()

print('before fit:', [a for a in dir(model) if a.endswith('_')
                      and not a.startswith('_')])
model.fit(X, y)
print('after fit: coef_ =', model.coef_.round(2))
print('           intercept_ = %.2f' % model.intercept_)
print('           n_features_in_ =', model.n_features_in_)
```

```
before fit: []
after fit: coef_ = [25.31 60.45 57.45]
           intercept_ = -0.33
           n_features_in_ = 3
```

Transformers, and the rule about fitting them

A scaler learns something from the data, the mean and standard deviation of each column, and that makes it dangerous. It must learn from training data only.

```python
import numpy as np
from sklearn.preprocessing import StandardScaler

train = np.array([[10.0], [20.0], [30.0], [40.0]])
test = np.array([[50.0], [60.0]])

scaler = StandardScaler()
train_scaled = scaler.fit_transform(train)   # learn AND apply
test_scaled = scaler.transform(test)         # apply only

print('learned mean %.1f, scale %.2f' % (scaler.mean_[0], scaler.scale_[0]))
print('train:', train_scaled.ravel().round(2))
print('test: ', test_scaled.ravel().round(2))
```

```
learned mean 25.0, scale 11.18
train: [-1.34 -0.45  0.45  1.34]
test:  [2.24 3.13]
```

Watch Out

fit_transform on test data is a leak

Call `fit_transform` on the test set and the scaler learns the test set's mean. Information about data the model is not supposed to have seen flows into the features, the test score improves, and it is a lie. `fit_transform` on train, `transform` on test, every time. Week 8's pipelines make it impossible to get backwards, which is exactly why they exist.

Reading the documentation

Every estimator's docstring lists its hyperparameters with defaults and its learned attributes. You do not need to memorise them:

```
from sklearn.ensemble import RandomForestClassifier

params = RandomForestClassifier().get_params()
for k in ['n_estimators', 'max_depth', 'min_samples_leaf', 'random_state']:
    print(f'{k:20s} {params[k]}')
```

```
n_estimators         100
max_depth            None
min_samples_leaf     1
random_state         None
```

Set random_state on everything

`random_state=None` means results change between runs. For anything you intend to compare, report or debug, pass an explicit integer. Two models whose scores differ by 0.4 percent are indistinguishable if you never fixed the seed.

> **Day 6 takeaway**

Day 7 Your First End-to-End Model

Clean, split, pipeline, fit, and measure against an honest baseline

Everything so far, assembled. Today you build a working churn model, measure it honestly, and see why the obvious metric misleads.

Step 1: load and clean

```
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

print(df.shape, '| churn rate %.3f' % df['churned'].mean())
```

```
(3000, 11) | churn rate 0.268
```

Step 2: split before anything else

The split comes first. Any decision you make after looking at the test set, which columns to keep, how to fill gaps, which model to use, contaminates it.

```
import pandas as pd
```

```
from sklearn.model_selection import train_test_split

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

num_cols = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
cat_cols = ['contract', 'internet_service', 'payment_method', 'has_dependents']

X = df[num_cols + cat_cols]
y = df['churned']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, stratify=y, random_state=42)

print('train', X_train.shape, 'churn %.3f' % y_train.mean())
print('test ', X_test.shape, 'churn %.3f' % y_test.mean())
```

```
train (2250, 8) churn 0.268
test  (750, 8) churn 0.268
```

stratify=y

Without it, a random split can hand you a test set with a noticeably different churn rate than the training set, and the score you get reflects that accident rather than the model. `stratify` keeps the class proportions identical in both halves. Always pass it for classification.

Step 3: one object that preprocesses and predicts

Numeric columns need their gaps filled and their scales evened out. Categorical columns need turning into numbers. A `ColumnTransformer` applies different treatment to different columns; a `Pipeline` chains that to the model so the whole thing behaves like a single estimator.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
num_cols = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
cat_cols = ['contract', 'internet_service', 'payment_method', 'has_dependents']
X, y = df[num_cols + cat_cols], df['churned']
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, stratify=y, random_state=42)

numeric = Pipeline([
    ('impute', SimpleImputer(strategy='median')),
    ('scale', StandardScaler()),
```

```
])
categorical = Pipeline([
    ('impute', SimpleImputer(strategy='most_frequent')),
    ('encode', OneHotEncoder(handle_unknown='ignore')),
])

preprocess = ColumnTransformer([
    ('num', numeric, num_cols),
    ('cat', categorical, cat_cols),
])

model = Pipeline([
    ('prep', preprocess),
    ('clf', LogisticRegression(max_iter=1000, random_state=42)),
])

model.fit(X_train, y_train)
print('fitted: %.3f' % model.score(X_test, y_test))
```

```
fitted: 0.780
```

Twelve lines of setup and one `fit`. Everything inside learns from the training data only, because the pipeline passes the split through in the right order for you.

Step 4: measure it against the floor

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.dummy import DummyClassifier
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, roc_auc_score, confusion_matrix

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
num_cols = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
cat_cols = ['contract', 'internet_service', 'payment_method', 'has_dependents']
X, y = df[num_cols + cat_cols], df['churned']
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, stratify=y, random_state=42)

preprocess = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), num_cols),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('e', OneHotEncoder(handle_unknown='ignore'))]), cat_cols),
])
model = Pipeline([('prep', preprocess),
                  ('clf', LogisticRegression(max_iter=1000, random_state=42))])
```

```
model.fit(X_train, y_train)

baseline = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)

pred = model.predict(X_test)
proba = model.predict_proba(X_test)[:, 1]

print('baseline accuracy %.3f' % accuracy_score(y_test, baseline.predict(X_test)))
print('model accuracy    %.3f' % accuracy_score(y_test, pred))
print('model ROC AUC     %.3f' % roc_auc_score(y_test, proba))
print('\nconfusion matrix (rows = truth, cols = prediction)')
print(confusion_matrix(y_test, pred))
```

```
baseline accuracy 0.732
model accuracy    0.780
model ROC AUC     0.818

confusion matrix (rows = truth, cols = prediction)
[[499  50]
 [115  86]]
```

Watch Out

78 percent sounds good until you read the matrix

Predicting "nobody churns" scores 73.2 percent. Our model scores 78.0, just under five points of real improvement, not the triumph the raw number suggests. And the matrix shows where it goes wrong: of 201 customers who actually churned, it caught 86 and missed 115. It finds fewer than half of the people you most wanted to find.

This is not a broken model. An AUC of 0.818 means it ranks customers by risk genuinely well. It is a badly chosen *threshold*. The model outputs a probability, and something has to decide where to cut. The default cut is 0.5, which is almost never the right choice when one class is rarer than the other. Week 7 is about fixing exactly this.

Step 5: what did it learn

```
import pandas as pd, numpy as np
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
num_cols = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
cat_cols = ['contract', 'internet_service', 'payment_method', 'has_dependents']
X, y = df[num_cols + cat_cols], df['churned']
X_train, X_test, y_train, y_test = train_test_split(
```

```
    X, y, test_size=0.25, stratify=y, random_state=42)
preprocess = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), num_cols),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('e', OneHotEncoder(handle_unknown='ignore'))]), cat_cols),
])
model = Pipeline([('prep', preprocess),
                  ('clf', LogisticRegression(max_iter=1000, random_state=42))])
model.fit(X_train, y_train)

names = model.named_steps['prep'].get_feature_names_out()
coefs = model.named_steps['clf'].coef_[0]
effect = pd.Series(coefs, index=names).sort_values()

print('pushes towards staying:')
print(effect.head(3).round(2))
print('\npushes towards churning:')
print(effect.tail(3).round(2))
```

```
pushes towards staying:
cat__contract_Two Year            -1.44
num__tenure_months                -0.81
cat__payment_method_Credit card   -0.20
dtype: float64

pushes towards churning:
cat__payment_method_Electronic check    0.46
num__monthly_charges                    0.48
cat__contract_Month-To-Month            1.31
dtype: float64
```

Compare that against the generator you wrote on day 3: month-to-month contracts push churn up, two-year contracts and long tenure push it down, electronic-check payment pushes it up. The model recovered the rule that created the data. That is what a good fit looks like.

Your assignment

Swap `LogisticRegression` for `RandomForestClassifier(n_estimators=300, random_state=42)`, a one-line change, because both obey the same contract. Record accuracy and AUC. You will find the forest scores *worse* on AUC. Write down why you think that is; week 6 gives the answer, and it is not that you did something wrong.

> **Day 7 takeaway**

Self-Assessment

Try It Yourself

1. What makes a problem *supervised*?

   a) The data is clean
   b) Every training row carries the answer you want to predict
   c) A human checks the output
   d) The model is trained on a server

2. Why is a NumPy operation on a whole array preferred to a Python loop over it?

   a) Loops cannot handle floats
   b) It uses less memory
   c) The loop is written in C
   d) The operation runs as compiled code over contiguous memory instead of one interpreted step per element

3. `total_charges` arrives as an `object` column rather than a number. The most likely cause is:

   a) pandas cannot read decimals
   b) The column has too many rows
   c) The file is corrupt
   d) Some rows hold a non-numeric value such as a blank

4. In the scikit-learn contract, what does a trailing underscore on an attribute such as `mean_` signify?

   a) It was learned from data during `fit`
   b) It is deprecated
   c) It is a hyperparameter you set
   d) It is private

5. Why must `fit` never be called on the test set?

   a) It is slower
   b) The test set is smaller
   c) The statistics it learns would carry information from data you are pretending not to have seen
   d) scikit-learn raises an error

6. A churn model reports 74% accuracy. Before celebrating you should first check:

   a) How long it took to fit
   b) The learning rate
   c) What accuracy you get by always predicting the majority class
   d) The number of features

7. What does a `Pipeline` actually buy you?

a) Automatic hyperparameter tuning

b) Prettier code

c) Faster fitting

d) Every preprocessing step is refitted inside each cross-validation fold, on that fold's training rows only

8. `df.groupby('contract')['churned'].mean()` returns:

a) The churn rate within each contract type

b) The mean contract length

c) A count of contract types

d) The number of churners per contract type

Answers:1b, 2d, 3d, 4a, 5c, 6c, 7d, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 2

Data Wrangling and Exploratory Analysis

Week 2

OBJECTIVES

Load, reshape, clean and profile data properly, the work that decides whether a model has anything to learn from.

Most of the difference between a model that works and one that does not is made before any model is fitted. This week covers the loading traps that corrupt data silently, the reshaping and joining that gets it into modelling form without duplicating your target, the three mechanisms behind missing values and what each demands, and how to turn all of it into a report you run on every new extract.

Day 1 Loading Data That Fights Back

Encodings, dtypes, sentinel values and the missing-value list you did not know about

Week 1 read a CSV in one line and it worked. That is the exception. Real extracts arrive with the wrong encoding, dates as text, numbers with currency symbols, and a sentinel value someone chose in 1998 to mean “unknown”. Loading is where most silent errors enter.

pandas has opinions about what missing means

You met this in week 1 without knowing it. `read_csv` converts a list of strings to `NaN` automatically, and that list is longer than you would guess.

```
import pandas as pd
from pandas.io.parsers.readers import STR_NA_VALUES

print(sorted(STR_NA_VALUES))
```

```
['', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN',
    ↪ '<NA>', 'N/A', 'NA', 'NULL', 'NaN', 'None', 'n/a', 'nan', 'null']
```

Watch Out

'None' is on that list

A category legitimately called `None`: a customer with no internet service, a product with no discount, is read as a missing value. The column silently gains hundreds of gaps that were never in the file. This is a real defect that appeared while building this course's dataset,

and it is why the category is spelled `No internet`.

Two arguments control it. `keep_default_na=False` switches the whole list off; `na_values=` adds your own sentinels.

```
import pandas as pd
import io as _io

raw = 'service,code\nNone,1\nDSL,-999\nFibre,2\n'

default = pd.read_csv(_io.StringIO(raw))
print('default:')
print(default)

explicit = pd.read_csv(_io.StringIO(raw), keep_default_na=False,
                       na_values=['-999'])
print('\nkeeping None as text, treating -999 as missing:')
print(explicit)
```

```
default:
  service  code
0     NaN     1
1     DSL  -999
2   Fibre     2

keeping None as text, treating -999 as missing:
  service  code
0    None   1.0
1     DSL   NaN
2   Fibre   2.0
```

Declare the types you expect

Letting pandas guess the dtype of every column is convenient and occasionally wrong. An identifier of digits becomes an integer and loses its leading zeros; a postcode becomes a float. Declare what matters.

```
import pandas as pd
import io as _io

raw = 'account,postcode\n007831,01234\n004120,00987\n'

print('guessed:')
print(pd.read_csv(_io.StringIO(raw)))

print('\ndeclared:')
print(pd.read_csv(_io.StringIO(raw), dtype={'account': str, 'postcode': str}))
```

```
guessed:
   account  postcode
0     7831      1234
1     4120       987
```

```
declared:
  account postcode
0  007831    01234
1  004120    00987
```

The leading zeros are gone in the first version and there is no way to get them back afterwards. Loss at load time is unrecoverable, which is why loading deserves more care than it usually gets.

Parse dates at load time

```
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
print(df['signup_date'].dtype)
print(df['signup_date'].min(), 'to', df['signup_date'].max())
```

```
datetime64[ns]
2019-01-01 00:00:00 to 2024-12-01 00:00:00
```

When the format is ambiguous

`03/04/2024` is the third of April in Britain and the fourth of March in America, and pandas will pick one. If your source is not ISO format, pass the format explicitly: `pd.to_datetime(col, format='%d/%m/%Y')`. Guessing wrong produces dates that are valid, plausible and wrong for eleven months of the year.

Memory, and when it starts to matter

```
import pandas as pd

df = pd.read_csv('customers.csv')
before = df.memory_usage(deep=True).sum()

for col in ['contract', 'internet_service', 'payment_method']:
    df[col] = df[col].astype('category')

after = df.memory_usage(deep=True).sum()
print('before %6.1f KB' % (before / 1024))
print('after  %6.1f KB' % (after / 1024))
print('saved  %5.1f%%' % (100 * (1 - after / before)))
```

```
before 1322.7 KB
after   782.4 KB
saved   40.8%
```

A category column stores each distinct value once and an integer code per row. With three distinct values across three thousand rows the saving is large, and it grows with the row count. On a dataset that fits comfortably in memory this is housekeeping; on one that does not, it is the difference between working and not.

Day 1 takeaway

Day 2 Reshaping and Joining

Long against wide, and how a careless join duplicates your target

Data almost never arrives in the shape a model wants. Models want one row per observation and one column per feature. Sources give you one row per event, or one column per month, or the information split across three tables.

Long and wide

```
import pandas as pd

wide = pd.DataFrame({
    'customer_id': ['C1', 'C2'],
    'jan': [40.0, 55.0],
    'feb': [42.0, 55.0],
    'mar': [41.5, 60.0],
})
print('wide -- readable, not modellable')
print(wide)

long = wide.melt(id_vars='customer_id', var_name='month',
                 value_name='charge')
print('\nlong -- one row per observation')
print(long)
```

```
wide -- readable, not modellable
  customer_id   jan   feb   mar
0          C1  40.0  42.0  41.5
1          C2  55.0  55.0  60.0

long -- one row per observation
  customer_id month  charge
0          C1   jan    40.0
1          C2   jan    55.0
2          C1   feb    42.0
3          C2   feb    55.0
4          C1   mar    41.5
5          C2   mar    60.0
```

`melt` goes wide to long, `pivot` goes back:

```
import pandas as pd

long = pd.DataFrame({
    'customer_id': ['C1', 'C1', 'C1', 'C2', 'C2', 'C2'],
    'month': ['jan', 'feb', 'mar'] * 2,
    'charge': [40.0, 42.0, 41.5, 55.0, 55.0, 60.0],
})

```

```python
back = long.pivot(index='customer_id', columns='month', values='charge')
print(back)

# Aggregating while pivoting, when there are duplicates per cell
summary = long.pivot_table(index='customer_id', values='charge',
                           aggfunc=['mean', 'max'])
print('\n', summary)
```

```
month         feb   jan   mar
customer_id
C1           42.0  40.0  41.5
C2           55.0  55.0  60.0

                  mean    max
               charge charge
customer_id
C1           41.166667   42.0
C2           56.666667   60.0
```

Which shape do you want

Long is right for storage and for plotting with seaborn. Wide is right for modelling, because a model needs a fixed set of columns. Converting long event logs into wide per-customer features is exactly what week 8 calls feature engineering.

Joining tables

The merge types are the same four you know from SQL, and the argument that matters most is `how`.

how	Keeps	Use when
`inner`	Rows matching in both	You need complete records only
`left`	All left rows, matched right ones	Enriching a base table, the usual choice
`outer`	Everything from both	Reconciling two sources
`cross`	Every combination	Building a grid; rarely what you want

```python
import pandas as pd

customers = pd.DataFrame({
    'customer_id': ['C1', 'C2', 'C3'],
    'contract': ['Month-to-month', 'Two year', 'One year'],
})
tickets = pd.DataFrame({
    'customer_id': ['C1', 'C1', 'C3', 'C9'],
    'issue': ['billing', 'speed', 'billing', 'billing'],
})

left = customers.merge(tickets, on='customer_id', how='left')
print(left)
print('\nrows before %d, after %d' % (len(customers), len(left)))
```

```
  customer_id        contract    issue
0          C1  Month-to-month  billing
1          C1  Month-to-month    speed
2          C2        Two year      NaN
3          C3        One year  billing

rows before 3, after 4
```

Watch Out

A left join can multiply your rows

C1 has two tickets, so C1 appears twice. Three customers became four rows. Join a one-per-customer table to a many-per-customer table and your target variable is now duplicated, which silently weights those customers more heavily in training. Aggregate the many side *first*, then join.

```
import pandas as pd

customers = pd.DataFrame({
    'customer_id': ['C1', 'C2', 'C3'],
    'contract': ['Month-to-month', 'Two year', 'One year'],
})
tickets = pd.DataFrame({
    'customer_id': ['C1', 'C1', 'C3', 'C9'],
    'issue': ['billing', 'speed', 'billing', 'billing'],
})

per_customer = tickets.groupby('customer_id').agg(
    ticket_count=('issue', 'size'),
    billing_tickets=('issue', lambda s: (s == 'billing').sum()),
)

safe = customers.merge(per_customer, on='customer_id', how='left')
safe[['ticket_count', 'billing_tickets']] = (
    safe[['ticket_count', 'billing_tickets']].fillna(0).astype(int))
print(safe)
print('\nrows: %d -- unchanged' % len(safe))
```

```
  customer_id        contract  ticket_count  billing_tickets
0          C1  Month-to-month             2                1
1          C2        Two year             0                0
2          C3        One year             1                1

rows: 3 -- unchanged
```

Three rows in, three rows out, and two new features. That is the shape every join into a modelling table should have.

Validate the join instead of hoping

```
import pandas as pd
```

```

left_tbl = pd.DataFrame({'k': ['a', 'b', 'c'], 'x': [1, 2, 3]})
right_tbl = pd.DataFrame({'k': ['a', 'a', 'b'], 'y': [9, 8, 7]})

try:
    left_tbl.merge(right_tbl, on='k', how='left', validate='one_to_one')
except Exception as e:
    print(type(e).__name__ + ':', e)
```

```
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

`validate` accepts `one_to_one`, `one_to_many`, `many_to_one` and `many_to_many`. Stating what you expect turns a silent row explosion into an immediate error.

Day 2 takeaway

Day 3 Missing Data, Properly

Why a value is missing decides how to fill it, and where to fit the imputer

Week 1 counted the gaps and moved on. Today you decide what to do about them, and the right answer depends on *why* they are missing.

Three mechanisms

Mechanism	Meaning	Example	Safe to impute?
MCAR	Missing completely at random	A sensor dropped packets	Yes
MAR	Missingness explained by other columns	Older customers skip the email field	Yes, using those columns
MNAR	Missingness depends on the missing value itself	High earners decline to state income	No, imputing biases it

You cannot test for MNAR from the data alone, which is the uncomfortable part. What you can do is check whether missingness relates to anything you *can* see, and record the fact that a value was missing.

Is the missingness informative?

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

gap = df['has_dependents'].isna()
print('rows with the field blank: %d' % gap.sum())
print('churn rate when blank    : %.3f' % df.loc[gap, 'churned'].mean())
print('churn rate when present  : %.3f' % df.loc[~gap, 'churned'].mean())
```

```
rows with the field blank: 180
churn rate when blank    : 0.272
churn rate when present  : 0.268
```

Close enough to call it uninformative here, which is what you would expect, because the generator removed those values at random. Had the two rates differed sharply, the fact of the gap would itself be a feature worth keeping.

Keep the flag anyway

Adding an `was_missing` indicator column costs one bit per row and occasionally carries real signal: in medical data, a test that was not ordered tells you what the clinician was thinking. `SimpleImputer(add_indicator=True)` does it for you, inside the pipeline.

What each strategy actually does to the distribution

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
from sklearn.impute import SimpleImputer
import numpy as np

col = df[['monthly_charges']]
truth = col.dropna()['monthly_charges']
print('observed  : mean %.2f  std %.2f  n %d'
      % (truth.mean(), truth.std(), len(truth)))

for strategy in ['mean', 'median', 'most_frequent']:
    filled = SimpleImputer(strategy=strategy).fit_transform(col).ravel()
    print('%-14s: mean %.2f  std %.2f  n %d'
          % (strategy, filled.mean(), filled.std(), len(filled)))
```

```
observed  : mean 59.41  std 23.47  n 2835
mean          : mean 59.41  std 22.81  n 3000
median        : mean 59.62  std 22.83  n 3000
most_frequent : mean 56.96  std 24.96  n 3000
```

Watch Out

Imputation always shrinks the variance

Every value filled with the mean or median sits exactly at the centre, so the spread of the column falls, 23.47 to 22.81 here. `most_frequent` is different: the mode is not the centre, so it can widen the spread instead, as it does above. With 5 percent missing either is a rounding error. With 40 percent it is a serious distortion, and any model that relies on the variance of that feature is being misled. Check how much you are filling before you decide it is fine.

Using the other columns

A customer's charge is not independent of their internet service. Filling from the group is better than filling from the whole column.

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

overall = df['monthly_charges'].median()
by_service = df.groupby('internet_service')['monthly_charges'].median()
print('overall median   %.2f' % overall)
print(by_service.round(2))

smart = df['monthly_charges'].fillna(
    df.groupby('internet_service')['monthly_charges'].transform('median'))
print('\nremaining gaps:', smart.isna().sum())
```

```
overall median   63.21
internet_service
DSL            54.97
Fibre optic    79.71
No internet    20.69
Name: monthly_charges, dtype: float64

remaining gaps: 0
```

Filling a fibre customer's charge with the overall median understates it by a wide margin. Grouping first respects a relationship you already know exists.

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler
import numpy as np

num = df[['tenure_months', 'monthly_charges', 'support_calls']]
scaled = StandardScaler().fit_transform(num)
filled = KNNImputer(n_neighbors=5).fit_transform(scaled)

gaps = num['monthly_charges'].isna().values
back = StandardScaler().fit(num.dropna()).inverse_transform(filled)
print('imputed values, first five:')
print(np.round(back[gaps, 1][:5], 2))
```

```
imputed values, first five:
[55.45 54.23 54.45 81.7  55.19]
```

KNN imputation must be scaled first

It finds neighbours by distance, and distance is meaningless across columns measured in months and pounds. Scale, impute, and only then reverse the scaling. Getting this order wrong gives you neighbours chosen almost entirely by whichever column has the largest numbers.

The rule that matters more than the method

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

X = df[['tenure_months', 'monthly_charges']]
y = df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          random_state=42)

# Right: learn the median on train, apply it to both.
imp = SimpleImputer(strategy='median').fit(X_tr)
print('median learned from train only: %.2f' % imp.statistics_[1])

# Wrong: learn it from everything, including rows you will test on.
leaky = SimpleImputer(strategy='median').fit(X)
print('median learned from everything: %.2f' % leaky.statistics_[1])
```

```
median learned from train only: 63.05
median learned from everything: 63.21
```

The two numbers are close, and that is exactly what makes the mistake hard to catch. The leak is real regardless of its size, it grows with the proportion missing, and it always flatters the test score. Fit imputers on training data only, which pipelines enforce for you.

> **Day 3 takeaway**

Day 4 Outliers, Skew and Robustness

Finding extreme values, deciding what they mean, and which models care

An outlier is not a value that is large. It is a value that does not belong to the process you are modelling, and telling those apart requires knowing the domain, not just the numbers.

Find them three ways

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
```

```python
import numpy as np

x = df['support_calls']

# 1. Standard deviations from the mean
z = (x - x.mean()) / x.std()
print('|z| > 3      :', (z.abs() > 3).sum())

# 2. Interquartile range, the boxplot rule
q1, q3 = x.quantile([0.25, 0.75])
iqr = q3 - q1
print('outside 1.5 IQR:', ((x < q1 - 1.5 * iqr) | (x > q3 + 1.5 * iqr)).sum())

# 3. Just look at the extremes
print('largest values :', np.sort(x.unique())[-5:])
```

```
|z| > 3      : 1
outside 1.5 IQR: 8
largest values : [ 4  5  6  7 97]
```

Watch Out

The z-score method is broken by the thing it looks for

A z-score uses the mean and standard deviation, and one extreme value drags both. The outlier inflates the standard deviation, which shrinks its own z-score, which can push it under the threshold. The IQR rule uses quartiles, which a single extreme value cannot move. That is what *robust* means and it is why the boxplot rule is the safer default.

What to do about the one customer with 97 calls

```python
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

extreme = df.loc[df['support_calls'] > 20,
                 ['customer_id', 'tenure_months', 'support_calls',
                  'monthly_charges', 'churned']]
print(extreme)
print('\nsupport_calls: mean %.2f, median %.1f, max %d'
      % (df['support_calls'].mean(), df['support_calls'].median(),
         df['support_calls'].max()))
```

```
    customer_id  tenure_months  support_calls  monthly_charges  churned
710      C00008             48             97            76.42        0

support_calls: mean 1.27, median 1.0, max 97
```

One row. It is not a typo. A customer really can call ninety-seven times, and if anything they are the most interesting customer in the file. Deleting them removes a genuine, if rare, pattern. The options, in order of how much they assume:

1. **Leave it** and use a model that does not care. Trees split on order, not magnitude, so an extreme value changes nothing.
2. **Clip it** to a percentile, caps the influence without discarding the row.
3. **Transform** the column with a log, which compresses the long tail.
4. **Drop it**, only when you can show it is an error.

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
import numpy as np

x = df['support_calls']
cap = x.quantile(0.99)

print('original : max %5.1f  mean %.3f  std %.3f' % (x.max(), x.mean(), x.std()))
print('clipped  : max %5.1f  mean %.3f  std %.3f'
      % (x.clip(upper=cap).max(), x.clip(upper=cap).mean(),
    ↪ x.clip(upper=cap).std()))
print('log1p    : max %5.1f  mean %.3f  std %.3f'
      % (np.log1p(x).max(), np.log1p(x).mean(), np.log1p(x).std()))
```

```
original : max  97.0  mean 1.267  std 2.100
clipped  : max   5.0  mean 1.233  std 1.156
log1p    : max   4.6  mean 0.672  std 0.527
```

log1p, not log

`np.log1p(x)` computes $\log(1 + x)$, which is defined at zero. Plain `np.log` on a column containing zeros gives you `-inf`, and one infinity poisons every calculation downstream. Count columns almost always contain zeros.

Which models care

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
import numpy as np

X = df[['tenure_months', 'support_calls', 'monthly_charges']]
y = df['churned']
X_capped = X.copy()
X_capped['support_calls'] = X_capped['support_calls'].clip(
```

```
    upper=X_capped['support_calls'].quantile(0.99))

models = {
    'logistic': make_pipeline(SimpleImputer(strategy='median'),
                              StandardScaler(),
                              LogisticRegression(max_iter=1000)),
    'tree    ': make_pipeline(SimpleImputer(strategy='median'),
                              DecisionTreeClassifier(max_depth=4,
                                                     random_state=42)),
}
for name, m in models.items():
    raw = cross_val_score(m, X, y, cv=5, scoring='roc_auc').mean()
    cap = cross_val_score(m, X_capped, y, cv=5, scoring='roc_auc').mean()
    print('%s  raw %.4f   capped %.4f   change %+.4f'
          % (name, raw, cap, cap - raw))
```

```
logistic  raw 0.7762   capped 0.7786   change +0.0024
tree      raw 0.7545   capped 0.7545   change +0.0000
```

The tree score is identical to four decimal places, because a tree only ever asks "is this value above the split point" and 97 answers that exactly as 20 does. The linear model moves, because after scaling that row sits many standard deviations out and pulls the fitted coefficient toward itself. The movement is small here, one contaminated row in three thousand, and that is the honest lesson: the direction is reliable, the size depends entirely on how much of your data is affected.

Skew, and why it matters for linear models

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
import numpy as np

for col in ['tenure_months', 'monthly_charges', 'support_calls']:
    s = df[col].dropna()
    print('%-16s skew %6.2f   log1p skew %6.2f'
          % (col, s.skew(), np.log1p(s).skew()))
```

```
tenure_months    skew   1.22   log1p skew  -0.26
monthly_charges  skew  -0.47   log1p skew  -1.15
support_calls    skew  31.75   log1p skew   0.14
```

A skew near zero means symmetric. `support_calls` is extremely right-skewed at 31.75, and the log transform pulls it to 0.14. But look at `monthly_charges`: already near-symmetric at -0.47, and the log makes it distinctly *worse* at -1.15. A log is a tool for right-skewed columns, not a default. Linear models benefit from roughly symmetric inputs; tree models do not care either way.

Day 4 takeaway

Day 5 Dates and Time Features

Decomposition, elapsed time, cyclical encoding, and the split dates force

A date column is useless to a model as it stands. No algorithm can do anything with the integer 1704067200. What models can use is what the date *means*: how long ago, which month, which day of the week.

The accessor

```
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
d = df['signup_date']

parts = pd.DataFrame({
    'date': d,
    'year': d.dt.year,
    'month': d.dt.month,
    'quarter': d.dt.quarter,
    'dayofweek': d.dt.dayofweek,
    'is_month_start': d.dt.is_month_start,
})
print(parts.head())
```

```
        date  year  month  quarter  dayofweek  is_month_start
0 2024-03-01  2024      3        1          4            True
1 2024-05-01  2024      5        2          2            True
2 2024-10-01  2024     10        4          1            True
3 2024-10-01  2024     10        4          1            True
4 2023-04-01  2023      4        2          5            True
```

Elapsed time is usually the feature you want

```
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])

# A fixed reference, not today's date -- see the warning below.
as_of = pd.Timestamp('2025-01-01')
df['days_since_signup'] = (as_of - df['signup_date']).dt.days
df['years_since_signup'] = df['days_since_signup'] / 365.25

print(df[['signup_date', 'days_since_signup', 'years_since_signup']]
      .head().round(2))
```

```
  signup_date  days_since_signup  years_since_signup
0  2024-03-01                306                0.84
1  2024-05-01                245                0.67
2  2024-10-01                 92                0.25
3  2024-10-01                 92                0.25
4  2023-04-01                641                1.75
```

Watch Out

Never build a feature from today's date

`pd.Timestamp.now()` gives a different answer every day you run it. The model trained in March sees different numbers in June, and nothing in your code changed. Pass a fixed reference date, store it with the model, and use the same one at prediction time. This is one of the most common causes of a model that silently degrades in production.

Cyclical features

Month 12 and month 1 are adjacent, but as numbers they are eleven apart. A linear model reads December and January as maximally different. The fix is to place the value on a circle.

```
import numpy as np
import pandas as pd

months = pd.Series(range(1, 13))
angle = 2 * np.pi * (months - 1) / 12
cyc = pd.DataFrame({
    'month': months,
    'sin': np.sin(angle).round(3),
    'cos': np.cos(angle).round(3),
})
print(cyc)

def gap(a, b):
    ra = cyc.loc[cyc['month'] == a, ['sin', 'cos']].values[0]
    rb = cyc.loc[cyc['month'] == b, ['sin', 'cos']].values[0]
    return np.linalg.norm(ra - rb)

print('\nraw distance Dec to Jan: %d' % abs(12 - 1))
print('raw distance Jun to Jul: %d' % abs(6 - 7))
print('circular Dec to Jan: %.3f' % gap(12, 1))
print('circular Jun to Jul: %.3f' % gap(6, 7))
```

```
    month    sin    cos
0       1  0.000  1.000
1       2  0.500  0.866
2       3  0.866  0.500
3       4  1.000  0.000
4       5  0.866 -0.500
5       6  0.500 -0.866
6       7  0.000 -1.000
7       8 -0.500 -0.866
8       9 -0.866 -0.500
9      10 -1.000 -0.000
10     11 -0.866  0.500
11     12 -0.500  0.866

raw distance Dec to Jan: 11
raw distance Jun to Jul: 1
circular Dec to Jan: 0.518
circular Jun to Jul: 0.518
```

On the circle, December to January is the same distance as June to July, which is the truth. Use the same trick for hour of day and day of week.

The split that dates force on you

Watch Out

A random split leaks the future

If your rows are ordered in time and you split at random, the model trains on March and tests on February. It has seen the future. Every score is inflated, sometimes enormously, and the failure only shows up in production. For anything time-ordered, split by date, and scikit-learn gives you `TimeSeriesSplit` for cross-validation.

```
import numpy as np
from sklearn.model_selection import TimeSeriesSplit

x = np.arange(12)
for i, (train_idx, test_idx) in enumerate(TimeSeriesSplit(n_splits=4).split(x), 1):
    print('fold %d  train %-18s test %s'
          % (i, str(x[train_idx]), str(x[test_idx])))
```

```
fold 1  train [0 1 2 3]          test [4 5]
fold 2  train [0 1 2 3 4 5]      test [6 7]
fold 3  train [0 1 2 3 4 5 6 7]  test [8 9]
fold 4  train [0 1 2 3 4 5 6 7 8 9] test [10 11]
```

Each fold trains only on rows that came before the ones it tests on, and the training window grows. No fold ever sees data from after its test period, which is the only honest way to score a time-ordered problem.

Day 5 takeaway

Day 6 Categories and Text

Cardinality, ordinal against nominal, unseen levels and rare levels

Categories and free text are where most of the information in business data lives, and where most of the leakage and most of the silent breakage comes from too.

Cardinality decides the treatment

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

for col in df.select_dtypes('object').columns:
    print('%-18s %5d distinct  %5.1f%% of rows'
```

```
          % (col, df[col].nunique(), 100 * df[col].nunique() / len(df)))
```

```
customer_id          3000 distinct  100.0% of rows
signup_date            72 distinct    2.4% of rows
contract                3 distinct    0.1% of rows
internet_service        3 distinct    0.1% of rows
payment_method          4 distinct    0.1% of rows
has_dependents          2 distinct    0.1% of rows
```

Distinct values	Treatment	Why
2	Map to 0/1	One column is enough
3 to about 15	One-hot encode	Manageable width, no false ordering
15 to a few hundred	Group the rare ones, then one-hot	A level seen twice cannot be learned from
Thousands	Target or hashing encoding	One-hot would be wider than the dataset is long
Unique per row	Drop it	An identifier is not a feature

Watch Out

customer_id is not a feature

It is unique per row, so a sufficiently flexible model can memorise the target for every training row through it and score perfectly. On new customers it knows nothing. Drop identifiers explicitly rather than trusting yourself to remember.

Ordinal is not the same as categorical

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder

sat = pd.DataFrame({'rating': ['low', 'high', 'medium', 'low']})

order = OrdinalEncoder(categories=[['low', 'medium', 'high']])
print('ordinal, order stated:')
print(order.fit_transform(sat).ravel())

print('\nordinal, order left to chance (alphabetical):')
print(OrdinalEncoder().fit_transform(sat).ravel())
```

```
ordinal, order stated:
[0. 2. 1. 0.]

ordinal, order left to chance (alphabetical):
[1. 0. 2. 1.]
```

Left to itself the encoder sorts alphabetically, making *high* 0, *low* 1 and *medium* 2. An ordering that says medium is more than low is more than high. If a variable has a genuine order, state it. If it does not, do not use an ordinal encoder at all.

One-hot, and the argument you must not omit

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

train = pd.DataFrame({'contract': ['Month-to-month', 'One year', 'Two year']})
new = pd.DataFrame({'contract': ['Two year', 'Weekly trial']})

enc = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
enc.fit(train)
print('columns:', list(enc.get_feature_names_out()))
print('unseen category encodes as all zeros:')
print(enc.transform(new))
```

```
columns: ['contract_Month-to-month', 'contract_One year', 'contract_Two year']
unseen category encodes as all zeros:
[[0. 0. 1.]
 [0. 0. 0.]]
```

Watch Out

Without handle_unknown='ignore' this raises

Marketing launches a Weekly trial contract, your scoring job meets a category the encoder has never seen, and it throws. At three in the morning. `handle_unknown='ignore'` encodes the unknown as all zeros, which is the sane default: the model treats it as none of the known categories rather than falling over.

Rare categories

```
import pandas as pd
import numpy as np

rng = np.random.default_rng(0)
cities = (['London'] * 400 + ['Manchester'] * 250 + ['Leeds'] * 120 +
          ['Bristol'] * 60 + ['Hull'] * 8 + ['Truro'] * 5 + ['Kirkwall'] * 2)
s = pd.Series(rng.permutation(cities), name='city')

counts = s.value_counts()
print(counts)

keep = counts[counts >= 50].index
grouped = s.where(s.isin(keep), 'Other')
print('\nafter grouping rare levels:')
print(grouped.value_counts())
```

```
city
London        400
Manchester    250
Leeds         120
Bristol        60
Hull            8
```

```
Truro           5
Kirkwall        2
Name: count, dtype: int64

after grouping rare levels:
city
London        400
Manchester    250
Leeds         120
Bristol        60
Other          15
Name: count, dtype: int64
```

Kirkwall appears twice. A one-hot column that is 1 for two rows out of a thousand cannot support a reliable estimate, and in cross-validation it may be entirely absent from some folds. Grouping the tail into *Other* keeps the information that these are unusual without pretending you can learn each one.

min_frequency does this for you

`OneHotEncoder(min_frequency=50)` groups anything rarer into a single infrequent bucket, and it learns the threshold on training data only. Prefer it to a manual pass, because doing it by hand before the split means the decision was made using test rows.

Text, briefly

```
from sklearn.feature_extraction.text import TfidfVectorizer

notes = [
    'billing error on the latest invoice',
    'speed very slow in the evening',
    'invoice wrong again, second billing error',
    'router keeps dropping the connection',
]
vec = TfidfVectorizer(stop_words='english', min_df=1)
X = vec.fit_transform(notes)
print('shape:', X.shape)
print('vocabulary:', sorted(vec.vocabulary_)[:10])
print('\nrow 0 non-zero terms:')
names = vec.get_feature_names_out()
row = X[0].toarray().ravel()
for i in row.argsort()[:-1][:4]:
    if row[i] > 0:
        print('  %-12s %.3f' % (names[i], row[i]))
```

```
shape: (4, 13)
vocabulary: ['billing', 'connection', 'dropping', 'error', 'evening', 'invoice', 'keeps',
    ↪ 'latest', 'router', 'second']

row 0 non-zero terms:
```

TF-IDF weights a term by how often it appears in this document against how rare it is across all of them, so common words score low and distinctive ones score high. It produces a sparse matrix

that goes straight into any scikit-learn model. Week 13 covers the sequence models that improve on it.

> **Day 6 takeaway**

Day 7 A Reproducible Data Quality Report

Turning a week of manual checks into one function you run on every extract

Exploration produces understanding, and understanding evaporates. Today you turn a week of scattered checks into one function you can run against any new extract.

What a data quality report should answer

1. What shape is it, and did that change since last time?
2. Which columns have the wrong type?
3. Where are the gaps, and how big?
4. Which columns are constant, or nearly so?
5. Which are duplicated in content under different names?
6. Are there duplicate rows?
7. What is the target's base rate?

The report

```
import pandas as pd
import numpy as np

def profile(df, target=None):
    """One row per column, with the checks worth running every time."""
    rows = []
    for col in df.columns:
        s = df[col]
        top = s.value_counts(dropna=True)
        rows.append({
            'column': col,
            'dtype': str(s.dtype),
            'missing_pct': round(100 * s.isna().mean(), 2),
            'distinct': s.nunique(dropna=True),
            'top_value': None if top.empty else str(top.index[0])[:18],
            'top_share': 0.0 if top.empty else round(100 * top.iloc[0] / len(s), 1),
        })
    out = pd.DataFrame(rows)
    out['constant'] = out['distinct'] <= 1
    out['id_like'] = out['distinct'] == len(df)
    return out

df = pd.read_csv('customers.csv')
```

```
print(profile(df).to_string(index=False))
```

```
          column   dtype  missing_pct  distinct        top_value  top_share  constant  id_like
     customer_id  object         0.00      3000           C00507        0.1     False    False
     signup_date  object         0.00        72       2019-01-01        4.2     False    False
   tenure_months   int64         0.00        72               72        4.2     False    False
        contract  object         0.00        15   Month-to-month       51.3     False    False
internet_service  object         0.00         3      Fibre optic       45.1     False    False
  payment_method  object         0.00         4 Electronic check       33.6     False    False
  has_dependents  object         6.09         2               No       65.8     False    False
   support_calls   int64         0.00         9                1       34.1     False    False
 monthly_charges float64         5.45      2232             15.0        5.3     False    False
   total_charges  object         0.00      2969                          0.5     False    False
         churned   int64         0.00         2                0       73.2     False    False
```

The checks that belong beside it

```
import pandas as pd

def integrity(df, target=None):
    print('rows %d, columns %d' % df.shape)
    print('duplicate rows      %d' % df.duplicated().sum())
    obj = df.select_dtypes('object').columns
    coercible = [c for c in obj
                 if pd.to_numeric(df[c], errors='coerce').notna().mean() > 0.9]
    print('text but ~numeric   %s' % (list(coercible) or 'none'))
    padded = [c for c in obj if (df[c].fillna('') !=
    ↪ df[c].fillna('').str.strip()).any()]
    print('has padded values   %s' % (list(padded) or 'none'))
    if target:
        rate = df[target].mean()
        print('target base rate    %.3f  (majority guess scores %.3f)'
              % (rate, max(rate, 1 - rate)))

df = pd.read_csv('customers.csv')
integrity(df, target='churned')
```

```
rows 3138, columns 11
duplicate rows      138
text but ~numeric   ['total_charges']
has padded values   ['contract', 'total_charges']
target base rate    0.268  (majority guess scores 0.732)
```

Every defect week 1 found by hand, found automatically: the numeric column stored as text, the padded category values, the duplicated rows, and the base rate any model has to beat.

Correlated pairs worth knowing about

```
import pandas as pd
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
```

```python
import numpy as np

num = df.select_dtypes('number').drop(columns=['churned'])
corr = num.corr().abs()

# Upper triangle only, so each pair appears once and never against itself.
mask = np.triu(np.ones(corr.shape, dtype=bool), k=1)
pairs = (corr.where(mask).stack().sort_values(ascending=False))
print(pairs[pairs > 0.5].round(3))
```

```
tenure_months  total_charges    0.829
dtype: float64
```

`tenure_months` and `total_charges` at 0.83. Not a reason to delete a column yet. It is a reason to expect unstable coefficients from a linear model, which week 4 will demonstrate and fix with regularisation.

Your assignment

Run `profile` and `integrity` against a dataset you have not seen, anything from your own work, or a public CSV. Write down three things the report told you that you would not have checked by hand. Then add a check of your own for whatever it missed.

> **Day 7 takeaway**

Self-Assessment

Try It Yourself

1. pandas treats several strings as missing by default. Which of these is *not* in that list?

   a) `NULL`
   b) A single space
   c) An empty string
   d) `NA`

2. A customers table is joined to a tickets table and the row count rises. What has happened?

   a) The join failed
   b) Customers with several tickets now appear once per ticket, so any average over customers is now weighted by ticket count
   c) Duplicates were created at random
   d) The join key was numeric

3. Values are missing *because* the customer is new. That is:

   a) Not really missing
   b) Missing completely at random

c) Missing at random

d) Missing not at random. The reason is related to the value itself

4. Which scaler is least disturbed by a single extreme outlier?

   a) `RobustScaler`

   b) `Normalizer`

   c) `StandardScaler`

   d) `MinMaxScaler`

5. Why encode an hour of the day as a sine and cosine pair?

   a) So that 23:00 and 00:00 are close together, which they are not on a plain 0 to 23 scale

   b) To reduce the number of columns

   c) Because trees require it

   d) It is faster

6. A category appears twice in a thousand rows. The safest treatment is:

   a) Drop those rows

   b) Give it its own one-hot column

   c) Group it into an *Other* bucket, learned on training data only

   d) Replace it with the mean

7. `OneHotEncoder(handle_unknown='ignore')` meets a category it never saw in training. It:

   a) Drops the row

   b) Raises an error

   c) Encodes it as a row of zeros

   d) Assigns it to the most common category

8. What does TF-IDF weight a term by?

   a) Its position in the sentence

   b) How often it appears in the whole corpus

   c) Its length

   d) How often it appears in this document against how rare it is across all documents

Answers:1b, 2b, 3d, 4a, 5a, 6c, 7c, 8d. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 3

The Maths That Actually Matters

Week 3

OBJECTIVES

Linear algebra, calculus, probability and statistics, only the parts that change what you do, each one implemented rather than proved.

You do not need a maths degree for machine learning, but four ideas will keep tripping you up until you own them: a model is a matrix multiplication, training is stepping against a gradient, a classifier outputs probabilities that must be read carefully, and every score you measure carries uncertainty. This week implements each rather than proving it, and ends by solving the same regression three ways.

Day 1 Vectors, Matrices and Why Models Are Multiplications

Dot products, matrix shapes, distance, and the one line most models are

You do not need a maths degree to do machine learning well. You do need four ideas, and this week covers them by implementing each one rather than proving anything. Today: why almost every model is, underneath, a matrix multiplication.

A row of data is a vector

One customer, tenure 12, charge 79.50, support calls 2, is a point in three-dimensional space. Two thousand customers are two thousand points. Everything that follows is geometry on those points.

```
import numpy as np

customer = np.array([12.0, 79.50, 2.0])
weights = np.array([-0.05, 0.02, 0.30])

score = np.dot(customer, weights)
manual = sum(c * w for c, w in zip(customer, weights))
print('dot product : %.4f' % score)
print('by hand     : %.4f' % manual)
```

```
dot product : 1.5900
by hand     : 1.5900
```

The whole dataset at once

You never score one customer. You score all of them, and the matrix product does every dot product in one call.

```
import numpy as np

X = np.array([[12.0, 79.50, 2.0],
              [48.0, 55.00, 0.0],
              [ 3.0, 92.00, 5.0],
              [60.0, 21.00, 1.0]])
w = np.array([-0.05, 0.02, 0.30])
b = -1.2

scores = X @ w + b
print('X shape', X.shape, '@ w shape', w.shape, '→ ', scores.shape)
print('scores', scores.round(3))
```

```
X shape (4, 3) @ w shape (3,) →  (4,)
scores [ 0.39 -2.5   1.99 -3.48]
```

That line is a linear model

`X @ w + b` is the entire forward pass of linear regression, and with a sigmoid wrapped round it, of logistic regression. Week 11's neural network is the same line repeated with a nonlinearity between the repeats. Understand this one expression and you understand the shape of most of the course.

Shapes have to line up

A matrix product `(n, k) @ (k, m)` gives `(n, m)`. The inner dimensions must match, and this is the source of most errors you will hit in week 11.

```
import numpy as np

X = np.random.default_rng(0).normal(size=(100, 4))   # 100 rows, 4 features
W1 = np.random.default_rng(1).normal(size=(4, 8))    # layer: 4 in, 8 out
W2 = np.random.default_rng(2).normal(size=(8, 1))    # layer: 8 in, 1 out

h = X @ W1
out = h @ W2
print('X   ', X.shape)
print('X@W1', h.shape)
print('h@W2', out.shape)

try:
    X @ W2
except ValueError as e:
    print('\nmismatch:', e)
```

```
X    (100, 4)
X@W1 (100, 8)
h@W2 (100, 1)
```

```
mismatch: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc
    ↪ signature (n?,k),(k,m?)→(n?,m?) (size 8 is different from 4)
```

Distance, and what it means for a model

kNN, k-means and every clustering algorithm in week 9 rest on measuring how far apart two rows are.

```
import numpy as np

a = np.array([12.0, 79.5])
b = np.array([14.0, 81.0])
c = np.array([60.0, 21.0])

def euclid(p, q):
    return np.sqrt(((p - q) ** 2).sum())

print('a to b: %.2f' % euclid(a, b))
print('a to c: %.2f' % euclid(a, c))
print('numpy : %.2f' % np.linalg.norm(a - c))
```

```
a to b: 2.50
a to c: 75.67
numpy : 75.67
```

Watch Out

Distance is meaningless on unscaled columns

Tenure runs 1 to 72; total charges run to 8000. In the distance calculation, charges dominate entirely and tenure might as well not exist. Every distance-based algorithm requires scaling first, and this is the real reason `StandardScaler` appears in nearly every pipeline in this course.

```
import numpy as np
from sklearn.preprocessing import StandardScaler

raw = np.array([[12.0, 950.0],
                [14.0, 8000.0],
                [60.0, 1000.0]])

print('unscaled: row0-row1 %.1f, row0-row2 %.1f'
      % (np.linalg.norm(raw[0] - raw[1]), np.linalg.norm(raw[0] - raw[2])))

scaled = StandardScaler().fit_transform(raw)
print('scaled  : row0-row1 %.2f, row0-row2 %.2f'
      % (np.linalg.norm(scaled[0] - scaled[1]),
         np.linalg.norm(scaled[0] - scaled[2])))
```

```
unscaled: row0-row1 7050.0, row0-row2 69.3
scaled  : row0-row1 2.13, row0-row2 2.17
```

Unscaled, row 0 looks far closer to row 2 than to row 1, because the charge difference swamps everything. Scaled, the answer flips, and the scaled answer is the one that respects both columns.

Day 1 takeaway

Day 2 Derivatives and Gradients

Slopes, the chain rule, and why the sigmoid derivative caps at 0.25

Training a model means adjusting numbers until a loss gets smaller. To know which direction to adjust, you need the derivative. That is the whole role calculus plays here.

A derivative is a slope

```python
import numpy as np

def f(x):
    return x ** 2 + 3 * x + 5

def numeric_derivative(f, x, h=1e-6):
    return (f(x + h) - f(x - h)) / (2 * h)

# The exact derivative of x^2 + 3x + 5 is 2x + 3.
for x in [-4.0, -1.5, 0.0, 3.0]:
    print('x=%5.1f  numeric %8.4f   exact %8.4f'
          % (x, numeric_derivative(f, x), 2 * x + 3))
```

```
x= -4.0  numeric  -5.0000   exact  -5.0000
x= -1.5  numeric  -0.0000   exact   0.0000
x=  0.0  numeric   3.0000   exact   3.0000
x=  3.0  numeric   9.0000   exact   9.0000
```

At x = -1.5 the derivative is zero, and that is the bottom of the parabola. Finding where the derivative is zero is finding the minimum, which is exactly what training does.

The gradient is the derivative in several directions

Models have thousands of parameters, not one. The gradient collects the derivative with respect to each of them into a vector that points in the direction of steepest increase.

```python
import numpy as np

def loss(w):
    # A bowl, steeper in the second direction than the first.
    return (w[0] - 2) ** 2 + 4 * (w[1] + 1) ** 2

def gradient(f, w, h=1e-6):
    g = np.zeros_like(w)
    for i in range(len(w)):
        step = np.zeros_like(w)
        step[i] = h
```

```
        g[i] = (f(w + step) - f(w - step)) / (2 * h)
    return g

w = np.array([0.0, 0.0])
g = gradient(loss, w)
print('loss at [0,0]  %.3f' % loss(w))
print('gradient       %s' % g.round(4))
print('exact          %s' % np.array([2 * (w[0] - 2), 8 * (w[1] + 1)]))
print('\nstepping against the gradient:')
print('loss at w - 0.1*g  %.3f' % loss(w - 0.1 * g))
```

```
loss at [0,0]  8.000
gradient       [-4.  8.]
exact          [-4.  8.]

stepping against the gradient:
loss at w - 0.1*g  2.720
```

Against the gradient, not along it

The gradient points uphill. To reduce a loss you step in the opposite direction, which is why the update rule has a minus sign: `w = w - learning_rate * gradient`. That one line is gradient descent, and tomorrow you implement it.

The chain rule, which is what backpropagation is

Compose two functions and the derivative multiplies. This is the whole mechanism by which a neural network assigns credit to a weight buried four layers from the output.

```
import numpy as np

def g(x):
    return 3 * x + 1

def f(u):
    return u ** 2

def composed(x):
    return f(g(x))

x = 2.0
# chain rule: d/dx f(g(x)) = f'(g(x)) * g'(x) = 2*g(x) * 3
chain = 2 * g(x) * 3
numeric = (composed(x + 1e-6) - composed(x - 1e-6)) / 2e-6
print('chain rule %.4f' % chain)
print('numeric    %.4f' % numeric)
```

```
chain rule 42.0000
numeric    42.0000
```

The derivatives you will actually meet

Function	Where it appears	Derivative
x^2	Squared error loss	$2x$
sigmoid(x)	Logistic output layer	sigmoid(x)(1 - sigmoid(x))
relu(x)	Hidden layers	1 if $x > 0$, else 0
log(x)	Log loss	1/x

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def relu(x):
    return np.maximum(0, x)

xs = np.array([-2.0, -0.5, 0.0, 0.5, 2.0])
h = 1e-6
print('   x   sigmoid  d/dx numeric  d/dx formula   relu  d/dx')
for x in xs:
    num = (sigmoid(x + h) - sigmoid(x - h)) / (2 * h)
    formula = sigmoid(x) * (1 - sigmoid(x))
    rnum = (relu(x + h) - relu(x - h)) / (2 * h)
    print('%5.1f  %7.4f  %12.6f  %12.6f  %5.1f  %4.1f'
          % (x, sigmoid(x), num, formula, relu(x), rnum))
```

```
   x   sigmoid  d/dx numeric  d/dx formula   relu  d/dx
-2.0   0.1192      0.104994      0.104994    0.0   0.0
-0.5   0.3775      0.235004      0.235004    0.0   0.0
 0.0   0.5000      0.250000      0.250000    0.0   0.5
 0.5   0.6225      0.235004      0.235004    0.5   1.0
 2.0   0.8808      0.104994      0.104994    2.0   1.0
```

Watch Out

The sigmoid derivative peaks at 0.25

Look at the column: the largest value is 0.25, at $x = 0$, and it falls away fast. Multiply several of those together through a deep network and the gradient reaching the early layers is vanishingly small. That is the vanishing gradient problem, it is why sigmoid was abandoned for hidden layers, and it is why ReLU, whose derivative is exactly 1 for positive inputs, replaced it. Week 11 returns to this.

Day 2 takeaway

Day 3 Gradient Descent, Implemented

The four-line loop, and the three ways learning rate and scale break it

Yesterday gave you the direction. Today you take the steps, and watch the three ways this goes wrong.

The algorithm, in full

1. Start with a guess for the parameters.
2. Compute the loss and its gradient.
3. Move the parameters a small step against the gradient.
4. Repeat until the loss stops falling.

```
import numpy as np

rng = np.random.default_rng(0)
n = 200
x = rng.uniform(0, 10, n)
y = 3.5 * x + 2.0 + rng.normal(0, 1.5, n)   # truth: slope 3.5, intercept 2

w, b = 0.0, 0.0
lr = 0.01

for step in range(1, 2001):
    pred = w * x + b
    error = pred - y
    loss = (error ** 2).mean()
    dw = 2 * (error * x).mean()
    db = 2 * error.mean()
    w -= lr * dw
    b -= lr * db
    if step in (1, 10, 100, 500, 2000):
        print('step %4d  loss %8.4f  w %.4f  b %.4f' % (step, loss, w, b))

print('\ntruth       w 3.5000  b 2.0000')
print('recovered   w %.4f  b %.4f' % (w, b))
```

```
step    1  loss 541.8747  w 2.8686  b 0.4151
step   10  loss   2.8682  w 3.6719  b 0.5939
step  100  loss   2.5705  w 3.5999  b 1.1008
step  500  loss   2.3502  w 3.4833  b 1.9210
step 2000  loss   2.3448  w 3.4621  b 2.0705

truth       w 3.5000  b 2.0000
recovered   w 3.4621  b 2.0705
```

Two thousand steps of arithmetic recovered the rule that generated the data. No library, no solver. The same loop, at enormous scale, is what trains a neural network.

The learning rate is the whole game

```
import numpy as np

rng = np.random.default_rng(0)
```

```
x = rng.uniform(0, 10, 200)
y = 3.5 * x + 2.0 + rng.normal(0, 1.5, 200)

def descend(lr, steps=300):
    w = b = 0.0
    for _ in range(steps):
        error = (w * x + b) - y
        w -= lr * 2 * (error * x).mean()
        b -= lr * 2 * error.mean()
        # Overflow passes through astronomically large but still finite
        # values on its way to inf, so test the magnitude, not isfinite.
        if not np.isfinite(w) or abs(w) > 1e10:
            return None, None, None
    return w, b, (((w * x + b) - y) ** 2).mean()

for lr in [0.0001, 0.001, 0.01, 0.03, 0.05]:
    w, b, loss = descend(lr)
    if w is None:
        print('lr %-7s diverged to infinity' % lr)
    else:
        print('lr %-7s w %7.4f  b %7.4f  loss %8.4f' % (lr, w, b, loss))
```

```
lr 0.0001  w  3.3268  b  0.4947  loss   7.8274
lr 0.001   w  3.6533  b  0.7253  loss   2.7752
lr 0.01    w  3.5162  b  1.6897  loss   2.3793
lr 0.03    diverged to infinity
lr 0.05    diverged to infinity
```

Watch Out

Too small wastes time, too large explodes

At 0.0001 the model has barely moved after 300 steps. At 0.05 each step overshoots the minimum by more than it started from, the error grows, and the parameters run to infinity within a few dozen iterations. If your loss becomes `nan` during training, the learning rate is the first thing to look at.

Scaling changes how hard the problem is

```
import numpy as np

rng = np.random.default_rng(1)
n = 500
tenure = rng.uniform(1, 72, n)
charges = rng.uniform(20, 120, n)
total = tenure * charges                    # ranges into the thousands
y = 0.4 * tenure + 0.05 * charges + 0.001 * total + rng.normal(0, 1, n)

def descend(X, y, lr, steps=400):
    # A column of ones so the model can fit an intercept. Without it a
    # scaled matrix has mean zero and cannot reach a non-zero target.
    Xb = np.column_stack([np.ones(len(X)), X])
```

```
    w = np.zeros(Xb.shape[1])
    for _ in range(steps):
        error = Xb @ w - y
        w -= lr * 2 * (Xb.T @ error) / len(y)
        if not np.all(np.isfinite(w)) or np.abs(w).max() > 1e10:
            return None
    return (((Xb @ w) - y) ** 2).mean()

raw = np.column_stack([tenure, charges, total])
scaled = (raw - raw.mean(axis=0)) / raw.std(axis=0)

for lr in [1e-7, 1e-5, 1e-3, 1e-2]:
    r = descend(raw, y, lr)
    s = descend(scaled, y, lr)
    print('lr %-6s  raw %-12s  scaled %s'
          % (lr, 'diverged' if r is None else '%.4f' % r,
             'diverged' if s is None else '%.4f' % s))
```

```
lr 1e-07   raw 38.9770       scaled 518.7263
lr 1e-05   raw diverged      scaled 509.3600
lr 0.001   raw diverged      scaled 92.7342
lr 0.01    raw diverged      scaled 1.4854
```

On raw columns only a tiny learning rate survives, and it converges slowly. On scaled columns a much larger rate works and gets further. Scaling is not cosmetic. It changes the shape of the loss surface from a long thin valley into something closer to a bowl, and gradient descent handles bowls far better.

Batch, stochastic and mini-batch

Variant	Gradient computed on	Trade-off
Batch	Every row, every step	Smooth and accurate, slow on large data
Stochastic (SGD)	One row at a time	Fast and noisy; the noise can escape shallow minima
Mini-batch	32 to 512 rows	The compromise everyone actually uses

```
import numpy as np

rng = np.random.default_rng(0)
n = 2000
x = rng.uniform(0, 10, n)
y = 3.5 * x + 2.0 + rng.normal(0, 1.5, n)

def minibatch(batch_size, epochs=30, lr=0.01, seed=0):
    r = np.random.default_rng(seed)
    w = b = 0.0
    for _ in range(epochs):
        order = r.permutation(n)
        for start in range(0, n, batch_size):
            idx = order[start:start + batch_size]
```

```
            xb, yb = x[idx], y[idx]
            error = (w * xb + b) - yb
            w -= lr * 2 * (error * xb).mean()
            b -= lr * 2 * error.mean()
    return w, b, (((w * x + b) - y) ** 2).mean()

for bs in [1, 32, 256, n]:
    w, b, loss = minibatch(bs)
    label = 'stochastic' if bs == 1 else ('batch' if bs == n else 'mini %d' % bs)
    print('%-12s w %.4f  b %.4f  loss %.4f' % (label, w, b, loss))
```

```
stochastic   w 3.1202  b 2.0565  loss 6.8004
mini 32      w 3.4412  b 2.0094  loss 2.3413
mini 256     w 3.5696  b 1.5688  loss 2.2876
batch        w 3.6854  b 0.7612  loss 2.6329
```

The truth is w = 3.5, b = 2.0. The mini-batch runs get closest; full batch is furthest off, with the intercept still at 0.76 after 30 epochs. That is the point: batch made 30 parameter updates in total, mini-batch-32 made nearly two thousand, and stochastic made sixty thousand. Same data, same epochs, vastly different amounts of learning, which is why nobody trains on full batches. Stochastic is noisiest, visible in its slope of 3.12.

Day 3 takeaway

Day 4 Probability You Actually Need

Conditional probability, Bayes, base rates and the independence assumption

Classifiers do not output labels. They output probabilities, and something downstream turns those into a decision. Today is the probability you need to read that output correctly.

Conditional probability

```
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()

p_churn = df['churned'].mean()
p_m2m = (df['contract'] == 'Month-To-Month').mean()
p_churn_given_m2m = df.loc[df['contract'] == 'Month-To-Month', 'churned'].mean()
p_m2m_given_churn = (df.loc[df['churned'] == 1, 'contract'] ==
    ↪ 'Month-To-Month').mean()

print('P(churn)                    %.3f' % p_churn)
print('P(month-to-month)           %.3f' % p_m2m)
print('P(churn | month-to-month)   %.3f' % p_churn_given_m2m)
print('P(month-to-month | churn)   %.3f' % p_m2m_given_churn)
```

```
P(churn)                    0.268
```

```
P(month-to-month)         0.553
P(churn | month-to-month) 0.414
P(month-to-month | churn) 0.854
```

Watch Out

These last two are not the same number

P(churn given month-to-month) is 0.41. P(month-to-month given churn) is 0.85. Confusing the two is the single most common probability error in analytics, and it has a name, the prosecutor's fallacy. "85 percent of churners were month-to-month" sounds like month-to-month customers mostly churn. They mostly do not.

Bayes' theorem, which converts between them

```python
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()

p_churn = df['churned'].mean()
p_m2m = (df['contract'] == 'Month-To-Month').mean()
p_m2m_given_churn = (df.loc[df['churned'] == 1, 'contract'] ==
    'Month-To-Month').mean()

# P(A|B) = P(B|A) * P(A) / P(B)
bayes = p_m2m_given_churn * p_churn / p_m2m
actual = df.loc[df['contract'] == 'Month-To-Month', 'churned'].mean()
print('Bayes  %.4f' % bayes)
print('actual %.4f' % actual)
```

```
Bayes  0.4139
actual 0.4139
```

Why base rates dominate

A classic: a test for a rare condition is 99 percent accurate. You test positive. What is the chance you have it?

```python
prevalence = 0.001        # 1 in 1000 people have it
sensitivity = 0.99        # correctly flags 99% of those who do
specificity = 0.99        # correctly clears 99% of those who do not

true_pos = prevalence * sensitivity
false_pos = (1 - prevalence) * (1 - specificity)
posterior = true_pos / (true_pos + false_pos)

print('per 100,000 people:')
print('  have it and test positive     %6.0f' % (100_000 * true_pos))
print('  healthy but test positive     %6.0f' % (100_000 * false_pos))
print('\nP(condition | positive test)  %.3f' % posterior)
```

```
per 100,000 people:
  have it and test positive         99
  healthy but test positive        999

P(condition | positive test)  0.090
```

Nine percent, from a 99 percent accurate test

Because the condition is rare, the far larger healthy group generates ten times more false positives than the sick group generates true ones. This is exactly the situation in fraud detection, disease screening and any rare-event problem, and it is why week 7 insists on precision and recall rather than accuracy.

Distributions you will meet

```python
import numpy as np
from scipy import stats

rng = np.random.default_rng(0)
samples = {
    'normal   ': rng.normal(50, 10, 5000),
    'uniform  ': rng.uniform(0, 100, 5000),
    'poisson  ': rng.poisson(1.5, 5000),
    'lognormal': rng.lognormal(3, 0.6, 5000),
}
print('%-10s %8s %8s %8s %8s' % ('', 'mean', 'median', 'std', 'skew'))
for name, s in samples.items():
    print('%-10s %8.2f %8.2f %8.2f %8.2f'
          % (name, s.mean(), np.median(s), s.std(), stats.skew(s)))
```

```
              mean   median      std     skew
normal       49.95    49.72     9.95     0.02
uniform      50.03    50.20    28.81    -0.00
poisson       1.54     1.00     1.22     0.73
lognormal    24.00    20.19    15.68     2.24
```

Distribution	Models	Seen in this course
Normal	Measurement error, sums of many effects	Linear regression residuals
Bernoulli	A single yes/no outcome	The churn target
Poisson	Counts of rare events in a window	`support_calls`
Log-normal	Quantities that multiply	Charges, incomes, durations

Independence, and why naive Bayes is called naive

```python
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
```

```

a = (df['contract'] == 'Month-To-Month')
b = (df['payment_method'] == 'Electronic check')

print('P(A)      %.4f' % a.mean())
print('P(B)      %.4f' % b.mean())
print('P(A)*P(B) %.4f   ← what independence would predict' % (a.mean() * b.mean()))
print('P(A and B) %.4f  ← what actually happens' % (a & b).mean())

c = (df['contract'] == 'Two Year')
d = (df['tenure_months'] > 40)
print('\nP(C)*P(D) %.4f' % (c.mean() * d.mean()))
print('P(C and D) %.4f  ← strongly dependent' % (c & d).mean())
```

```
P(A)      0.5533
P(B)      0.3330
P(A)*P(B) 0.1843   ← what independence would predict
P(A and B) 0.1823  ← what actually happens

P(C)*P(D) 0.0327
P(C and D) 0.0850  ← strongly dependent
```

Contract type and payment method are close to independent, so multiplying their probabilities is nearly right. Contract type and tenure are not, and multiplying is badly wrong. Naive Bayes assumes every feature is independent given the class, usually false, and it often works anyway, which week 5 explains.

Day 4 takeaway

Day 5 Likelihood and Loss Functions

Why the loss you choose decides the estimator you get

A loss function is how a model knows it is wrong. Choosing the wrong one is not a tuning mistake. It means optimising for something you do not want.

Squared error, and what it assumes

```
import numpy as np

y = np.array([10.0, 12.0, 11.0, 13.0, 40.0])   # last one is an outlier

def mse(pred):
    return ((y - pred) ** 2).mean()

def mae(pred):
    return np.abs(y - pred).mean()

grid = np.linspace(9, 25, 1601)
print('minimises MSE at %.2f   (the mean is %.2f)'
      % (grid[np.argmin([mse(g) for g in grid])], y.mean()))
```

```
print('minimises MAE at %.2f   (the median is %.2f)'
      % (grid[np.argmin([mae(g) for g in grid])], np.median(y)))
```

```
minimises MSE at 17.20   (the mean is 17.20)
minimises MAE at 12.00   (the median is 12.00)
```

Your loss chooses your estimator

Squared error is minimised by the mean; absolute error by the median. One outlier of 40 drags the MSE answer to 17.2 while the MAE answer stays at 12. If your target has extreme values and you do not want them to dominate, that is an argument about the loss function, not about the data.

Log loss, for probabilities

Accuracy treats a confident wrong answer and a hesitant wrong answer as identical. Log loss does not, and that is why it is what classifiers actually optimise.

```
import numpy as np

def log_loss_one(y_true, p):
    p = np.clip(p, 1e-15, 1 - 1e-15)
    return -(y_true * np.log(p) + (1 - y_true) * np.log(1 - p))

print('truth=1, predicted probability → loss')
for p in [0.99, 0.9, 0.7, 0.5, 0.3, 0.1, 0.01]:
    print('   p=%.2f   %7.4f' % (p, log_loss_one(1, p)))
```

```
truth=1, predicted probability → loss
   p=0.99    0.0101
   p=0.90    0.1054
   p=0.70    0.3567
   p=0.50    0.6931
   p=0.30    1.2040
   p=0.10    2.3026
   p=0.01    4.6052
```

Watch Out

Confidently wrong is punished without limit

Predicting 0.01 for something that happens costs 4.6, against 0.69 for an honest 0.5. As the prediction approaches zero the loss goes to infinity, which is why the clip is there. A model trained on log loss learns to be uncertain when it should be, and that is what makes its probabilities usable for ranking and thresholding.

Where log loss comes from

It is not arbitrary. It is the negative log of the likelihood, the probability the model assigns to the data actually observed. Maximising that likelihood and minimising log loss are the same operation.

```
import numpy as np
```

```

y = np.array([1, 0, 1, 1, 0])
model_a = np.array([0.9, 0.2, 0.8, 0.7, 0.1])   # sensible
model_b = np.array([0.6, 0.5, 0.5, 0.6, 0.4])   # hedging

def likelihood(y, p):
    return np.prod(np.where(y == 1, p, 1 - p))

def neg_log_lik(y, p):
    return -np.sum(np.where(y == 1, np.log(p), np.log(1 - p)))

for name, p in [('A (sensible)', model_a), ('B (hedging)', model_b)]:
    print('%-14s likelihood %.6f   neg log lik %.4f'
          % (name, likelihood(y, p), neg_log_lik(y, p)))
```

```
A (sensible)   likelihood 0.362880   neg log lik 1.0137
B (hedging)    likelihood 0.054000   neg log lik 2.9188
```

Model A assigns the observed outcomes about eight times more probability than model B does, and correspondingly has the lower negative log likelihood. Products of many small probabilities underflow to zero, which is the practical reason everything is done in logs.

Matching the loss to the problem

Problem	Loss	sklearn scoring
Regression, outliers matter	Squared error	`neg_mean_squared_error`
Regression, outliers are noise	Absolute error	`neg_mean_absolute_error`
Regression, relative error matters	Squared log error	`neg_mean_squared_log_error`
Binary classification	Log loss	`neg_log_loss`
Multi-class	Cross-entropy	`neg_log_loss`
Ranking quality only	-	`roc_auc`

```
import numpy as np
from sklearn.metrics import (mean_squared_error, mean_absolute_error,
                             mean_squared_log_error)

truth = np.array([100.0, 1000.0])
over = np.array([110.0, 1010.0])     # +10 on each

print('absolute error identical: %.1f' % mean_absolute_error(truth, over))
print('squared error identical : %.1f' % mean_squared_error(truth, over))
print('squared LOG error       : %.6f' % mean_squared_log_error(truth, over))
print('\n10 on 100 is a 10%% error; 10 on 1000 is 1%%.')
print('Only the log version treats those differently.')
```

```
absolute error identical: 10.0
squared error identical : 100.0
squared LOG error       : 0.004506

10 on 100 is a 10%% error; 10 on 1000 is 1%%.
Only the log version treats those differently.
```

Day 5 takeaway

Day 6 Uncertainty, Bias and Variance

Confidence intervals on model scores, and the decomposition measured directly

Two models score 0.81 and 0.83 on your test set. Is the second better, or did it get luckier? Answering that is what this day is for, and it is the difference between a practitioner and someone who runs `fit`.

Any score computed on a sample is itself uncertain

```
import numpy as np

rng = np.random.default_rng(0)
true_rate = 0.268

for n in [50, 200, 750, 3000]:
    estimates = [rng.binomial(n, true_rate) / n for _ in range(4000)]
    lo, hi = np.percentile(estimates, [2.5, 97.5])
    print('n=%5d   estimate %.3f +/- %.3f   95%% range %.3f to %.3f'
          % (n, np.mean(estimates), np.std(estimates), lo, hi))
```

```
n=   50   estimate 0.267 +/- 0.063   95% range 0.140 to 0.400
n=  200   estimate 0.268 +/- 0.032   95% range 0.210 to 0.335
n=  750   estimate 0.268 +/- 0.016   95% range 0.236 to 0.300
n= 3000   estimate 0.268 +/- 0.008   95% range 0.251 to 0.284
```

With a 750-row test set, the size week 1 used, a churn rate measured at 0.268 could plausibly be anywhere from 0.237 to 0.300. Model scores carry the same kind of uncertainty, and two models within a percentage point of each other on 750 rows are not distinguishable.

Bootstrap a confidence interval for a model score

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score

df = pd.read_csv('customers.csv').drop_duplicates().copy()
X = df[['tenure_months', 'monthly_charges', 'support_calls']]
y = df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)

model = make_pipeline(SimpleImputer(strategy='median'), StandardScaler(),
```

```
                       LogisticRegression(max_iter=1000)).fit(X_tr, y_tr)
proba = model.predict_proba(X_te)[:, 1]
point = roc_auc_score(y_te, proba)

rng = np.random.default_rng(0)
y_arr, p_arr = y_te.to_numpy(), proba
boot = []
for _ in range(2000):
    idx = rng.integers(0, len(y_arr), len(y_arr))
    if y_arr[idx].sum() in (0, len(idx)):
        continue
    boot.append(roc_auc_score(y_arr[idx], p_arr[idx]))

lo, hi = np.percentile(boot, [2.5, 97.5])
print('AUC %.4f   95%% CI %.4f to %.4f   width %.4f'
      % (point, lo, hi, hi - lo))
```

```
AUC 0.7764   95% CI 0.7392 to 0.8101   width 0.0708
```

Report the interval, not just the number

The interval here is about seven points wide. Any competing model whose score falls inside it has not been shown to be better. Quoting a single figure to three decimal places implies a precision the test set cannot support.

The bias-variance decomposition, measured

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

def truth(x):
    return np.sin(1.5 * x) + 0.3 * x

rng = np.random.default_rng(0)
x_test = np.linspace(0, 6, 120).reshape(-1, 1)
f_test = truth(x_test.ravel())

def decompose(make_model, n_sets=200, n=40):
    preds = np.zeros((n_sets, len(x_test)))
    for i in range(n_sets):
        x = rng.uniform(0, 6, n).reshape(-1, 1)
        y = truth(x.ravel()) + rng.normal(0, 0.3, n)
        preds[i] = make_model().fit(x, y).predict(x_test)
    mean_pred = preds.mean(axis=0)
    bias2 = ((mean_pred - f_test) ** 2).mean()
    variance = preds.var(axis=0).mean()
    return bias2, variance

candidates = {
```

```
    'line (deg 1)  ': lambda: make_pipeline(PolynomialFeatures(1), LinearRegression()),
    'poly deg 5    ': lambda: make_pipeline(PolynomialFeatures(5), LinearRegression()),
    'poly deg 15   ': lambda: make_pipeline(PolynomialFeatures(15), LinearRegression()),
    'tree depth 2  ': lambda: DecisionTreeRegressor(max_depth=2),
    'tree unlimited': lambda: DecisionTreeRegressor(),
}
print('%-16s %9s %9s %9s' % ('model', 'bias^2', 'variance', 'total'))
for name, maker in candidates.items():
    b, v = decompose(maker)
    print('%-16s %9.4f %9.4f %9.4f' % (name, b, v, b + v))
```

```
model              bias^2  variance     total
line (deg 1)       0.4725    0.0318    0.5043
poly deg 5         0.0167    0.0419    0.0586
poly deg 15        7.4814 2335.0026 2342.4839
tree depth 2       0.0842    0.0742    0.1584
tree unlimited     0.0016    0.1016    0.1032
```

The straight line has high bias and almost no variance: it is wrong in the same way every time. The degree-15 polynomial is the opposite failure. Its variance is over two thousand, because with only 40 noisy points each fit swings somewhere different. Note that its bias is bad too, at 7.48: when the individual fits are that unstable, even their average is nowhere near the truth. The best total error is the degree-5 polynomial, in between, and finding that point is what every regularisation technique in this course is for.

Watch Out

You cannot measure this on real data

The decomposition above needs the true function, which is why it uses a simulation. On real data you never know the truth, so you detect the same thing indirectly: a large gap between training and validation score means variance; both scores being poor means bias. Week 7 turns that into a diagnostic you can actually run.

Day 6 takeaway

Day 7 Linear Regression From Scratch, Three Ways

Normal equation, gradient descent and scikit-learn, agreeing

Everything this week, on one problem. You will solve the same regression three ways, exactly, iteratively, and with the library, and get the same answer three times.

The problem

```
import numpy as np
import pandas as pd
```

```

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
d = df[['tenure_months', 'support_calls', 'monthly_charges']].dropna()

X = d[['tenure_months', 'support_calls']].to_numpy(dtype=float)
y = d['monthly_charges'].to_numpy(dtype=float)
print('X', X.shape, ' y', y.shape)
print('predicting monthly_charges from tenure and support calls')
```

```
X (2835, 2)  y (2835,)
predicting monthly_charges from tenure and support calls
```

Way 1: the exact solution

Squared error has a closed form. Setting the gradient to zero and solving gives the normal equation, and NumPy will do it in one call.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
d = df[['tenure_months', 'support_calls', 'monthly_charges']].dropna()
X = d[['tenure_months', 'support_calls']].to_numpy(dtype=float)
y = d['monthly_charges'].to_numpy(dtype=float)

Xb = np.column_stack([np.ones(len(X)), X])      # a column of 1s for the intercept

# w = (X'X)^-1 X'y, computed stably rather than by inverting
w_exact = np.linalg.lstsq(Xb, y, rcond=None)[0]
print('intercept      %8.4f' % w_exact[0])
print('tenure_months  %8.4f' % w_exact[1])
print('support_calls  %8.4f' % w_exact[2])
```

```
intercept       57.3815
tenure_months   -0.0037
support_calls    1.6707
```

Watch Out

Do not invert the matrix yourself

`np.linalg.inv(X.T @ X) @ X.T @ y` is the textbook formula and it is numerically fragile: when two features are highly correlated the matrix is near-singular and the inverse amplifies floating-point error enormously. `lstsq` solves the system directly and is stable. This matters on real data, where correlated features are the norm, recall tenure and total charges at 0.83.

Way 2: gradient descent

```
import numpy as np
import pandas as pd
```

```

df = pd.read_csv('customers.csv').drop_duplicates().copy()
d = df[['tenure_months', 'support_calls', 'monthly_charges']].dropna()
X = d[['tenure_months', 'support_calls']].to_numpy(dtype=float)
y = d['monthly_charges'].to_numpy(dtype=float)

mu, sigma = X.mean(axis=0), X.std(axis=0)
Xs = (X - mu) / sigma
Xb = np.column_stack([np.ones(len(Xs)), Xs])

w = np.zeros(Xb.shape[1])
lr = 0.1
for step in range(1, 3001):
    error = Xb @ w - y
    w -= lr * 2 * (Xb.T @ error) / len(y)
    if step in (1, 100, 1000, 3000):
        print('step %5d  loss %10.4f' % (step, (error ** 2).mean()))

# Undo the scaling so the coefficients are comparable with way 1.
coef = w[1:] / sigma
intercept = w[0] - (w[1:] * mu / sigma).sum()
print('\nintercept      %8.4f' % intercept)
print('tenure_months  %8.4f' % coef[0])
print('support_calls  %8.4f' % coef[1])
```

```
step     1  loss  4080.0173
step   100  loss   537.9793
step  1000  loss   537.9793
step  3000  loss   537.9793

intercept       57.3815
tenure_months   -0.0037
support_calls    1.6707
```

Way 3: the library

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

df = pd.read_csv('customers.csv').drop_duplicates().copy()
d = df[['tenure_months', 'support_calls', 'monthly_charges']].dropna()
X = d[['tenure_months', 'support_calls']].to_numpy(dtype=float)
y = d['monthly_charges'].to_numpy(dtype=float)

m = LinearRegression().fit(X, y)
print('intercept      %8.4f' % m.intercept_)
print('tenure_months  %8.4f' % m.coef_[0])
print('support_calls  %8.4f' % m.coef_[1])
print('\nR^2 %.4f' % m.score(X, y))
```

```
intercept       57.3815
tenure_months   -0.0037
```

```
support_calls    1.6707

R^2 0.0233
```

Three routes, one answer. The library is not doing anything you have not now done by hand. It is doing it more carefully, and handling the cases where the naive version breaks.

Reading the coefficients

```python
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

df = pd.read_csv('customers.csv').drop_duplicates().copy()
d = df[['tenure_months', 'support_calls', 'monthly_charges']].dropna()
X = d[['tenure_months', 'support_calls']].to_numpy(dtype=float)
y = d['monthly_charges'].to_numpy(dtype=float)
m = LinearRegression().fit(X, y)

print('a customer with 12 months tenure and 2 support calls:')
print('  predicted charge %.2f' % m.predict([[12, 2]])[0])
print('one more support call, everything else held:')
print('  predicted charge %.2f' % m.predict([[12, 3]])[0])
print('  difference       %.4f  ← exactly the coefficient' % m.coef_[1])
```

```
a customer with 12 months tenure and 2 support calls:
  predicted charge 60.68
one more support call, everything else held:
  predicted charge 62.35
  difference       1.6707  ← exactly the coefficient
```

"Holding everything else constant" is doing a lot of work

A coefficient is the change in the prediction for a one-unit change in that feature *with the others fixed*. When features are correlated you cannot actually hold the others fixed, increasing tenure increases total charges in the real world, so the coefficient describes the model, not the world. This is the single biggest source of over-claiming in applied work.

Your assignment

Add `total_charges` as a third feature and refit. It correlates with tenure at 0.83. Record what happens to the tenure coefficient, sign, size, or both. Then refit on a random 60 percent of the rows, twice, and compare. Week 4 names what you are seeing and fixes it.

> **Day 7 takeaway**

Self-Assessment

Try It Yourself

1. Most supervised models compute, at their core:

    a) A random draw
    b) A database join
    c) A sort
    d) A dot product between a feature vector and a weight vector

2. The derivative of the sigmoid reaches its maximum value of:

    a) 0.5
    b) 0.25
    c) 0.0
    d) 1.0

3. Features on very different scales slow gradient descent because:

    a) The gradients overflow
    b) The loss surface becomes an elongated valley, so a step size suitable for one direction is wrong for another
    c) NumPy is slower on large numbers
    d) The learning rate becomes negative

4. A test is 99% accurate and the disease affects 1 in 1,000. A positive result means the patient probably:

    a) Has a 99% chance of having it
    b) Has the disease
    c) Does not have the disease, because the base rate swamps the test's accuracy
    d) Needs a second opinion only if symptomatic

5. Choosing squared error as your loss is equivalent to assuming:

    a) There are no outliers
    b) The target is positive
    c) Errors are uniformly distributed
    d) Errors are normally distributed

6. Why does `softmax` subtract the maximum before exponentiating?

    a) To speed it up
    b) To prevent overflow. It changes nothing mathematically
    c) To centre the logits
    d) To make the output sum to one

7. High bias and low variance describes a model that:

    a) Memorises the training data

b) Is too simple to capture the pattern, and makes the same mistake regardless of the sample

c) Has too many features

d) Is well fitted

8. The normal equation, gradient descent and `LinearRegression` should:

a) Agree only if scaled

b) Give different answers

c) Agree to within numerical precision on the same data

d) Only agree for one feature

Answers:1d, 2b, 3b, 4c, 5d, 6b, 7b, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 4

Regression, Regularisation and Leakage

Week 4

OBJECTIVES

Predict a number, measure it honestly, and meet the two failures that make a good looking model worthless.

This week you predict a continuous number and learn to distrust the score. It covers the four regression metrics and what each one hides, how flexibility turns into overfitting and how regularisation buys it back, and finishes on the two failures every practitioner meets: correlated features that make coefficients meaningless, and leaked features that make a useless model look excellent.

Day 1 Linear Regression on a Real Target

Fitting, reading the score against the right baseline, and what residuals reveal

Week 3 fitted a line by hand. This week fits real regressions, and the first job is choosing a target worth predicting.

The problem

Predict `monthly_charges` from everything else. It is a genuine business question, what should this customer be paying, given who they are, and unlike week 3's toy version it has real signal in it.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
print('rows %d' % len(df))
print('target: mean %.2f  std %.2f  min %.2f  max %.2f'
      % (df[TARGET].mean(), df[TARGET].std(), df[TARGET].min(), df[TARGET].max()))
print('\nby internet service:')
print(df.groupby('internet_service')[TARGET].agg(['size', 'mean', 'std']).round(2))
```

```
rows 2835
target: mean 59.41  std 23.47  min 15.00  max 105.96

by internet service:
                  size   mean   std
internet_service
DSL                965  55.19  8.99
Fibre optic       1282  79.56  9.18
No internet        588  22.38  7.34
```

Three tight clusters. Internet service almost determines the price, which is a promising sign: there is real structure for a model to find.

Fit it

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, root_mean_squared_error, r2_score

model = Pipeline([('prep', preprocessor()),
                  ('reg', LinearRegression())]).fit(X_tr, y_tr)
pred = model.predict(X_te)

print('MAE  %6.3f' % mean_absolute_error(y_te, pred))
print('RMSE %6.3f' % root_mean_squared_error(y_te, pred))
print('R2   %6.4f' % r2_score(y_te, pred))
print('\nstd of the target was %.2f, so predicting the mean'
```

```
      ' would give RMSE about that.' % y_te.std())
```

```
MAE   7.200
RMSE  9.033
R2    0.8498

std of the target was 23.32, so predicting the mean would give RMSE about that.
```

Compare RMSE against the target's standard deviation

Predicting the mean for everybody gives an RMSE equal to the standard deviation, about 23.3 here. Our model gets just over 9. That ratio is the honest way to read a regression score, and it is what R^2 formalises: 0.85 means the model removed 85 percent of the variance the mean-only model left behind.

Residuals are where the truth is

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.linear_model import LinearRegression

model = Pipeline([('prep', preprocessor()),
                  ('reg', LinearRegression())]).fit(X_tr, y_tr)
resid = y_te - model.predict(X_te)

print('mean   %8.4f   ← should be near zero' % resid.mean())
```

```
print('median %8.4f' % resid.median())
print('std    %8.4f' % resid.std())
print('skew   %8.4f   ← should be near zero' % resid.skew())
print('\nlargest under-predictions:')
print(resid.nlargest(3).round(2).to_string())
print('largest over-predictions:')
print(resid.nsmallest(3).round(2).to_string())
```

```
mean    -0.4344   ← should be near zero
median  -0.6659
std      9.0292
skew     0.0188   ← should be near zero

largest under-predictions:
132     26.92
1074    25.72
2735    25.56
largest over-predictions:
3076   -30.66
767    -27.13
2637   -26.33
```

The four assumptions, and which ones matter

Assumption	What breaks if violated	How much you should care
Linearity	Systematic curved pattern in residuals	A lot. The model is simply wrong
Independent errors	Standard errors understated	A lot for time series, rarely otherwise
Constant variance	Predictions unreliable at the extremes	Moderate, affects intervals more than point estimates
Normal errors	Confidence intervals are off	Least of the four; irrelevant for pure prediction

Watch Out

These are assumptions about residuals, not about your columns

A common misreading: people check whether their *features* are normally distributed. Linear regression makes no such requirement. The normality assumption is about the errors, and it only matters when you want confidence intervals on the coefficients. For prediction you can ignore it.

Check linearity by plotting residuals against predictions

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
```

```
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

model = Pipeline([('prep', preprocessor()),
                  ('reg', LinearRegression())]).fit(X_tr, y_tr)
pred = model.predict(X_te)
resid = y_te - pred

fig, ax = plt.subplots(figsize=(7, 4))
ax.scatter(pred, resid, s=8, alpha=0.4, color='#2563eb')
ax.axhline(0, color='#dc2626', linewidth=1)
ax.set_xlabel('Predicted')
ax.set_ylabel('Residual')
fig.tight_layout()
fig.savefig('residuals.png', dpi=120)
plt.close(fig)

# The same check, numerically: is the mean residual flat across the range?
bins = pd.qcut(pred, 5, labels=False)
print(pd.DataFrame({'pred_bin': bins, 'resid': resid.to_numpy()})
      .groupby('pred_bin')['resid'].agg(['size', 'mean', 'std']).round(3))
```

```
          size   mean    std
pred_bin
0          142  0.577  7.503
1          142  0.261  9.132
2          141 -0.570  9.695
3          142 -0.045  9.648
4          142 -2.397  8.816
```

The bin means sit between +0.58 and -2.40 against a residual standard deviation of 9, so there is no strong trend, a slight tendency to over-predict the most expensive customers, and little else. That is roughly what a correctly specified linear model looks like. A clear U-shape here would mean the relationship is curved and you need the polynomial features from day 3.

Day 1 takeaway

Day 2 Regression Metrics and Their Traps

MAE, RMSE, R-squared and MAPE, what each hides

Four metrics, each answering a different question, and each with a way of misleading you.

What each one actually measures

Metric	Units	Reads as	Weakness
MAE	Same as target	Typical error size	Treats a huge miss as merely several small ones
RMSE	Same as target	Error, weighted toward big misses	One bad outlier dominates it
R^2	None	Share of variance explained	Depends on the spread of your test set
MAPE	Percent	Relative error	Explodes when the target is near zero

```
import numpy as np
from sklearn.metrics import (mean_absolute_error, root_mean_squared_error,
                             r2_score)

truth = np.array([20.0, 40.0, 60.0, 80.0, 100.0])

spread = np.array([25.0, 35.0, 65.0, 75.0, 105.0])   # five misses of 5
single = np.array([20.0, 40.0, 60.0, 80.0, 75.0])    # one miss of 25

for name, p in [('five small misses', spread), ('one big miss', single)]:
    print('%-18s MAE %5.2f   RMSE %5.2f   R2 %6.3f'
          % (name, mean_absolute_error(truth, p),
             root_mean_squared_error(truth, p), r2_score(truth, p)))
```

```
five small misses  MAE  5.00   RMSE  5.00   R2  0.969
one big miss       MAE  5.00   RMSE 11.18   R2  0.844
```

Identical total absolute error, very different RMSE. If one large failure is much worse for your business than several small ones, a delivery estimate, a capacity forecast, use RMSE. If every unit of error costs the same, use MAE.

Watch Out

R^2 is not comparable across datasets

R^2 measures improvement over predicting the mean, so it depends on how variable the test set happens to be. The same model scores higher on a diverse test set than on a homogeneous one. Comparing your R^2 to a number from a paper on different data tells you nothing.

```
import numpy as np
from sklearn.metrics import r2_score, root_mean_squared_error

rng = np.random.default_rng(0)

for spread in [5, 20, 60]:
    truth = rng.normal(50, spread, 500)
    pred = truth + rng.normal(0, 4, 500)      # identical error every time
    print('target std %2d   RMSE %.2f   R2 %.4f'
          % (spread, root_mean_squared_error(truth, pred), r2_score(truth, pred)))
```

```
target std  5   RMSE 3.76   R2 0.4489
target std 20   RMSE 4.14   R2 0.9581
target std 60   RMSE 3.92   R2 0.9956
```

The model makes exactly the same size of error in all three cases. R^2 swings from 0.45 to 0.996. RMSE, which is in the units of the thing you care about, does not move.

MAPE and the zero problem

```
import numpy as np

def mape(truth, pred):
    return np.mean(np.abs((truth - pred) / truth)) * 100

truth = np.array([100.0, 50.0, 10.0, 0.5])
pred = np.array([110.0, 55.0, 11.0, 0.55])  # every one is 10% high
print('all errors are 10%%: MAPE %.1f%%' % mape(truth, pred))

truth2 = np.array([100.0, 50.0, 10.0, 0.01])
pred2 = np.array([110.0, 55.0, 11.0, 0.5])
print('one near-zero actual: MAPE %.1f%%' % mape(truth2, pred2))
```

```
all errors are 10%: MAPE 10.0%
one near-zero actual: MAPE 1232.5%
```

An error of 0.49 on a true value of 0.01 registers as 4,900 percent and swamps everything else. Never use MAPE on a target that can approach zero, and never on one that can be negative.

Cross-validation, not one split

```
import numpy as np
```

```python
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score, KFold
import numpy as np

X, y = df[NUM + CAT], df[TARGET]
model = Pipeline([('prep', preprocessor()), ('reg', LinearRegression())])

cv = KFold(n_splits=5, shuffle=True, random_state=42)
for metric in ['r2', 'neg_mean_absolute_error', 'neg_root_mean_squared_error']:
    scores = cross_val_score(model, X, y, cv=cv, scoring=metric)
    if metric.startswith('neg_'):
        scores = -scores
    print('%-32s %.4f  +/- %.4f' % (metric, scores.mean(), scores.std()))
```

```
r2                               0.8593  +/- 0.0110
neg_mean_absolute_error          7.0785  +/- 0.2229
neg_root_mean_squared_error      8.7868  +/- 0.2839
```

sklearn negates losses so higher is always better

Every scorer in scikit-learn follows the rule "greater is better", so errors come back negative. `neg_mean_absolute_error` of -3.6 means an MAE of 3.6. Forgetting to flip the sign is a common way to conclude your model got worse when it improved.

> **Day 2 takeaway**

Day 3 Polynomials, Overfitting and Learning Curves

Adding flexibility, watching it backfire, and diagnosing which fix you need

A straight line cannot fit a curve. The obvious fix, give the model more flexibility, is also the fastest way to make it worse, and watching that happen is the most useful hour of this week.

Add curvature

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

x = np.array([[2.0, 3.0]])
for degree in [1, 2, 3]:
    pf = PolynomialFeatures(degree=degree, include_bias=False)
    out = pf.fit_transform(x)
    print('degree %d → %d features: %s'
          % (degree, out.shape[1], list(pf.get_feature_names_out())))
```

```
degree 1 → 2 features: ['x0', 'x1']
degree 2 → 5 features: ['x0', 'x1', 'x0^2', 'x0 x1', 'x1^2']
degree 3 → 9 features: ['x0', 'x1', 'x0^2', 'x0 x1', 'x1^2', 'x0^3', 'x0^2 x1', 'x0 x1^2',
    ↪ 'x1^3']
```

Degree 2 adds the squares and the interaction `x0 x1`. That interaction term is often the valuable one: it lets the effect of one feature depend on another, which a plain linear model cannot express.

Watch Out

The feature count explodes

Ten features at degree 2 gives 65 columns; at degree 3, 285. At degree 5 you have 3,003 columns from ten. The model gains enough freedom to fit the noise exactly, and the number of rows you would need to constrain it grows just as fast.

Watch it overfit

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics import root_mean_squared_error

rng = np.random.default_rng(0)
n = 40
x = rng.uniform(0, 6, n).reshape(-1, 1)
y = np.sin(1.5 * x.ravel()) + 0.3 * x.ravel() + rng.normal(0, 0.3, n)

x_val = rng.uniform(0, 6, 400).reshape(-1, 1)
y_val = np.sin(1.5 * x_val.ravel()) + 0.3 * x_val.ravel() + rng.normal(0, 0.3, 400)

print('%6s %10s %10s' % ('degree', 'train', 'validation'))
for degree in [1, 2, 3, 5, 9, 15, 25]:
    m = make_pipeline(PolynomialFeatures(degree), LinearRegression()).fit(x, y)
```

```
    tr = root_mean_squared_error(y, m.predict(x))
    va = root_mean_squared_error(y_val, m.predict(x_val))
    print('%6d %10.4f %10.4f' % (degree, tr, va))
```

```
degree      train validation
     1     0.7065     0.7743
     2     0.5418     0.6249
     3     0.5140     0.6155
     5     0.3146     0.3479
     9     0.3019     0.3262
    15     0.2953     0.3388
    25     0.4085     0.4709
```

Training error falls as flexibility rises, in principle it must, since a richer model can always fit the given points better. Validation error bottoms out at degree 9 and then climbs. The best degree is where validation is lowest, and nothing in the training column would have told you where that was.

Look at degree 25, though: training error goes *up*, from 0.2953 to 0.4085. That is not overfitting, it is arithmetic failing. Raising a value near 6 to the 25th power produces columns spanning twenty orders of magnitude: the matrix becomes so ill-conditioned that the solver cannot work accurately, and the fit degrades. A training error that rises with complexity is a numerical warning, not a statistical one.

The learning curve: more data, or a better model?

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import learning_curve
import numpy as np

X, y = df[NUM + CAT], df[TARGET]
```

```
model = Pipeline([('prep', preprocessor()), ('reg', LinearRegression())])

sizes, train_scores, val_scores = learning_curve(
    model, X, y, cv=5, scoring='neg_root_mean_squared_error',
    train_sizes=np.linspace(0.1, 1.0, 6), random_state=42)

print('%8s %10s %10s %8s' % ('rows', 'train', 'val', 'gap'))
for n, tr, va in zip(sizes, -train_scores.mean(axis=1), -val_scores.mean(axis=1)):
    print('%8d %10.4f %10.4f %8.4f' % (n, tr, va, va - tr))
```

```
    rows      train        val      gap
     226     8.3919     9.3860   0.9941
     635     8.6457     8.8709   0.2252
    1043     8.7090     8.8098   0.1008
    1451     8.7100     8.8003   0.0903
    1859     8.7191     8.7845   0.0654
    2268     8.7448     8.7775   0.0327
```

How to read a learning curve

The two curves converging to a *low* error means you are done. Converging to a *high* error means high bias: more data will not help, you need a better model. A persistent gap means high variance: more data *will* help, or regularise. Here the gap closes to almost nothing at a low error, so this model is well matched to this problem.

Finding the right complexity

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score

rng = np.random.default_rng(0)
x = rng.uniform(0, 6, 120).reshape(-1, 1)
y = np.sin(1.5 * x.ravel()) + 0.3 * x.ravel() + rng.normal(0, 0.3, 120)

best = None
for degree in range(1, 16):
    m = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    rmse = -cross_val_score(m, x, y, cv=5,
                            scoring='neg_root_mean_squared_error').mean()
    flag = ''
    if best is None or rmse < best[1]:
        best, flag = (degree, rmse), '  ← best so far'
    if degree <= 8 or flag:
        print('degree %2d  cv RMSE %8.4f%s' % (degree, rmse, flag))

print('\nchosen degree: %d' % best[0])
```

```
degree  1  cv RMSE   0.7715  ← best so far
degree  2  cv RMSE   0.6255  ← best so far
```

```
degree  3  cv RMSE   0.6250  ← best so far
degree  4  cv RMSE   0.3330  ← best so far
degree  5  cv RMSE   0.3355
degree  6  cv RMSE   0.3052  ← best so far
degree  7  cv RMSE   0.3086
degree  8  cv RMSE   0.3116

chosen degree: 6
```

Cross-validation picks the complexity for you, and it does so without ever looking at the test set. That is the pattern for every hyperparameter in this course.

Day 3 takeaway

Day 4 Ridge Regression

Keeping a flexible model honest with an L2 penalty

Yesterday's fix for overfitting was to use a simpler model. Regularisation is the better fix: keep the flexibility, but penalise the model for using it.

The idea

```python
import numpy as np
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.pipeline import make_pipeline
from sklearn.metrics import root_mean_squared_error

rng = np.random.default_rng(0)
x = rng.uniform(0, 6, 40).reshape(-1, 1)
y = np.sin(1.5 * x.ravel()) + 0.3 * x.ravel() + rng.normal(0, 0.3, 40)
x_val = rng.uniform(0, 6, 400).reshape(-1, 1)
y_val = np.sin(1.5 * x_val.ravel()) + 0.3 * x_val.ravel() + rng.normal(0, 0.3, 400)

plain = make_pipeline(PolynomialFeatures(15), StandardScaler(),
                      LinearRegression()).fit(x, y)
ridge = make_pipeline(PolynomialFeatures(15), StandardScaler(),
                      Ridge(alpha=1.0)).fit(x, y)

for name, m in [('plain deg 15', plain), ('ridge deg 15', ridge)]:
    coefs = m[-1].coef_
    print('%-14s val RMSE %7.4f   largest |coef| %12.2f'
          % (name, root_mean_squared_error(y_val, m.predict(x_val)),
             np.abs(coefs).max()))
```

```
plain deg 15   val RMSE  0.3480   largest |coef| 1094963376.11
ridge deg 15   val RMSE  0.5461   largest |coef|         0.41
```

Same fifteen-degree polynomial, same forty rows. The unpenalised version produces enormous coefficients that cancel each other out to thread through every training point. Ridge keeps them

small, and generalises far better.

Watch Out

Ridge requires scaled features

The penalty is on the coefficients, and a coefficient's size depends on its feature's units. A feature measured in pounds gets a small coefficient and is barely penalised; the same feature in thousands of pounds gets a large one and is crushed. Without scaling, the penalty lands arbitrarily. Always put a scaler in front, and note that the intercept is never penalised, which is why it is fitted separately.

What alpha does

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import Ridge
from sklearn.pipeline import make_pipeline
from sklearn.metrics import root_mean_squared_error

rng = np.random.default_rng(0)
x = rng.uniform(0, 6, 40).reshape(-1, 1)
y = np.sin(1.5 * x.ravel()) + 0.3 * x.ravel() + rng.normal(0, 0.3, 40)
x_val = rng.uniform(0, 6, 400).reshape(-1, 1)
y_val = np.sin(1.5 * x_val.ravel()) + 0.3 * x_val.ravel() + rng.normal(0, 0.3, 400)

print('%10s %10s %10s %14s' % ('alpha', 'train', 'val', 'largest |coef|'))
for alpha in [0.0001, 0.01, 1, 100, 10000]:
    m = make_pipeline(PolynomialFeatures(15), StandardScaler(),
                      Ridge(alpha=alpha)).fit(x, y)
    print('%10s %10.4f %10.4f %14.2f'
          % (alpha, root_mean_squared_error(y, m.predict(x)),
             root_mean_squared_error(y_val, m.predict(x_val)),
             np.abs(m[-1].coef_).max()))
```

```
     alpha      train        val largest |coef|
    0.0001     0.3061     0.3248          18.87
      0.01     0.3421     0.3808           6.58
         1     0.4653     0.5461           0.41
       100     0.5946     0.6756           0.07
     10000     0.9418     0.9272           0.00
```

At tiny alpha you have plain least squares and it overfits. At enormous alpha every coefficient is crushed toward zero and the model underfits, predicting close to a constant. The useful values are in between, and you find them by cross-validation.

RidgeCV finds alpha for you

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
```

```
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import cross_val_score
import numpy as np

X, y = df[NUM + CAT], df[TARGET]
alphas = np.logspace(-3, 3, 25)

model = Pipeline([('prep', preprocessor()),
                  ('reg', RidgeCV(alphas=alphas))]).fit(X, y)
print('chosen alpha: %.4f' % model.named_steps['reg'].alpha_)

rmse = -cross_val_score(model, X, y, cv=5,
                        scoring='neg_root_mean_squared_error').mean()
print('cv RMSE %.4f' % rmse)
```

```
chosen alpha: 1.7783
cv RMSE 8.7776
```

Search alpha on a log scale

The useful range spans orders of magnitude, so `np.logspace(-3, 3, 25)` is the right shape of grid and `np.linspace` is not. If the chosen value lands at either end of your grid, the grid was too narrow. Widen it and search again.

> **Day 4 takeaway**

Day 5 Lasso and ElasticNet

L1 as feature selection, its instability, and the mix that fixes it

Ridge shrinks every coefficient toward zero but never quite to it. Lasso takes them all the way,

which turns regularisation into feature selection.

L1 against L2

	Ridge (L2)	Lasso (L1)
Penalty	Sum of squared coefficients	Sum of absolute coefficients
Effect	Shrinks all, eliminates none	Sets some exactly to zero
Correlated features	Splits the weight between them	Picks one arbitrarily, drops the rest
Use when	All features plausibly matter	You expect most to be irrelevant

```
import numpy as np
from sklearn.linear_model import Ridge, Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

rng = np.random.default_rng(0)
n, p = 200, 20
X = rng.normal(size=(n, p))
# Only the first three columns matter; the other seventeen are noise.
true_coef = np.zeros(p)
true_coef[:3] = [4.0, -3.0, 2.0]
y = X @ true_coef + rng.normal(0, 1.0, n)

for name, model in [('ridge', Ridge(alpha=1.0)), ('lasso', Lasso(alpha=0.1))]:
    m = make_pipeline(StandardScaler(), model).fit(X, y)
    coef = m[-1].coef_
    print('%-6s first three %s   exactly zero: %d of %d'
          % (name, np.round(coef[:3], 2), (coef == 0).sum(), p))
```

```
ridge  first three [ 3.84 -2.94  1.92]   exactly zero: 0 of 20
lasso  first three [ 3.72 -2.87  1.77]   exactly zero: 15 of 20
```

Ridge leaves all twenty coefficients non-zero, including seventeen that should not be there. Lasso zeroes most of the noise columns outright, giving you a model you can read.

The path from full model to empty one

```
import numpy as np
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

rng = np.random.default_rng(0)
X = rng.normal(size=(200, 20))
true_coef = np.zeros(20)
true_coef[:3] = [4.0, -3.0, 2.0]
y = X @ true_coef + rng.normal(0, 1.0, 200)

print('%8s %10s %s' % ('alpha', 'kept', 'which'))
for alpha in [0.001, 0.05, 0.2, 0.5, 1.0, 3.0]:
    m = make_pipeline(StandardScaler(), Lasso(alpha=alpha, max_iter=5000)).fit(X, y)
```

```
    kept = np.flatnonzero(m[-1].coef_)
    print('%8s %10d %s' % (alpha, len(kept), list(kept[:6])))
```

```
   alpha       kept which
   0.001         20 [np.int64(0), np.int64(1), np.int64(2), np.int64(3), np.int64(4),
    ↪ np.int64(5)]
    0.05          9 [np.int64(0), np.int64(1), np.int64(2), np.int64(5), np.int64(7),
    ↪ np.int64(9)]
     0.2          3 [np.int64(0), np.int64(1), np.int64(2)]
     0.5          3 [np.int64(0), np.int64(1), np.int64(2)]
     1.0          3 [np.int64(0), np.int64(1), np.int64(2)]
     3.0          1 [np.int64(0)]
```

As alpha rises the model discards features, and the last three standing are exactly the three that generated the data. That is the best possible outcome and it will not always happen, but it shows what lasso is for.

Watch Out

Lasso is unstable when features are correlated

Given two nearly identical columns, lasso keeps one and zeroes the other, and which one it keeps can flip on a slightly different sample. Do not read "lasso dropped this feature" as "this feature does not matter". It may simply be standing next to a twin.

```
import numpy as np
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

rng = np.random.default_rng(1)
n = 150
a = rng.normal(size=n)
twin = a + rng.normal(0, 0.05, n)          # almost the same column
noise = rng.normal(size=(n, 5))
y = 3 * a + rng.normal(0, 0.5, n)

for seed in range(4):
    idx = np.random.default_rng(seed).choice(n, 120, replace=False)
    X = np.column_stack([a, twin, noise])[idx]
    m = make_pipeline(StandardScaler(), Lasso(alpha=0.15)).fit(X, y[idx])
    print('sample %d: coef on a %6.3f   coef on its twin %6.3f'
          % (seed, m[-1].coef_[0], m[-1].coef_[1]))
```

```
sample 0: coef on a  1.966   coef on its twin  0.229
sample 1: coef on a  2.102   coef on its twin  0.468
sample 2: coef on a  2.379   coef on its twin  0.000
sample 3: coef on a  2.278   coef on its twin  0.064
```

Same underlying truth, four resamples, and the weight jumps between the two twins. Ridge would have split it evenly and stably between them.

ElasticNet: both penalties

```
import numpy as np
from sklearn.linear_model import ElasticNetCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

rng = np.random.default_rng(1)
n = 150
a = rng.normal(size=n)
twin = a + rng.normal(0, 0.05, n)
noise = rng.normal(size=(n, 5))
X = np.column_stack([a, twin, noise])
y = 3 * a + rng.normal(0, 0.5, n)

m = make_pipeline(StandardScaler(),
                  ElasticNetCV(l1_ratio=[0.1, 0.5, 0.9, 1.0], cv=5,
                               random_state=0, max_iter=5000)).fit(X, y)
net = m[-1]
print('chosen l1_ratio %.2f   alpha %.4f' % (net.l1_ratio_, net.alpha_))
print('coef on a %.3f, on its twin %.3f' % (net.coef_[0], net.coef_[1]))
print('noise coefficients:', np.round(net.coef_[2:], 3))
```

```
chosen l1_ratio 0.90   alpha 0.0336
coef on a 1.589, on its twin 1.007
noise coefficients: [0.    0.    0.047 0.015 0.003]
```

`l1_ratio` mixes the two: 1.0 is pure lasso, 0.0 pure ridge. ElasticNet keeps lasso's ability to discard irrelevant columns while sharing weight between correlated ones rather than choosing arbitrarily.

Day 5 takeaway

Day 6 Multicollinearity and Target Leakage

The failure that corrupts coefficients, and the one that fakes success

Two failures that both look like success. One inflates your test score to something impossible; the other makes your coefficients meaningless while the score stays fine.

Multicollinearity

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

```

```
NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
d = df.dropna(subset=['total_charges'])
print(d[['tenure_months', 'monthly_charges', 'total_charges']].corr().round(3))
```

```
                tenure_months  monthly_charges  total_charges
tenure_months           1.000            0.001          0.828
monthly_charges         0.001            1.000          0.450
total_charges           0.828            0.450          1.000
```

```
import numpy as np
from sklearn.linear_model import LinearRegression, Ridge

rng = np.random.default_rng(0)
n = 300
x1 = rng.normal(0, 1, n)
x2 = x1 + rng.normal(0, 0.02, n)      # correlation about 0.9998
y = 2 * x1 + rng.normal(0, 0.3, n)

print('correlation between the two features: %.4f' % np.corrcoef(x1, x2)[0, 1])
print('\nrefitting on four different subsamples:')
print('%12s %10s %10s %10s' % ('', 'coef x1', 'coef x2', 'sum'))
for seed in range(4):
    idx = np.random.default_rng(seed).choice(n, 250, replace=False)
    X = np.column_stack([x1, x2])[idx]
    m = LinearRegression().fit(X, y[idx])
    print('%12s %10.3f %10.3f %10.3f'
          % ('sample %d' % seed, m.coef_[0], m.coef_[1], m.coef_.sum()))
```

```
correlation between the two features: 0.9998

refitting on four different subsamples:
                coef x1    coef x2        sum
    sample 0      1.421      0.594      2.015
    sample 1      0.799      1.212      2.011
    sample 2      0.950      1.060      2.010
    sample 3      1.364      0.640      2.004
```

The individual coefficients swing violently, and their sum stays near 2, the truth. The model knows the total effect perfectly well; it simply cannot tell which of two identical columns deserves the credit. Reporting either coefficient on its own would be meaningless.

Measure it with the variance inflation factor

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

```

```
NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
import numpy as np
from sklearn.linear_model import LinearRegression

d = df.dropna(subset=['total_charges'])
cols = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
M = d[cols].to_numpy(dtype=float)

print('%-18s %8s' % ('feature', 'VIF'))
for i, name in enumerate(cols):
    others = np.delete(M, i, axis=1)
    r2 = LinearRegression().fit(others, M[:, i]).score(others, M[:, i])
    print('%-18s %8.2f' % (name, 1 / (1 - r2)))
```

```
feature                VIF
tenure_months         7.03
support_calls         1.02
monthly_charges       2.79
total_charges         8.82
```

How to read a VIF

VIF is how much the variance of a coefficient is inflated by its correlation with the others. Below 5 is fine, above 10 is a problem worth acting on. The fixes, in order of preference: drop one of the pair, combine them into a single meaningful feature, or use ridge, which handles collinearity gracefully by design.

Leakage, which is the more dangerous of the two

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
d = df.dropna(subset=['total_charges']).copy()
d = d[d['tenure_months'] > 0]
d['implied_monthly'] = d['total_charges'] / d['tenure_months']

print('correlation with the target: %.4f'
      % d['implied_monthly'].corr(d[TARGET]))
print('\nfirst five rows:')
print(d[['tenure_months', 'total_charges', 'implied_monthly', TARGET]]
      .head().round(2).to_string(index=False))
```

```
correlation with the target: 0.9956

first five rows:
 tenure_months  total_charges  implied_monthly  monthly_charges
            10         904.14            90.41            90.21
             8         621.20            77.65            81.45
             3         181.14            60.38            57.29
             3          44.38            14.79            15.00
            21         564.82            26.90            25.85
```

`total_charges` divided by `tenure_months` is, almost exactly, `monthly_charges`, because that is how the bill was calculated. A perfectly sensible-looking engineered feature reconstructs the answer.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

honest = Pipeline([('prep', preprocessor()),
                   ('reg', LinearRegression())]).fit(X_tr, y_tr)
print('honest model      R2 %.4f' % r2_score(y_te, honest.predict(X_te)))

leaky_df = df.dropna(subset=['total_charges']).copy()
leaky_df = leaky_df[leaky_df['tenure_months'] > 0]
leaky_df['implied_monthly'] = (leaky_df['total_charges']
                               / leaky_df['tenure_months'])
LNUM = NUM + ['implied_monthly']
Xl, yl = leaky_df[LNUM + CAT], leaky_df[TARGET]
```

```
Xl_tr, Xl_te, yl_tr, yl_te = train_test_split(Xl, yl, test_size=0.25,
                                              random_state=42)
leaky = Pipeline([('prep', preprocessor(num=LNUM)),
                  ('reg', LinearRegression())]).fit(Xl_tr, yl_tr)
print('with the leak     R2 %.4f' % r2_score(yl_te, leaky.predict(Xl_te)))
```

```
honest model      R2 0.8498
with the leak     R2 0.9920
```

Watch Out

An R^2 that jumps to 0.99 is not good news

It is the strongest available signal that a feature knows the answer. The instinct on seeing a score like that should be suspicion, not celebration. Ask of every feature: *would I have this value, for a new customer, at the moment I need the prediction?* If the answer is no, or "only after the outcome is known", it leaks.

Where leakage hides

- **Aggregates computed over the whole dataset**: a per-category mean that included the test rows.
- **Anything recorded after the event**: a cancellation reason code, when predicting cancellation.
- **Identifiers correlated with time**: sequential IDs encode when a row was created.
- **Duplicated rows across the split**: exactly the 138 you removed in week 1.
- **Preprocessing fitted before the split**: the reason every scaler in this course sits inside a pipeline.

Day 6 takeaway

Day 7 A Complete Regression Project

Every decision of the week, applied end to end

A complete regression, start to finish, with every decision this week justified.

1. Frame and split

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
```

```
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
print('train %d rows, test %d rows' % (len(X_tr), len(X_te)))
print('target mean: train %.2f, test %.2f' % (y_tr.mean(), y_te.mean()))
```

```
train 2126 rows, test 709 rows
target mean: train 59.48, test 59.20
```

2. Establish the floor

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.dummy import DummyRegressor
from sklearn.metrics import root_mean_squared_error, r2_score

base = DummyRegressor(strategy='mean').fit(X_tr, y_tr)
pred = base.predict(X_te)
print('predicting the mean: RMSE %.3f   R2 %.4f'
      % (root_mean_squared_error(y_te, pred), r2_score(y_te, pred)))
```

```
predicting the mean: RMSE 23.310   R2 -0.0001
```

R^2 of essentially zero, by construction. Every model from here has to beat an RMSE of about 23.4.

3. Compare candidates by cross-validation

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
```

```
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score, KFold
import numpy as np

cv = KFold(n_splits=5, shuffle=True, random_state=42)
candidates = {
    'linear    ': LinearRegression(),
    'ridge     ': RidgeCV(alphas=np.logspace(-3, 3, 25)),
    'lasso     ': LassoCV(cv=5, random_state=0, max_iter=5000),
    'forest    ': RandomForestRegressor(n_estimators=200, random_state=42),
}
for name, reg in candidates.items():
    pipe = Pipeline([('prep', preprocessor()), ('reg', reg)])
    s = -cross_val_score(pipe, X_tr, y_tr, cv=cv,
                         scoring='neg_root_mean_squared_error')
    print('%s cv RMSE %.4f  +/- %.4f' % (name, s.mean(), s.std()))
```

```
linear     cv RMSE 8.7237  +/- 0.1761
ridge      cv RMSE 8.7234  +/- 0.1735
lasso      cv RMSE 8.7068  +/- 0.1714
forest     cv RMSE 9.9159  +/- 0.2897
```

Cross-validate on the training set only

The test set has not been touched yet and must not be, until the model is chosen. Every comparison, every hyperparameter, every feature decision comes from cross-validation within the training data. The test set is spent the first time you look at it.

4. Fit the winner and score once

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.linear_model import RidgeCV
from sklearn.metrics import (root_mean_squared_error, mean_absolute_error,
                             r2_score)
import numpy as np

final = Pipeline([('prep', preprocessor()),
                  ('reg', RidgeCV(alphas=np.logspace(-3, 3, 25)))])
final.fit(X_tr, y_tr)
pred = final.predict(X_te)

print('TEST SET')
print('  RMSE %.4f' % root_mean_squared_error(y_te, pred))
print('  MAE  %.4f' % mean_absolute_error(y_te, pred))
print('  R2   %.4f' % r2_score(y_te, pred))
print('\nagainst a mean-only RMSE of %.3f' % y_te.std())
```

```
TEST SET
  RMSE 9.0293
  MAE  7.2018
  R2   0.8499

against a mean-only RMSE of 23.325
```

5. Explain it

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.linear_model import RidgeCV
import numpy as np
import pandas as pd

final = Pipeline([('prep', preprocessor()),
                  ('reg', RidgeCV(alphas=np.logspace(-3, 3, 25)))]).fit(X_tr, y_tr)

names = final.named_steps['prep'].get_feature_names_out()
coefs = pd.Series(final.named_steps['reg'].coef_, index=names)
print(coefs.reindex(coefs.abs().sort_values(ascending=False).index)
      .head(8).round(3).to_string())
```

```
cat__internet_service_No internet      -30.107
cat__internet_service_Fibre optic       27.346
cat__internet_service_DSL                2.762
cat__contract_Month-To-Month             0.322
cat__payment_method_Electronic check    -0.304
num__tenure_months                       0.299
cat__contract_One Year                  -0.290
cat__payment_method_Bank transfer        0.215
```

Fibre optic adds about 27 against the baseline and having no internet subtracts about 30, while contract type and payment method barely move the price at all. The model has recovered the pricing structure, which is exactly what the generator in week 1 put there, and note that the

features which drive *churn* are almost irrelevant to *price*. Different target, different important features.

6. Check the residuals one last time

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
df = df.dropna(subset=['monthly_charges'])

NUM = ['tenure_months', 'support_calls']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'monthly_charges'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X = df[NUM + CAT]
y = df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, random_state=42)
from sklearn.linear_model import RidgeCV
import numpy as np
import pandas as pd

final = Pipeline([('prep', preprocessor()),
                  ('reg', RidgeCV(alphas=np.logspace(-3, 3, 25)))]).fit(X_tr, y_tr)
pred = final.predict(X_te)
resid = y_te - pred

print('residual mean %.4f, std %.4f, skew %.4f'
      % (resid.mean(), resid.std(), resid.skew()))
worst = resid.abs().nlargest(3).index
print('\nworst three predictions:')
print(pd.DataFrame({'actual': y_te.loc[worst].round(2),
                    'predicted': pred[[list(y_te.index).index(i) for i in
    ↪ worst]].round(2),
                    'service': df.loc[worst, 'internet_service']}).to_string())
```

```
residual mean -0.4340, std 9.0253, skew 0.0221
```

```
worst three predictions:
      actual  predicted      service
3076   48.61      79.22  Fibre optic
767    52.42      79.49  Fibre optic
132   105.96      78.98  Fibre optic
```

Your assignment

Rebuild this predicting `total_charges` instead. You will find an R^2 above 0.99, because tenure multiplied by monthly charges *is* the total. Then remove both of those features and try again. Write down which version you would deploy, and why the first one is worthless despite scoring better.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. An R-squared of 0.0 means the model is:

    a) Undefined
    b) Perfect
    c) As good as always predicting the mean of the target
    d) Worse than useless

2. Compared with MAE, RMSE:

    a) Is always smaller
    b) Cannot be used for regression
    c) Ignores outliers
    d) Penalises large errors more heavily, because they are squared

3. MAPE is a poor choice when:

    a) The target can be zero or near zero, which makes the percentage explode
    b) There are many features
    c) The model is linear
    d) The target is large

4. Training error keeps falling as polynomial degree rises, while validation error turns upward. That gap is:

    a) Bias
    b) Variance. The model is fitting the sample rather than the pattern
    c) Leakage
    d) Underfitting

5. The practical difference between ridge and lasso is that lasso:

a) Requires no scaling
b) Is faster
c) Can drive coefficients to exactly zero, so it selects features
d) Never overfits

6. Two highly correlated features are given to lasso. The likely result is:

   a) Both are zeroed
   b) The fit fails
   c) Both get equal weight
   d) One is kept and the other zeroed, and which one can change with a small change in the data

7. A model reaches R-squared 0.98 on a business problem where 0.6 would be good. The first thing to suspect is:

   a) Target leakage, a feature that encodes the answer
   b) Too little regularisation
   c) An unlucky split
   d) A very good model

8. Why must scaling happen inside the pipeline rather than before the split?

   a) It is faster
   b) Otherwise the mean and standard deviation are computed using test rows
   c) Scalers cannot handle splits
   d) It changes the column order

Answers:1c, 2d, 3a, 4b, 5c, 6d, 7a, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 5

Classification Fundamentals

Week 5

OBJECTIVES

Logistic regression, kNN, naive Bayes and SVMs, and the threshold decision that matters more than any of them.

Predicting a label rather than a number. This week covers the four classical classifiers and, more importantly, the decision that sits on top of all of them: where to cut the predicted probability. Week 1 left you with a model that missed most churners; day 2 fixes it without retraining anything.

Day 1 Logistic Regression

The sigmoid, log-odds, odds ratios, and regularisation you did not ask for

Linear regression predicts a number on an unbounded scale. A probability lives between 0 and 1. Logistic regression is what you get when you insist on that constraint, and it remains the model to beat on tabular data.

The sigmoid

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

for z in [-6, -2, -1, 0, 1, 2, 6]:
    print('z = %3d  →  probability %.4f' % (z, sigmoid(z)))
```

```
z =  -6  →  probability 0.0025
z =  -2  →  probability 0.1192
z =  -1  →  probability 0.2689
z =   0  →  probability 0.5000
z =   1  →  probability 0.7311
z =   2  →  probability 0.8808
z =   6  →  probability 0.9975
```

Any real number in, a probability out. Zero maps to 0.5, and the curve saturates at both ends, which is why a very confident model needs an enormous change in `z` to become slightly more confident.

Log-odds, which is the scale the coefficients live on

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

print('%10s %10s %12s' % ('probability', 'odds', 'log-odds (z)'))
for prob in [0.1, 0.25, 0.5, 0.75, 0.9]:
    odds = prob / (1 - prob)
    print('%10.2f %10.3f %12.4f' % (prob, odds, np.log(odds)))
```

```
probability       odds log-odds (z)
      0.10      0.111      -2.1972
      0.25      0.333      -1.0986
      0.50      1.000       0.0000
      0.75      3.000       1.0986
      0.90      9.000       2.1972
```

A coefficient is a change in log-odds

That is why they are hard to read directly. Exponentiate one and you get an *odds ratio*: a coefficient of 0.7 means `exp(0.7) = 2.01`, so a one-unit increase roughly doubles the odds. Doubling the odds is not doubling the probability, from 0.1 it goes to 0.18, from 0.5 it goes to 0.67.

Fit it on the churn problem

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, log_loss, accuracy_score

model = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000,
                                             random_state=42))]).fit(X_tr, y_tr)
proba = model.predict_proba(X_te)[:, 1]

print('accuracy %.4f' % accuracy_score(y_te, model.predict(X_te)))
print('ROC AUC  %.4f' % roc_auc_score(y_te, proba))
print('log loss %.4f' % log_loss(y_te, proba))
print('\npredicted probabilities: min %.3f, median %.3f, max %.3f'
      % (proba.min(), np.median(proba), proba.max()))
```

```
accuracy 0.7800
ROC AUC  0.8174
log loss 0.4451

predicted probabilities: min 0.000, median 0.232, max 0.829
```

Read the coefficients as odds ratios

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
```

```
import numpy as np
import pandas as pd

model = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000,
                                             random_state=42))]).fit(X_tr, y_tr)
names = model.named_steps['prep'].get_feature_names_out()
coef = model.named_steps['clf'].coef_[0]

table = pd.DataFrame({'coefficient': coef, 'odds_ratio': np.exp(coef)},
                     index=names).sort_values('coefficient')
print(table.round(3).to_string())
```

```
                                  coefficient  odds_ratio
cat__contract_Two Year                 -1.454       0.234
num__tenure_months                     -0.978       0.376
cat__payment_method_Credit card        -0.205       0.815
cat__payment_method_Bank transfer      -0.174       0.841
cat__internet_service_No internet      -0.163       0.850
cat__has_dependents_Yes                -0.102       0.903
cat__payment_method_Mailed check       -0.088       0.916
cat__internet_service_DSL              -0.060       0.942
cat__has_dependents_No                  0.087       1.091
cat__contract_One Year                  0.128       1.137
cat__internet_service_Fibre optic       0.207       1.230
num__support_calls                      0.279       1.322
num__monthly_charges                    0.436       1.547
cat__payment_method_Electronic check    0.451       1.570
cat__contract_Month-To-Month            1.310       3.706
```

An odds ratio above 1 pushes toward churn, below 1 pushes away. Month-to-month roughly triples the odds against the average; a two-year contract cuts them to about a quarter. Compare that with the generator in week 1: it used +1.55 and -0.85 on exactly those two.

Watch Out

Odds ratios on scaled features are per standard deviation

The numeric columns went through `StandardScaler`, so their coefficients describe a one *standard deviation* change, not one month or one pound. To talk to a business audience in real units, divide the coefficient by the scaler's `scale_` for that column, or fit an unscaled model purely for explanation.

Regularisation is on by default

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
```

```
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
import numpy as np

print('%10s %12s %14s' % ('C', 'test AUC', 'sum |coef|'))
for C_val in [0.001, 0.01, 0.1, 1.0, 100.0]:
    m = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(C=C_val, max_iter=2000,
                                             random_state=42))]).fit(X_tr, y_tr)
    auc = roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])
    print('%10s %12.4f %14.3f'
          % (C_val, auc, np.abs(m.named_steps['clf'].coef_).sum()))
```

```
         C     test AUC     sum |coef|
     0.001       0.8205          0.885
      0.01       0.8230          3.114
       0.1       0.8198          5.147
       1.0       0.8174          6.122
     100.0       0.8169          8.108
```

Watch Out

C is the inverse of alpha

Ridge takes `alpha`, where larger means more penalty. `LogisticRegression` takes `C`, where *smaller* means more penalty. The default of `C=1.0` means you are already regularising whether you meant to or not, which surprises people comparing against an unpenalised implementation elsewhere.

Day 1 takeaway

Day 2 The Decision Threshold

Why 0.5 is almost always wrong, and how to choose the number that is right

Week 1 ended with a model that found fewer than half the churners, and the promise that the model was fine and the threshold was wrong. Today you collect on that.

predict() is predict_proba() plus an arbitrary cut

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
import numpy as np

model = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000,
                                             random_state=42))]).fit(X_tr, y_tr)
proba = model.predict_proba(X_te)[:, 1]

print('predict() agrees with proba > 0.5 on %d of %d rows'
      % ((model.predict(X_te) == (proba > 0.5).astype(int)).sum(), len(proba)))
print('\nrows with probability above:')
for t in [0.2, 0.3, 0.4, 0.5, 0.6]:
    print('  %.1f  %4d  (%.1f%% of the test set)'
          % (t, (proba > t).sum(), 100 * (proba > t).mean()))
```

```
predict() agrees with proba > 0.5 on 750 of 750 rows

rows with probability above:
```

```
0.2   400  (53.3% of the test set)
0.3   294  (39.2% of the test set)
0.4   209  (27.9% of the test set)
0.5   138  (18.4% of the test set)
0.6    70  (9.3% of the test set)
```

0.5 is a default, not a decision

It is only optimal when the classes are balanced and the two kinds of mistake cost the same. Churn is 27 percent, and missing a leaver who was worth keeping costs far more than a wasted retention call. Neither condition holds, so 0.5 is simply the wrong number.

The four outcomes

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, precision_score, recall_score
import numpy as np

model = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000,
                                             random_state=42))]).fit(X_tr, y_tr)
proba = model.predict_proba(X_te)[:, 1]

print('%10s %6s %6s %6s %6s %10s %8s'
      % ('threshold', 'TN', 'FP', 'FN', 'TP', 'precision', 'recall'))
for t in [0.15, 0.2, 0.27, 0.35, 0.5, 0.7]:
```

```
    pred = (proba >= t).astype(int)
    tn, fp, fn, tp = confusion_matrix(y_te, pred).ravel()
    print('%10.2f %6d %6d %6d %6d %10.3f %8.3f'
          % (t, tn, fp, fn, tp,
             precision_score(y_te, pred, zero_division=0),
             recall_score(y_te, pred)))
```

```
 threshold     TN     FP     FN     TP  precision   recall
      0.15    287    262     18    183      0.411    0.910
      0.20    328    221     22    179      0.448    0.891
      0.27    384    165     42    159      0.491    0.791
      0.35    433    116     66    135      0.538    0.672
      0.50    498     51    114     87      0.630    0.433
      0.70    544      5    180     21      0.808    0.104
```

Term	Means	Cost in a churn programme
True positive	Predicted churn, did churn	A save, the point of the exercise
False positive	Predicted churn, stayed	A wasted retention offer
False negative	Predicted stay, churned	A customer lost silently
True negative	Predicted stay, stayed	Nothing spent

Choose the threshold from the cost of being wrong

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import confusion_matrix
import numpy as np

model = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000,
                                             random_state=42))]).fit(X_tr, y_tr)
proba = model.predict_proba(X_te)[:, 1]

OFFER_COST = 30       # what a retention offer costs us
CUSTOMER_VALUE = 250  # what keeping a customer is worth
SAVE_RATE = 0.35      # share of contacted churners we actually keep

best = None
for t in np.arange(0.05, 0.95, 0.01):
    pred = (proba >= t).astype(int)
    tn, fp, fn, tp = confusion_matrix(y_te, pred).ravel()
    value = tp * SAVE_RATE * CUSTOMER_VALUE - (tp + fp) * OFFER_COST
    if best is None or value > best[1]:
        best = (t, value, tp, fp)

print('best threshold %.2f → net value %.0f (contacted %d, saved about %.0f)'
      % (best[0], best[1], best[2] + best[3], best[2] * SAVE_RATE))

pred_default = (proba >= 0.5).astype(int)
tn, fp, fn, tp = confusion_matrix(y_te, pred_default).ravel()
print('at the default 0.5 → net value %.0f'
      % (tp * SAVE_RATE * CUSTOMER_VALUE - (tp + fp) * OFFER_COST))
```

```
best threshold 0.34 → net value 4537 (contacted 260, saved about 49)
at the default 0.5 → net value 3472
```

Same model, same data, no retraining. Moving the threshold is free, and it is the single highest-return change available on most classification projects.

ROC and precision-recall curves

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
```

```
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (roc_curve, precision_recall_curve,
                             roc_auc_score, average_precision_score)
import numpy as np

model = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000,
                                             random_state=42))]).fit(X_tr, y_tr)
proba = model.predict_proba(X_te)[:, 1]

fpr, tpr, _ = roc_curve(y_te, proba)
prec, rec, _ = precision_recall_curve(y_te, proba)
print('ROC AUC              %.4f' % roc_auc_score(y_te, proba))
print('average precision    %.4f' % average_precision_score(y_te, proba))
print('baseline for AP      %.4f  ← the positive class rate' % y_te.mean())
print('baseline for ROC AUC 0.5000')
```

```
ROC AUC              0.8174
average precision    0.5993
baseline for AP      0.2680  ← the positive class rate
baseline for ROC AUC 0.5000
```

Watch Out

ROC AUC flatters models on imbalanced data

The false positive rate has the large negative class in its denominator, so a flood of false positives barely moves it. A model can look strong on ROC and be useless in practice. The precision-recall curve uses no true negatives at all, so it stays honest, and its baseline is the positive rate, 0.27 here, not 0.5. Report both, and lead with average precision when positives are rare.

Day 2 takeaway

Day 3 k-Nearest Neighbours

Voting among similar rows, and why distance stops working in high dimensions

The simplest possible classifier: find the most similar customers you have seen and copy their answer. It has no training phase at all.

How it works

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score, accuracy_score

for k in [1, 5, 25, 101]:
    m = Pipeline([('prep', preprocessor()),
                  ('clf', KNeighborsClassifier(n_neighbors=k))]).fit(X_tr, y_tr)
    print('k=%3d  train acc %.4f   test acc %.4f   test AUC %.4f'
          % (k, accuracy_score(y_tr, m.predict(X_tr)),
             accuracy_score(y_te, m.predict(X_te)),
             roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

```
k=  1  train acc 0.9991   test acc 0.7333   test AUC 0.6507
k=  5  train acc 0.8258   test acc 0.7453   test AUC 0.7517
k= 25  train acc 0.7924   test acc 0.7640   test AUC 0.8023
k=101  train acc 0.7831   test acc 0.7653   test AUC 0.8122
```

k=1 scores perfectly on training data, by definition

The nearest neighbour of a training row is itself, at distance zero. That perfect training accuracy is the purest example of why training scores are meaningless. Larger k averages over more neighbours, smoothing the decision boundary and trading variance for bias.

Scaling is not optional here

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import roc_auc_score

# Two columns on wildly different scales: months against pounds.
WIDE = ['tenure_months', 'total_charges']
Xn_tr, Xn_te = df.loc[X_tr.index, WIDE], df.loc[X_te.index, WIDE]

unscaled = make_pipeline(SimpleImputer(strategy='median'),
                         KNeighborsClassifier(25)).fit(Xn_tr, y_tr)
scaled = make_pipeline(SimpleImputer(strategy='median'), StandardScaler(),
                       KNeighborsClassifier(25)).fit(Xn_tr, y_tr)

print('unscaled AUC %.4f' % roc_auc_score(y_te, unscaled.predict_proba(Xn_te)[:,
    ↪ 1]))
print('scaled   AUC %.4f' % roc_auc_score(y_te, scaled.predict_proba(Xn_te)[:, 1]))
print('\ncolumn ranges:')
print((Xn_tr.max() - Xn_tr.min()).round(1).to_string())
```

```
unscaled AUC 0.6316
scaled   AUC 0.7411

column ranges:
tenure_months      71.0
total_charges    7591.6
```

Eleven points of AUC, from one line. `total_charges` spans 7,592 while tenure spans 71, so unscaled the distance calculation is effectively *only* about total charges, tenure contributes about one percent of the typical gap between two rows and is ignored. Scaling gives each column an equal say. No other algorithm in this course is punished this hard for skipping it.

The curse of dimensionality

```
import numpy as np
```

```

rng = np.random.default_rng(0)
print('%6s %14s %14s %10s' % ('dims', 'nearest', 'furthest', 'ratio'))
for d in [2, 5, 20, 100, 500]:
    pts = rng.uniform(size=(1000, d))
    query = rng.uniform(size=d)
    dist = np.linalg.norm(pts - query, axis=1)
    print('%6d %14.4f %14.4f %10.3f'
          % (d, dist.min(), dist.max(), dist.max() / dist.min()))
```

```
  dims        nearest       furthest      ratio
     2         0.0098         1.3341    135.618
     5         0.1443         1.6328     11.317
    20         1.0474         2.2671      2.164
   100         3.0126         4.8993      1.626
   500         8.1975         9.7549      1.190
```

Watch Out

In high dimensions, everything is equally far away

At two dimensions the furthest point is several times further than the nearest. At 500, the ratio approaches 1: every point is roughly the same distance from every other. "Nearest neighbour" stops meaning anything, and kNN degrades toward guessing. This is why kNN is a poor choice after one-hot encoding produces a wide sparse matrix, and why week 10's dimensionality reduction exists.

Distance weighting

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split
```

```

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score

for weights in ['uniform', 'distance']:
    m = Pipeline([('prep', preprocessor()),
                  ('clf', KNeighborsClassifier(25, weights=weights))]).fit(X_tr,
    ↪ y_tr)
    print('%-9s AUC %.4f'
          % (weights, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

```
uniform   AUC 0.8023
distance  AUC 0.8009
```

`weights='distance'` gives closer neighbours more say, which usually helps when k is large.

	kNN	Logistic regression
Training cost	None	One optimisation
Prediction cost	Scans the training set	One dot product
Memory	Stores all training data	Stores k+1 numbers
Boundary	Arbitrarily shaped	A straight hyperplane
Interpretable	No	Yes, as odds ratios
Needs scaling	Absolutely	For regularisation to be fair

Day 3 takeaway

Day 4 Naive Bayes

A false assumption that ranks well and calibrates badly

Naive Bayes applies the theorem from week 3 directly, under an assumption that is almost always false. It works anyway, and understanding why is worth more than the algorithm itself.

The assumption

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'

```

```python
# Are contract and tenure independent given the class? Week 3 said no.
sub = df[df['churned'] == 1]
a = (sub['contract'] == 'Month-To-Month')
b = (sub['tenure_months'] > 24)
print('among churners:')
print('  P(month-to-month)          %.4f' % a.mean())
print('  P(tenure > 24)             %.4f' % b.mean())
print('  product (if independent)   %.4f' % (a.mean() * b.mean()))
print('  actual joint probability   %.4f' % (a & b).mean())
```

```
among churners:
  P(month-to-month)          0.8545
  P(tenure > 24)             0.1182
  product (if independent)   0.1010
  actual joint probability   0.0945
```

Not independent, not even close. Naive Bayes will assume they are. The surprise is that this often costs less than it should.

Fit it anyway

```python
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, accuracy_score, log_loss

for name, clf in [('naive bayes', GaussianNB()),
```

```
                  ('logistic   ', LogisticRegression(max_iter=1000,
                                                     random_state=42))]:
    m = Pipeline([('prep', preprocessor()), ('clf', clf)]).fit(X_tr, y_tr)
    proba = m.predict_proba(X_te)[:, 1]
    print('%s acc %.4f   AUC %.4f   log loss %.4f'
          % (name, accuracy_score(y_te, m.predict(X_te)),
             roc_auc_score(y_te, proba), log_loss(y_te, proba)))
```

```
naive bayes acc 0.7173   AUC 0.8044   log loss 1.1953
logistic    acc 0.7800   AUC 0.8174   log loss 0.4451
```

Good ranking, bad probabilities

Look at the gap between AUC and log loss. Naive Bayes ranks customers by risk reasonably well, but its actual probability estimates are poor, because multiplying correlated probabilities as though they were independent pushes the result toward 0 or 1. If you only need an ordering it is fine. If you need a number a business can act on, “this customer has a 30 percent chance”. It is not.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.naive_bayes import GaussianNB
import numpy as np

m = Pipeline([('prep', preprocessor()), ('clf', GaussianNB())]).fit(X_tr, y_tr)
proba = m.predict_proba(X_te)[:, 1]
print('share of predictions below 0.01: %.3f' % (proba < 0.01).mean())
print('share of predictions above 0.99: %.3f' % (proba > 0.99).mean())
```

```
print('share between 0.2 and 0.8:       %.3f'
      % ((proba > 0.2) & (proba < 0.8)).mean())
```

```
share of predictions below 0.01: 0.320
share of predictions above 0.99: 0.068
share between 0.2 and 0.8:       0.131
```

Most predictions are pinned at the extremes. A well-calibrated model on a 27 percent problem should produce a spread of intermediate probabilities. Week 7 shows how to measure and repair calibration.

Which variant for which data

Variant	Features	Typical use
`GaussianNB`	Continuous	Numeric tabular data
`MultinomialNB`	Counts	Word counts in text classification
`BernoulliNB`	Binary	Presence or absence flags
`CategoricalNB`	Categorical codes	Survey responses

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

notes = ['billing error on my invoice', 'charged twice this month',
         'internet connection keeps dropping', 'speed is very slow',
         'wrong amount on my bill', 'internet drops every evening']
labels = ['billing', 'billing', 'technical', 'technical',
          'billing', 'technical']

m = make_pipeline(CountVectorizer(), MultinomialNB()).fit(notes, labels)
for text in ['my bill is wrong again', 'internet connection drops',
             'charged the wrong amount']:
    print('%-32s → %s' % (text, m.predict([text])[0]))
```

```
my bill is wrong again           → billing
internet connection drops        → technical
charged the wrong amount         → billing
```

This is what naive Bayes is genuinely good at. Text has thousands of sparse features, the independence assumption is less damaging when each word contributes a little, and training takes milliseconds. It remains a sensible baseline for text classification.

Day 4 takeaway

Day 5 Support Vector Machines

Margins, kernels, and the two parameters that must be tuned together

Logistic regression draws a boundary that separates the classes. A support vector machine

asks a better question: of all the boundaries that separate them, which one leaves the most room on either side?

The margin

```
import numpy as np
from sklearn.svm import SVC
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=200, centers=2, cluster_std=1.2, random_state=7)
m = SVC(kernel='linear', C=1.0).fit(X, y)

print('training points  %d' % len(X))
print('support vectors  %d' % len(m.support_))
print('so the boundary depends on %.1f%% of the data'
      % (100 * len(m.support_) / len(X)))
```

```
training points  200
support vectors  3
so the boundary depends on 1.5% of the data
```

The kernel trick

Some classes cannot be separated by a straight line at all. A kernel maps the data into a higher-dimensional space where they can be, without ever computing the coordinates in that space.

```
import numpy as np
from sklearn.svm import SVC
from sklearn.datasets import make_circles
from sklearn.model_selection import cross_val_score

X, y = make_circles(n_samples=400, factor=0.4, noise=0.12, random_state=0)
print('one class forms a ring around the other -- no line can split it')

for kernel in ['linear', 'poly', 'rbf']:
    acc = cross_val_score(SVC(kernel=kernel, random_state=0), X, y, cv=5).mean()
    print('  %-8s accuracy %.4f' % (kernel, acc))
```

```
one class forms a ring around the other -- no line can split it
  linear   accuracy 0.5450
  poly     accuracy 0.6250
  rbf      accuracy 0.9975
```

The linear kernel manages barely better than guessing, because no straight line can separate a ring from its centre. The RBF kernel handles it almost perfectly.

C and gamma

Parameter	Low value	High value
`C`	Wide margin, tolerates mistakes (more bias)	Narrow margin, fits training data hard (more variance)
`gamma`	Each point influences a wide area (smoother)	Influence is local, boundary wraps individual points (overfits)

```
import numpy as np
from sklearn.svm import SVC
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X, y = make_circles(n_samples=400, factor=0.4, noise=0.12, random_state=0)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.3, random_state=0)

print('%8s %8s %10s %10s' % ('C', 'gamma', 'train', 'test'))
for C_val, g in [(1, 0.1), (1, 1), (1, 100), (1000, 100)]:
    m = SVC(C=C_val, gamma=g, random_state=0).fit(X_tr, y_tr)
    print('%8s %8s %10.4f %10.4f'
          % (C_val, g, accuracy_score(y_tr, m.predict(X_tr)),
             accuracy_score(y_te, m.predict(X_te))))
```

```
       C    gamma      train       test
       1      0.1     0.9750     0.9583
       1        1     1.0000     1.0000
       1      100     1.0000     0.9917
    1000      100     1.0000     0.9917
```

At gamma 100 the model scores near-perfectly on training data and worse on test: each support vector now influences only its immediate neighbourhood, so the boundary wraps individual points. Classic overfitting, and visible only because we looked at both columns.

On the churn problem

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
```

```
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
import time

for name, clf in [('svm rbf   ', SVC(probability=True, random_state=42)),
                  ('logistic  ', LogisticRegression(max_iter=1000,
                                                    random_state=42))]:
    t = time.perf_counter()
    m = Pipeline([('prep', preprocessor()), ('clf', clf)]).fit(X_tr, y_tr)
    elapsed = time.perf_counter() - t
    auc = roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])
    print('%s AUC %.4f   fitted in %.2fs' % (name, auc, elapsed))
```

```
svm rbf    AUC 0.7804   fitted in 0.50s
logistic   AUC 0.8174   fitted in 0.02s
```

Watch Out

SVMs scale badly

Training cost grows between quadratically and cubically with the number of rows. At three thousand rows that is a second; at a million it is impractical. `probability=True` makes it worse still, because it fits an extra calibration model by internal cross-validation. For large tabular datasets, reach for the gradient boosting of week 6 instead.

Day 5 takeaway

Day 6 More Than Two Classes

One-vs-rest, confusion matrices, averaging traps and softmax

Everything so far has had two classes. Most real problems have more: which of nine defect types, which of five support queues, which of three risk bands.

Two strategies

Strategy	Models fitted	Each model learns	Cost
One-vs-rest	One per class	This class against all others	n models
One-vs-one	One per pair	This class against that one	n(n-1)/2 models

```
from sklearn.datasets import load_wine
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier, OneVsOneClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score

X, y = load_wine(return_X_y=True)
print('classes %d, rows %d, features %d' % (len(set(y)), *X.shape))

base = LogisticRegression(max_iter=5000, random_state=0)
for name, clf in [('built-in    ', base),
                  ('one-vs-rest ', OneVsRestClassifier(base)),
                  ('one-vs-one  ', OneVsOneClassifier(base))]:
    acc = cross_val_score(make_pipeline(StandardScaler(), clf), X, y, cv=5).mean()
    print('%s accuracy %.4f' % (name, acc))
```

```
classes 3, rows 178, features 13
built-in     accuracy 0.9832
one-vs-rest  accuracy 0.9889
one-vs-one   accuracy 0.9833
```

scikit-learn handles multi-class automatically for most estimators, so you rarely wrap anything by hand. Knowing the strategies matters when you need to explain why a model produced an inconsistent set of scores.

The confusion matrix earns its name here

```
from sklearn.datasets import load_wine
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

X, y = load_wine(return_X_y=True)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.35,
                                          stratify=y, random_state=0)
m = make_pipeline(StandardScaler(),
                  LogisticRegression(max_iter=5000, random_state=0)).fit(X_tr, y_tr)

print('rows = truth, columns = prediction')
print(confusion_matrix(y_te, m.predict(X_te)))
print()
print(classification_report(y_te, m.predict(X_te), digits=3))
```

```
rows = truth, columns = prediction
[[21  0  0]
 [ 1 24  0]
 [ 0  0 17]]

              precision    recall  f1-score   support

           0      0.955     1.000     0.977        21
           1      1.000     0.960     0.980        25
           2      1.000     1.000     1.000        17

    accuracy                          0.984        63
   macro avg      0.985     0.987     0.985        63
weighted avg      0.985     0.984     0.984        63
```

The off-diagonal cells tell you which classes the model confuses, and that is usually more actionable than the overall accuracy. Two classes that get mixed up may need a feature that distinguishes them specifically.

Averaging, and why the choice matters

```python
import numpy as np
from sklearn.metrics import f1_score, precision_score, recall_score

# 100 of class 0, 100 of class 1, 10 of class 2 (the rare, important one)
truth = np.array([0] * 100 + [1] * 100 + [2] * 10)
pred = truth.copy()
pred[200:] = 0            # every single class-2 row misclassified

for avg in ['micro', 'macro', 'weighted']:
    print('%-9s F1 %.4f' % (avg, f1_score(truth, pred, average=avg)))
```

```
micro     F1 0.9524
macro     F1 0.6508
weighted  F1 0.9297
```

Watch Out

micro and weighted hide a total failure on a rare class

The model gets every one of the rare class wrong, and micro-averaged F1 still reads 0.95 because the rare class is 5 percent of the rows. Macro-averaging treats every class equally regardless of size and drops to 0.65, which is the honest figure. When the rare class is the one you care about, fraud, defects, disease, use `macro`, or report per-class figures.

Probabilities across several classes

```python
import numpy as np

def softmax(z):
    z = z - z.max()          # subtract the max for numerical stability
    e = np.exp(z)
```

```
    return e / e.sum()

scores = np.array([2.0, 1.0, 0.1])
probs = softmax(scores)
print('raw scores  ', scores)
print('probabilities', probs.round(4))
print('they sum to  %.4f' % probs.sum())

big = np.array([1000.0, 999.0, 998.0])
print('\nwithout the max subtraction exp(1000) overflows;')
print('with it:', softmax(big).round(4))
```

```
raw scores   [2.  1.  0.1]
probabilities [0.659  0.2424 0.0986]
they sum to  1.0000

without the max subtraction exp(1000) overflows;
with it: [0.6652 0.2447 0.09  ]
```

Softmax is the multi-class sigmoid, and it appears again as the output layer of every classification network in week 12. Subtracting the maximum changes nothing mathematically and prevents overflow, which is why every real implementation does it.

Day 6 takeaway

Day 7 Comparing Classifiers Honestly

Five algorithms, cross-validation, and whether the differences are real

Five algorithms, one problem, one honest comparison, including how long each takes and whether the differences mean anything.

Compare by cross-validation

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
```

```
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_validate, StratifiedKFold
import numpy as np

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
candidates = {
    'baseline  ': DummyClassifier(strategy='most_frequent'),
    'logistic  ': LogisticRegression(max_iter=1000, random_state=42),
    'kNN k=25  ': KNeighborsClassifier(25),
    'naive bayes': GaussianNB(),
    'svm rbf   ': SVC(probability=True, random_state=42),
    'tree d=5  ': DecisionTreeClassifier(max_depth=5, random_state=42),
}

print('%-12s %14s %16s %10s' % ('model', 'ROC AUC', 'avg precision', 'fit secs'))
for name, clf in candidates.items():
    pipe = Pipeline([('prep', preprocessor()), ('clf', clf)])
    r = cross_validate(pipe, X_tr, y_tr, cv=cv,
                       scoring=['roc_auc', 'average_precision'])
    print('%-12s %.4f +/-%.3f %10.4f +/-%.3f %10.2f'
          % (name, r['test_roc_auc'].mean(), r['test_roc_auc'].std(),
             r['test_average_precision'].mean(),
             r['test_average_precision'].std(), r['fit_time'].mean()))
```

```
model               ROC AUC    avg precision   fit secs
baseline     0.5000 +/-0.000     0.2680 +/-0.001       0.01
logistic     0.8208 +/-0.014     0.5974 +/-0.029       0.03
kNN k=25     0.8001 +/-0.010     0.5659 +/-0.022       0.01
naive bayes  0.8098 +/-0.006     0.5774 +/-0.027       0.01
svm rbf      0.7944 +/-0.020     0.6017 +/-0.034       0.34
tree d=5     0.7886 +/-0.014     0.5161 +/-0.025       0.01
```

Read the standard deviations before declaring a winner

If two models differ by less than the spread across folds, you have not shown one is better. Week 3's bootstrap made the same point about a single test set. The honest summary is usually "these three are equivalent, and one of them is ten times faster".

Is the difference real?

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score, StratifiedKFold
from scipy import stats
import numpy as np

cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
a = cross_val_score(Pipeline([('prep', preprocessor()),
                              ('clf', LogisticRegression(max_iter=1000,
                                                         random_state=42))]),
                    X_tr, y_tr, cv=cv, scoring='roc_auc')
b = cross_val_score(Pipeline([('prep', preprocessor()),
                              ('clf', SVC(probability=True, random_state=42))]),
                    X_tr, y_tr, cv=cv, scoring='roc_auc')

print('logistic %.4f +/- %.4f' % (a.mean(), a.std()))
print('svm      %.4f +/- %.4f' % (b.mean(), b.std()))
t, pval = stats.ttest_rel(a, b)
print('\npaired t-test on the same folds: p = %.4f' % pval)
```

```
logistic 0.8228 +/- 0.0175
svm      0.7926 +/- 0.0247

paired t-test on the same folds: p = 0.0002
```

A paired test compares the two models fold by fold, which removes the variation caused by some folds simply being harder. It is the right test here precisely because both models saw identical splits.

Choosing

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (roc_auc_score, average_precision_score,
                             confusion_matrix, classification_report)
import numpy as np

final = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000,
                                             random_state=42))]).fit(X_tr, y_tr)
proba = final.predict_proba(X_te)[:, 1]

# The threshold chosen on day 2, applied once to held-out data.
OFFER_COST, VALUE, SAVE_RATE = 30, 250, 0.35
best_t = max(np.arange(0.05, 0.95, 0.01),
             key=lambda t: (lambda c: c[3] * SAVE_RATE * VALUE
                            - (c[3] + c[1]) * OFFER_COST)(
                 confusion_matrix(y_te, (proba >= t).astype(int)).ravel()))

print('TEST SET')
print('  ROC AUC           %.4f' % roc_auc_score(y_te, proba))
print('  average precision %.4f' % average_precision_score(y_te, proba))
```

```python
print('  chosen threshold  %.2f' % best_t)
print()
print(classification_report(y_te, (proba >= best_t).astype(int),
                            target_names=['stayed', 'churned'], digits=3))
```

```
TEST SET
  ROC AUC           0.8174
  average precision 0.5993
  chosen threshold  0.34

              precision    recall  f1-score   support

      stayed      0.878     0.783     0.828       549
     churned      0.542     0.701     0.612       201

    accuracy                         0.761       750
   macro avg      0.710     0.742     0.720       750
weighted avg      0.788     0.761     0.770       750
```

Recall on the churned class is now far above what the default threshold gave in week 1, at the cost of precision, which is exactly the trade the business case asked for.

Your assignment

Take the two best models from the comparison and average their predicted probabilities. Score the average. You will often find it beats both. That is ensembling, and week 6 explains why it works. Then check whether the improvement is larger than the fold-to-fold spread before you believe it.

> **Day 7 takeaway**

Self-Assessment

Try It Yourself

1. Logistic regression's coefficients are additive on the scale of:

   a) Probability
   b) Log-odds
   c) Percentage points
   d) Counts

2. Why is a decision threshold of 0.5 usually wrong?

   a) It favours the majority class
   b) Probabilities are never exactly 0.5
   c) It implicitly assumes a false positive and a false negative cost the same
   d) The model is not calibrated

3. k-nearest neighbours degrades in high dimensions because:

   a) It cannot handle categories

b) k must grow with dimension
c) It runs out of memory
d) Distances between all points become similar, so "nearest" stops meaning anything

4. Naive Bayes assumes features are independent given the class. In practice this makes it:

   a) Often a decent ranker but a poor estimator of probability
   b) Perfectly calibrated
   c) Slow
   d) Useless

5. For an SVM with an RBF kernel, `C` and `gamma`:

   a) Can be tuned one at a time
   b) Interact, so they must be tuned together
   c) Have no effect on accuracy
   d) Are set automatically

6. Macro-averaged F1 differs from micro-averaged F1 in that macro:

   a) Only applies to binary problems
   b) Weights each class by its size
   c) Gives every class equal weight, so a small class can drag the score down
   d) Is always higher

7. Which model family is most affected by failing to scale the features?

   a) Random forest
   b) Gradient boosting
   c) Decision tree
   d) k-nearest neighbours

8. Five classifiers score within 0.01 of each other in cross-validation. The correct conclusion is:

   a) The differences may well be noise; check the spread across folds before declaring a winner
   b) More folds are needed
   c) The data is broken
   d) The best one wins

Answers:1b, 2c, 3d, 4a, 5b, 6c, 7d, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 6

Trees, Forests and Boosting

Week 6

OBJECTIVES

Decision trees, bagging, random forests and gradient boosting, and an honest look at when they lose to a straight line.

Tree ensembles win most tabular competitions, and this week builds them up from a single greedy split to boosted forests. It also ends by settling the question week 1 left open: why the random forest scored worse than logistic regression on our churn data, and what that tells you about choosing an algorithm at all.

Day 1 How a Decision Tree Decides

Impurity, greedy splitting, and the only model you can read end to end

Every model so far draws a single boundary through the whole space. A decision tree does something different: it asks a question, splits the data, and asks a different question of each half.

How a split gets chosen

The tree considers every feature and every possible cut point, and keeps the one that most reduces impurity.

```
import numpy as np

def gini(labels):
    labels = np.asarray(labels)
    if len(labels) == 0:
        return 0.0
    p1 = labels.mean()
    return 1 - (p1 ** 2 + (1 - p1) ** 2)

print('all one class      %.4f' % gini([1, 1, 1, 1]))
print('three to one       %.4f' % gini([1, 1, 1, 0]))
print('even mix           %.4f' % gini([1, 1, 0, 0]))
```

```
all one class      0.0000
three to one       0.3750
even mix           0.5000
```

```
import numpy as np
import pandas as pd
```

```

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
import numpy as np

def gini(labels):
    labels = np.asarray(labels)
    if len(labels) == 0:
        return 0.0
    p1 = labels.mean()
    return 1 - (p1 ** 2 + (1 - p1) ** 2)

y = df[TARGET].to_numpy()
x = df['tenure_months'].to_numpy()
parent = gini(y)

print('impurity before any split: %.4f' % parent)
print('\n%12s %10s %10s %12s' % ('split at', 'left n', 'right n', 'gain'))
best = None
for cut in [6, 12, 18, 24, 36, 48]:
    left, right = y[x <= cut], y[x > cut]
    weighted = (len(left) * gini(left) + len(right) * gini(right)) / len(y)
    gain = parent - weighted
    if best is None or gain > best[1]:
        best = (cut, gain)
    print('%12d %10d %10d %12.5f' % (cut, len(left), len(right), gain))

print('\nbest of these: tenure <= %d, gain %.5f' % best)
```

```
impurity before any split: 0.3924

    split at     left n    right n         gain
           6        345       2655      0.00723
          12        949       2051      0.02653
          18       1545       1455      0.03244
          24       1918       1082      0.03664
          36       2420        580      0.02537
          48       2686        314      0.01562

best of these: tenure <= 24, gain 0.03664
```

That is the entire algorithm, applied recursively. Split, then repeat on each side, until a stopping rule fires.

Let scikit-learn do it and read the result

```
import numpy as np
import pandas as pd

```

```
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.tree import DecisionTreeClassifier, export_text

tree = Pipeline([('prep', preprocessor(scale=False)),
                 ('clf', DecisionTreeClassifier(max_depth=3,
                                                random_state=42))]).fit(X_tr, y_tr)
names = list(tree.named_steps['prep'].get_feature_names_out())
print(export_text(tree.named_steps['clf'], feature_names=names))
```

```
|--- cat__contract_Month-To-Month <= 0.50
|   |--- num__tenure_months <= 17.50
|   |   |--- cat__contract_One Year <= 0.50
|   |   |   |--- class: 0
|   |   |--- cat__contract_One Year >  0.50
|   |   |   |--- class: 0
|   |--- num__tenure_months >  17.50
|   |   |--- cat__contract_Two Year <= 0.50
|   |   |   |--- class: 0
|   |   |--- cat__contract_Two Year >  0.50
|   |   |   |--- class: 0
|--- cat__contract_Month-To-Month >  0.50
|   |--- num__monthly_charges <= 51.02
|   |   |--- num__tenure_months <= 9.50
|   |   |   |--- class: 0
|   |   |--- num__tenure_months >  9.50
|   |   |   |--- class: 0
|   |--- num__monthly_charges >  51.02
|   |   |--- num__tenure_months <= 32.50
|   |   |   |--- class: 1
|   |   |--- num__tenure_months >  32.50
```

```
|   |   |   |--- class: 0
```

This is why people like trees

You can read the whole model. Every prediction is a path from the root to a leaf, and you can print that path for any customer and hand it to someone who has never heard of machine learning. No other model in this course offers that.

Trees do not need scaling, and barely care about outliers

```python
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score

for scale in [True, False]:
    m = Pipeline([('prep', preprocessor(scale=scale)),
                  ('clf', DecisionTreeClassifier(max_depth=5,
                                                 random_state=42))]).fit(X_tr, y_tr)
    print('scaled=%-6s AUC %.6f'
          % (scale, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

```
scaled=True   AUC 0.774524
scaled=False  AUC 0.774524
```

Identical to six decimal places. A split asks “is this value above the threshold”, and scaling preserves order, so it changes nothing. The same reasoning is why the 97-call outlier in week 2 moved a tree not at all.

Gini or entropy

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score

for criterion in ['gini', 'entropy', 'log_loss']:
    m = Pipeline([('prep', preprocessor(scale=False)),
                  ('clf', DecisionTreeClassifier(criterion=criterion,
                                                 max_depth=5,
                                                 random_state=42))]).fit(X_tr, y_tr)
    print('%-9s AUC %.4f'
          % (criterion, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

```
gini      AUC 0.7745
entropy   AUC 0.7684
log_loss  AUC 0.7684
```

Practically indistinguishable, which is the usual result. Do not spend time tuning the criterion; spend it on depth, which is tomorrow.

Day 1 takeaway

Day 2 Depth, Pruning and Instability

Stopping a tree memorising, and the weakness that becomes tomorrow's strength

Left alone, a tree will keep splitting until every leaf is pure. That means it memorises the training set perfectly and generalises terribly.

Watch it happen

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score

print('%8s %10s %10s %8s %8s' % ('depth', 'train', 'test', 'gap', 'leaves'))
for depth in [1, 2, 3, 5, 8, 12, None]:
    m = Pipeline([('prep', preprocessor(scale=False)),
                  ('clf', DecisionTreeClassifier(max_depth=depth,
                                                 random_state=42))]).fit(X_tr, y_tr)
    tr = roc_auc_score(y_tr, m.predict_proba(X_tr)[:, 1])
    te = roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])
    print('%8s %10.4f %10.4f %8.4f %8d'
```

```
          % (depth, tr, te, tr - te, m.named_steps['clf'].get_n_leaves()))
```

```
 depth      train       test        gap   leaves
     1     0.7057     0.7056     0.0001        2
     2     0.7660     0.7733    -0.0072        4
     3     0.7980     0.7774     0.0206        8
     5     0.8409     0.7745     0.0664       29
     8     0.9047     0.7309     0.1738      116
    12     0.9779     0.6763     0.3016      311
  None     1.0000     0.6291     0.3709      539
```

Unlimited depth gives a perfect training score and the worst test score of the lot. Hundreds of leaves, most holding a single customer. The tree has recorded the training set rather than learned from it.

The four ways to stop it

Parameter	Controls	Sensible starting point
`max_depth`	How many questions deep	3 to 8 for a single tree
`min_samples_leaf`	Smallest allowed leaf	1 to 5 percent of rows
`min_samples_split`	Smallest node worth splitting	Twice the leaf minimum
`max_leaf_nodes`	Total leaves, grown best-first	An alternative to depth

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score

print('%18s %10s %10s' % ('min_samples_leaf', 'test AUC', 'leaves'))
for leaf in [1, 5, 20, 50, 150]:
    m = Pipeline([('prep', preprocessor(scale=False)),
                  ('clf', DecisionTreeClassifier(min_samples_leaf=leaf,
                                                 random_state=42))]).fit(X_tr, y_tr)
    print('%18d %10.4f %10d'
          % (leaf, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1]),
             m.named_steps['clf'].get_n_leaves()))
```

```
 min_samples_leaf   test AUC     leaves
                1     0.6291        539
                5     0.7260        232
               20     0.7630         75
               50     0.7915         31
              150     0.7974         12
```

min_samples_leaf is the better knob

Depth limits every branch equally, whether or not that branch had enough data to justify going deeper. A minimum leaf size adapts: dense regions of the data get to split further, sparse ones stop early. If you only tune one parameter on a tree, tune this one.

Cost-complexity pruning

Rather than guessing a limit in advance, grow the tree fully and then cut back the branches that do not pay for themselves.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
```

```
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score
import numpy as np

prep = preprocessor(scale=False).fit(X_tr)
Xt_tr, Xt_te = prep.transform(X_tr), prep.transform(X_te)

path = DecisionTreeClassifier(random_state=42).cost_complexity_pruning_path(
    Xt_tr, y_tr)
alphas = path.ccp_alphas[:max(1, len(path.ccp_alphas) // 8)][:8]

print('%12s %10s %10s %8s' % ('ccp_alpha', 'train', 'test', 'leaves'))
for a in alphas:
    t = DecisionTreeClassifier(random_state=42, ccp_alpha=a).fit(Xt_tr, y_tr)
    print('%12.6f %10.4f %10.4f %8d'
          % (a, roc_auc_score(y_tr, t.predict_proba(Xt_tr)[:, 1]),
             roc_auc_score(y_te, t.predict_proba(Xt_te)[:, 1]),
             t.get_n_leaves()))
```

```
   ccp_alpha      train       test   leaves
    0.000000     1.0000     0.6291      539
    0.000111     1.0000     0.6308      537
    0.000221     0.9999     0.6071      533
    0.000254     0.9999     0.6093      530
    0.000267     0.9999     0.6141      521
    0.000267     0.9999     0.6141      521
    0.000274     0.9998     0.6143      518
    0.000279     0.9998     0.6199      515
```

As alpha rises the tree shrinks and the training score falls, while the test score improves up to a point. Pruning finds a smaller and better tree than growing an unrestricted one ever could.

A single tree is unstable

The measure that matters is not whether the printed tree looks different, but whether it *predicts* differently. Fit trees on overlapping samples and count how often two of them disagree about the same customer.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
```

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
import numpy as np
from itertools import combinations

def disagreement(depth, n_models=6):
    probas = []
    for seed in range(n_models):
        Xs, _, ys, _ = train_test_split(X_tr, y_tr, train_size=0.7,
                                        random_state=seed)
        m = Pipeline([('prep', preprocessor(scale=False)),
                      ('clf', DecisionTreeClassifier(max_depth=depth,
                                                     random_state=42))]).fit(Xs, ys)
        probas.append(m.predict_proba(X_te)[:, 1])
    label = np.mean([((a > 0.5) != (b > 0.5)).mean()
                     for a, b in combinations(probas, 2)])
    prob = np.mean([np.abs(a - b).mean() for a, b in combinations(probas, 2)])
    return label, prob

print('%10s %16s %16s' % ('max_depth', 'labels differ', 'mean |p1 - p2|'))
for depth in [2, 4, 8, 12, None]:
    label, prob = disagreement(depth)
    print('%10s %15.1f%% %16.4f' % (depth, 100 * label, prob))
```

```
 max_depth    labels differ   mean |p1 - p2|
         2            19.9%           0.0342
         4            15.2%           0.0899
         8            18.8%           0.1741
        12            23.5%           0.2371
      None            23.6%           0.2366
```

Watch Out

Two trees on 70 percent of the same data, disagreeing on a quarter of customers

Read the probability column first: it rises steadily with depth, from 0.03 to 0.24. The greedy algorithm commits at each step with no way to reconsider, so once two candidate splits are close, a handful of rows decides which wins and every branch below inherits that coin toss. The deeper the tree, the more of those coin tosses accumulate.

The label column is not monotone, and the exception is worth understanding. A depth-2 tree has four leaves, so its probabilities are four numbers, extremely stable. But if one of those numbers sits near 0.5, a small shift flips an entire leaf's worth of customers at once. Stable probabilities, unstable labels. It is a compact reminder that thresholding throws information away.

Either way, this variance is what tomorrow exploits. If a deep tree disagrees with itself depending on which rows it saw, build hundreds on different samples and average them: the disagreement cancels, the agreement survives.

Day 2 takeaway

Day 3 Bagging and Random Forests

Averaging away instability, and the validation set you get for free

Yesterday ended on instability. Today that becomes the mechanism: average many unstable models and the noise cancels while the signal survives.

Bagging

```
import numpy as np

rng = np.random.default_rng(0)
n = 1000
rows = np.arange(n)
sample = rng.choice(rows, size=n, replace=True)

unique = len(np.unique(sample))
print('bootstrap sample of %d drawn from %d rows' % (n, n))
print('distinct rows included: %d (%.1f%%)' % (unique, 100 * unique / n))
print('left out entirely:      %d (%.1f%%)' % (n - unique, 100 * (n - unique) / n))
print('\ntheory says 1 - 1/e = %.3f are included' % (1 - np.exp(-1)))
```

```
bootstrap sample of 1000 drawn from 1000 rows
distinct rows included: 622 (62.2%)
left out entirely:      378 (37.8%)

theory says 1 - 1/e = 0.632 are included
```

About 63 percent of rows appear in any given bootstrap sample. The 37 percent left out are that tree's *out-of-bag* rows, and they give you a free validation set.

Random forests add a second source of randomness

Bagging alone leaves the trees correlated: if one feature is much stronger than the rest, every tree splits on it first. A random forest fixes that by offering each split only a random subset of features.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score

models = {
    'single tree ': DecisionTreeClassifier(random_state=42),
    'bagging x200': BaggingClassifier(DecisionTreeClassifier(random_state=42),
                                      n_estimators=200, random_state=42),
    'forest x200 ': RandomForestClassifier(n_estimators=200, random_state=42),
}
for name, clf in models.items():
    m = Pipeline([('prep', preprocessor(scale=False)), ('clf', clf)]).fit(X_tr,
    ↪ y_tr)
    print('%s AUC %.4f'
          % (name, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

```
single tree  AUC 0.6291
bagging x200 AUC 0.7699
forest x200  AUC 0.7788
```

The out-of-bag score

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score, accuracy_score

m = Pipeline([('prep', preprocessor(scale=False)),
              ('clf', RandomForestClassifier(n_estimators=300, oob_score=True,
                                             random_state=42))]).fit(X_tr, y_tr)
forest = m.named_steps['clf']
print('out-of-bag accuracy %.4f' % forest.oob_score_)
print('test accuracy       %.4f' % accuracy_score(y_te, m.predict(X_te)))
```

```
out-of-bag accuracy 0.7560
test accuracy       0.7587
```

A validation score for free

Each tree is scored on the rows it never saw, and those scores are pooled. You get something close to cross-validation without fitting anything extra. Useful when a full cross-validation would be expensive, though it is slightly pessimistic because each row is judged by only about a third of the forest.

What to tune

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score

print('%12s %10s' % ('n_estimators', 'test AUC'))
for n in [5, 25, 100, 300, 800]:
    m = Pipeline([('prep', preprocessor(scale=False)),
                  ('clf', RandomForestClassifier(n_estimators=n,
                                                 random_state=42))]).fit(X_tr, y_tr)
    print('%12d %10.4f'
          % (n, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))

print('\n%18s %10s' % ('min_samples_leaf', 'test AUC'))
for leaf in [1, 5, 20, 50]:
    m = Pipeline([('prep', preprocessor(scale=False)),
                  ('clf', RandomForestClassifier(n_estimators=300,
                                                 min_samples_leaf=leaf,
                                                 random_state=42))]).fit(X_tr, y_tr)
    print('%18d %10.4f'
          % (leaf, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

```
n_estimators   test AUC
           5     0.7205
```

```
         25     0.7671
        100     0.7728
        300     0.7776
        800     0.7779

min_samples_leaf   test AUC
               1     0.7776
               5     0.7989
              20     0.8098
              50     0.8158
```

Watch Out

More trees never overfit; deeper trees still can

Adding trees to a forest cannot make it worse, the score plateaus and stays there, so `n_estimators` is a budget decision, not a tuning decision. The parameters that genuinely matter are `min_samples_leaf` and `max_features`. This is the opposite of boosting, where more rounds absolutely can overfit.

Day 3 takeaway

Day 4 Boosting

Fitting each model to the last one's mistakes, implemented then industrialised

A forest builds hundreds of independent trees and averages them. Boosting builds them in sequence, each one working on what the previous ones got wrong.

The idea, implemented

```python
import numpy as np
from sklearn.tree import DecisionTreeRegressor

rng = np.random.default_rng(0)
x = rng.uniform(0, 6, 300).reshape(-1, 1)
y = np.sin(1.5 * x.ravel()) + 0.3 * x.ravel() + rng.normal(0, 0.2, 300)

prediction = np.full(len(y), y.mean())    # start with the mean
lr = 0.3

print('%8s %12s' % ('round', 'train MSE'))
print('%8s %12.5f' % ('start', ((y - prediction) ** 2).mean()))
for r in range(1, 41):
    residual = y - prediction             # what we still get wrong
    stump = DecisionTreeRegressor(max_depth=2).fit(x, residual)
    prediction = prediction + lr * stump.predict(x)
    if r in (1, 5, 10, 20, 40):
        print('%8d %12.5f' % (r, ((y - prediction) ** 2).mean()))
```

```
 round    train MSE
 start      0.86697
     1      0.49189
     5      0.10329
    10      0.04882
    20      0.03073
    40      0.02490
```

Thirty lines, and it is the core of the algorithm that wins most tabular competitions.

The library versions

```python
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import (AdaBoostClassifier, GradientBoostingClassifier,
                              HistGradientBoostingClassifier,
                              RandomForestClassifier)
from sklearn.metrics import roc_auc_score
import time

models = {
    'random forest': RandomForestClassifier(n_estimators=300, random_state=42),
    'adaboost     ': AdaBoostClassifier(n_estimators=200, random_state=42),
    'grad boosting': GradientBoostingClassifier(random_state=42),
    'hist grad    ': HistGradientBoostingClassifier(random_state=42),
}
for name, clf in models.items():
```

```
    t = time.perf_counter()
    m = Pipeline([('prep', preprocessor(scale=False)), ('clf', clf)]).fit(X_tr,
    ↪ y_tr)
    el = time.perf_counter() - t
    print('%s AUC %.4f   %.2fs'
          % (name, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1]), el))
```

```
random forest AUC 0.7776   2.00s
adaboost      AUC 0.8098   1.19s
grad boosting AUC 0.8034   0.74s
hist grad     AUC 0.7797   5.20s
```

Learning rate against number of rounds

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import roc_auc_score

print('%8s %8s %10s %10s' % ('lr', 'rounds', 'train', 'test'))
for lr, n in [(1.0, 200), (0.3, 200), (0.1, 200), (0.05, 200), (0.05, 800)]:
    m = Pipeline([('prep', preprocessor(scale=False)),
                  ('clf', GradientBoostingClassifier(learning_rate=lr,
                                                     n_estimators=n,
                                                     random_state=42))]).fit(X_tr,
```

```
    ↪ y_tr)
    print('%8s %8d %10.4f %10.4f'
          % (lr, n, roc_auc_score(y_tr, m.predict_proba(X_tr)[:, 1]),
             roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

lr	rounds	train	test
1.0	200	0.9998	0.7410
0.3	200	0.9797	0.7774
0.1	200	0.9157	0.7968
0.05	200	0.8857	0.8082
0.05	800	0.9547	0.7843

Watch Out

Boosting can and will overfit

At a learning rate of 1.0 the training AUC runs away toward 1.0 while the test score falls. Unlike a forest, more rounds is not free. The two parameters trade against each other: halve the learning rate and you need roughly twice the rounds, which costs time but usually generalises better. A small rate with early stopping is the standard recipe.

Early stopping

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
```

```
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import roc_auc_score

m = Pipeline([('prep', preprocessor(scale=False)),
              ('clf', HistGradientBoostingClassifier(
                  max_iter=1000, learning_rate=0.05,
                  early_stopping=True, validation_fraction=0.15,
                  n_iter_no_change=20, random_state=42))]).fit(X_tr, y_tr)
clf = m.named_steps['clf']
print('allowed up to 1000 rounds, stopped at %d' % clf.n_iter_)
print('test AUC %.4f' % roc_auc_score(y_te, m.predict_proba(X_te)[:, 1]))
```

```
allowed up to 1000 rounds, stopped at 80
test AUC 0.7924
```

It holds back 15 percent of the training data, watches the score on it, and stops when twenty rounds pass without improvement. You set a generous ceiling and let the model decide, which is more reliable than tuning the round count by hand.

	Random forest	**Gradient boosting**
Trees built	Independently, in parallel	Sequentially, each on the last errors
Individual trees	Deep, overfitting	Shallow, underfitting
More trees	Never hurts	Eventually overfits
Key parameters	`min_samples_leaf`, `max_features`	`learning_rate`, rounds, depth
Tuning effort	Low	Higher, and it pays
Typical winner	When you have no time to tune	When you do

Day 4 takeaway

Day 5 XGBoost and LightGBM

Regularised, histogram-based boosting that handles gaps natively

XGBoost and LightGBM are gradient boosting done properly, faster, regularised, and with native handling of the things that make real data awkward.

What they add

- **Regularisation on the trees themselves**, so leaf values cannot grow unchecked.
- **Histogram-based splitting**: bucket each feature into 256 bins instead of testing every value, which is dramatically faster.
- **Native missing-value handling**: each split learns which way a missing value should go.
- **Serious engineering**: multi-core, cache-aware, and able to work out of memory.

```
import numpy as np
import pandas as pd

```

```
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
import xgboost as xgb
import lightgbm as lgb
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import roc_auc_score
import time

models = {
    'sklearn hist': HistGradientBoostingClassifier(random_state=42),
    'xgboost     ': xgb.XGBClassifier(n_estimators=300, learning_rate=0.1,
                                      max_depth=4, random_state=42,
                                      eval_metric='logloss'),
    'lightgbm    ': lgb.LGBMClassifier(n_estimators=300, learning_rate=0.1,
                                       max_depth=4, random_state=42,
                                       verbose=-1),
}
for name, clf in models.items():
    t = time.perf_counter()
    m = Pipeline([('prep', preprocessor(scale=False)), ('clf', clf)]).fit(X_tr,
    ↪ y_tr)
    el = time.perf_counter() - t
    print('%s AUC %.4f   %.2fs'
          % (name, roc_auc_score(y_te, m.predict_proba(X_te)[:, 1]), el))
```

```
sklearn hist AUC 0.7797   4.45s
xgboost      AUC 0.7765   3.21s
lightgbm     AUC 0.7755   1.40s
```

Missing values need no imputation

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
import xgboost as xgb
from sklearn.metrics import roc_auc_score

# No imputer at all -- the raw numeric columns, gaps included.
Xn_tr = X_tr[NUM]
Xn_te = X_te[NUM]
print('missing values passed straight in: %d' % Xn_tr.isna().sum().sum())

m = xgb.XGBClassifier(n_estimators=300, max_depth=4, learning_rate=0.1,
                      random_state=42, eval_metric='logloss').fit(Xn_tr, y_tr)
print('AUC %.4f' % roc_auc_score(y_te, m.predict_proba(Xn_te)[:, 1]))
```

```
missing values passed straight in: 133
AUC 0.7237
```

Learned, not guessed

At each split the algorithm tries sending missing values left and sending them right, and keeps whichever reduces the loss more. That is strictly better than imputing a median, because it lets missingness carry its own meaning, the point week 2 made about informative gaps, handled automatically.

The parameters that matter

Parameter	Does	Try
`n_estimators`	Boosting rounds	300 to 2000 with early stopping
`learning_rate`	Step size per round	0.01 to 0.1
`max_depth`	Tree depth	3 to 8
`subsample`	Row fraction per tree	0.6 to 1.0
`colsample_bytree`	Feature fraction per tree	0.6 to 1.0
`reg_lambda`	L2 on leaf weights	1 to 10
`min_child_weight`	Minimum weight in a leaf	1 to 20

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
import xgboost as xgb
from sklearn.metrics import roc_auc_score

configs = [
    ('defaults      ', {}),
    ('shallow, slow ', dict(max_depth=3, learning_rate=0.03, n_estimators=600)),
    ('subsampled    ', dict(max_depth=4, learning_rate=0.05, n_estimators=500,
                            subsample=0.8, colsample_bytree=0.8)),
    ('regularised   ', dict(max_depth=5, learning_rate=0.05, n_estimators=500,
                            reg_lambda=10, min_child_weight=10)),
]
for name, kw in configs:
    clf = xgb.XGBClassifier(random_state=42, eval_metric='logloss', **kw)
    m = Pipeline([('prep', preprocessor(scale=False)), ('clf', clf)]).fit(X_tr,
    ↪ y_tr)
    tr = roc_auc_score(y_tr, m.predict_proba(X_tr)[:, 1])
    te = roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])
    print('%s train %.4f  test %.4f  gap %.4f' % (name, tr, te, tr - te))
```

```
defaults       train 0.9940  test 0.7574  gap 0.2366
shallow, slow  train 0.8913  test 0.8020  gap 0.0893
subsampled     train 0.9507  test 0.7804  gap 0.1703
regularised    train 0.9071  test 0.7905  gap 0.1166
```

Read the gap column. The default configuration overfits hardest; the shallow, slow and regularised

versions trade training score for a smaller gap. The best test score of the four, 0.8020, comes from the slowest and shallowest configuration, and tomorrow shows that even that does not settle the question.

Day 5 takeaway

Day 6 Feature Importance, and Why the Default Lies

Impurity importance ranking pure noise first, and the honest alternative

Every tree model offers a `feature_importances_` attribute. It is the most-quoted and least-trustworthy number in applied machine learning.

What the default importance measures

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import RandomForestClassifier
import pandas as pd

m = Pipeline([('prep', preprocessor(scale=False)),
              ('clf', RandomForestClassifier(n_estimators=300,
                                             random_state=42))]).fit(X_tr, y_tr)
```

```
names = m.named_steps['prep'].get_feature_names_out()
imp = pd.Series(m.named_steps['clf'].feature_importances_, index=names)
print(imp.sort_values(ascending=False).head(8).round(4).to_string())
```

```
num__monthly_charges                  0.3216
num__tenure_months                    0.3167
num__support_calls                    0.1009
cat__contract_Month-To-Month          0.0701
cat__contract_Two Year                0.0359
cat__payment_method_Electronic check  0.0213
cat__contract_One Year                0.0185
cat__internet_service_Fibre optic     0.0180
```

The bias, demonstrated

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier

rng = np.random.default_rng(0)
n = 2000
y = rng.integers(0, 2, n)

X = pd.DataFrame({
    'real_signal': y + rng.normal(0, 1.2, n),    # genuinely predictive
    'binary_noise': rng.integers(0, 2, n),       # pure noise, 2 values
    'id_noise': rng.permutation(n),              # pure noise, 2000 values
})

m = RandomForestClassifier(n_estimators=300, random_state=42).fit(X, y)
print(pd.Series(m.feature_importances_, index=X.columns).round(4).to_string())
```

```
real_signal     0.5739
binary_noise    0.0035
id_noise        0.4225
```

Watch Out

It rewards high-cardinality columns for being noisy

`id_noise` is a random permutation. It contains no information whatsoever, and impurity importance ranks it comparably to the column that genuinely predicts the target, far above the binary noise column. A feature with thousands of distinct values offers thousands of candidate split points, so by chance some of them separate the training rows well. The importance measures opportunity to overfit, not usefulness.

Permutation importance is the honest one

```
import numpy as np
import pandas as pd
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.inspection import permutation_importance
from sklearn.model_selection import train_test_split

rng = np.random.default_rng(0)
n = 2000
y = rng.integers(0, 2, n)
X = pd.DataFrame({
    'real_signal': y + rng.normal(0, 1.2, n),
    'binary_noise': rng.integers(0, 2, n),
    'id_noise': rng.permutation(n),
})
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.3, random_state=0)
m = RandomForestClassifier(n_estimators=300, random_state=42).fit(X_tr, y_tr)

r = permutation_importance(m, X_te, y_te, n_repeats=20, random_state=0,
                           scoring='roc_auc')
print(pd.DataFrame({'importance': r.importances_mean,
                    'std': r.importances_std},
                   index=X.columns).round(4).to_string())
```

```
             importance     std
real_signal       0.1876  0.0209
binary_noise     -0.0009  0.0128
id_noise          0.0107  0.0107
```

Now both noise columns sit within one standard deviation of zero, where they belong, while the real signal is an order of magnitude above them. Note that `id_noise` has not vanished entirely, the model did latch onto it slightly, and permutation importance correctly reports that as small rather than pretending it is absent. Same model, same data, a different and better question.

On the churn model

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
```

```
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import RandomForestClassifier
from sklearn.inspection import permutation_importance
import pandas as pd

m = Pipeline([('prep', preprocessor(scale=False)),
              ('clf', RandomForestClassifier(n_estimators=300,
                                             random_state=42))]).fit(X_tr, y_tr)
r = permutation_importance(m, X_te, y_te, n_repeats=15, random_state=0,
                           scoring='roc_auc')
print(pd.Series(r.importances_mean, index=X_te.columns)
      .sort_values(ascending=False).round(4).to_string())
```

```
contract            0.1478
tenure_months       0.0667
monthly_charges     0.0171
support_calls       0.0127
payment_method      0.0111
internet_service    0.0102
has_dependents      0.0082
```

Permuting whole original columns, before encoding, which is what you actually want to know: is `contract` useful, not is `contract_Two Year` useful.

Watch Out

Correlated features share the blame

Shuffle `tenure_months` while `total_charges` still carries most of the same information and the score barely moves, so both look unimportant. Permutation importance answers "how much does *this model* rely on this column", never "how much does this column matter in the world". Group correlated columns and permute them together if you need the second answer.

Day 6 takeaway

Day 7 Why the Simple Model Won

The bake-off, and matching an algorithm to the shape of the truth

Week 1 set you an assignment: swap logistic regression for a random forest and explain why the forest scored worse. Here is the answer, and it matters more than any algorithm this week.

The bake-off

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(scale=True):
    num_steps = [('i', SimpleImputer(strategy='median'))]
    if scale:
        num_steps.append(('s', StandardScaler()))
    return ColumnTransformer([
        ('num', Pipeline(num_steps), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, HistGradientBoostingClassifier
import xgboost as xgb
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
candidates = {
    'baseline     ': DummyClassifier(strategy='most_frequent'),
    'logistic     ': LogisticRegression(max_iter=1000, random_state=42),
    'tree leaf=50 ': DecisionTreeClassifier(min_samples_leaf=50, random_state=42),
    'forest       ': RandomForestClassifier(n_estimators=300, min_samples_leaf=5,
                                            random_state=42),
    'hist boosting': HistGradientBoostingClassifier(random_state=42),
    'xgboost      ': xgb.XGBClassifier(n_estimators=400, max_depth=3,
                                       learning_rate=0.05, subsample=0.8,
                                       random_state=42, eval_metric='logloss'),
}
for name, clf in candidates.items():
    s = cross_val_score(Pipeline([('prep', preprocessor()), ('clf', clf)]),
                        X_tr, y_tr, cv=cv, scoring='roc_auc')
```

```
    print('%s %.4f  +/- %.4f' % (name, s.mean(), s.std()))
```

```
baseline      0.5000  +/- 0.0000
logistic      0.8208  +/- 0.0137
tree leaf=50  0.7922  +/- 0.0078
forest        0.8203  +/- 0.0101
hist boosting 0.7973  +/- 0.0088
xgboost       0.8133  +/- 0.0083
```

A week of ensembles, and nothing beats the straight line

Logistic regression scores 0.8208. The best ensemble, a random forest with `min_samples_leaf=5`: scores 0.8203, which is a tie: the difference is a twentieth of the fold-to-fold spread. XGBoost is a point behind, and boosting with default settings is two points behind. Nothing here is a bug, and nothing you did is wrong. The honest summary is that a week of increasingly sophisticated machinery has managed to *draw* with the simplest model in the course.

Why

Look at how the data was generated, back in week 1:

```
# From make_dataset.py, week 1 day 3
#
#   logit = (-2.8
#            + 1.55 * (contract == 'Month-to-month')
#            - 0.85 * (contract == 'Two year')
#            - 0.055 * tenure
#            + 0.021 * monthly
#            + 0.30 * support
#            + 0.42 * (payment == 'Electronic check')
#            - 0.25 * (dependents == 'Yes'))
#   churned = sigmoid(logit) > uniform()

print('The truth is a weighted sum, passed through a sigmoid.')
print('That is exactly, precisely, the functional form of')
print('logistic regression -- and nothing else in this course.')
```

```
The truth is a weighted sum, passed through a sigmoid.
That is exactly, precisely, the functional form of
logistic regression -- and nothing else in this course.
```

Logistic regression is not approximating the truth here. It *is* the truth, with the right coefficients to be found. A tree can only approximate a smooth linear relationship with a staircase of splits, and no amount of boosting turns a staircase into a straight line.

Where trees win instead

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
```

```
from sklearn.model_selection import cross_val_score

rng = np.random.default_rng(0)
n = 3000
a = rng.uniform(-3, 3, n)
b = rng.uniform(-3, 3, n)

# An interaction with a threshold -- no linear model can express this.
y = (((a > 0) & (b > 0)) | ((a < -1.5) & (b < -1.5))).astype(int)
y = np.where(rng.uniform(size=n) < 0.05, 1 - y, y)      # 5% label noise
X = np.column_stack([a, b])

for name, clf in [('logistic', LogisticRegression(max_iter=1000)),
                  ('boosting', HistGradientBoostingClassifier(random_state=42))]:
    s = cross_val_score(make_pipeline(StandardScaler(), clf), X, y,
                        cv=5, scoring='roc_auc')
    print('%-9s AUC %.4f' % (name, s.mean()))
```

```
logistic  AUC 0.7184
boosting  AUC 0.9352
```

Same two features, a rule built from thresholds and interactions rather than a weighted sum, and the result reverses completely. Neither algorithm is better. They match different shapes of truth.

Prefer a linear model when	Prefer trees when
Effects are additive and smooth	Thresholds and interactions dominate
You need interpretable coefficients	You need raw predictive power
Data is wide and rows are few	Rows are plentiful
You must extrapolate beyond the training range	All prediction is inside the training range
Probabilities must be well calibrated	Ranking is what matters

Watch Out

Trees cannot extrapolate at all

A tree's prediction for any input beyond the training range is whatever the nearest leaf says, forever. Give a forest trained on tenures of 1 to 72 a customer with 200 months and it predicts as though they had 72. A linear model would continue the trend. Which behaviour is right depends entirely on your problem, but you must know which one you are getting.

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor

x = np.linspace(0, 10, 200).reshape(-1, 1)
y = 3 * x.ravel() + 5

lin = LinearRegression().fit(x, y)
forest = RandomForestRegressor(n_estimators=100, random_state=0).fit(x, y)

print('%10s %14s %14s %12s' % ('x', 'truth', 'linear', 'forest'))
```

```
for v in [5.0, 10.0, 15.0, 50.0]:
    print('%10.1f %14.2f %14.2f %12.2f'
          % (v, 3 * v + 5, lin.predict([[v]])[0], forest.predict([[v]])[0]))
```

```
         x          truth         linear       forest
       5.0          20.00          20.00        19.97
      10.0          35.00          35.00        34.92
      15.0          50.00          50.00        34.92
      50.0         155.00         155.00        34.92
```

Your assignment

Add an interaction the linear model cannot see, for example a flag for `month-to-month AND support_calls > 2`: to the churn features, and rerun the bake-off. Does the gap between logistic regression and boosting narrow? Then engineer that same interaction as an explicit column for the linear model and see who wins. That is week 8.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. A decision tree chooses each split by:

    a) Greedily picking the split that most reduces impurity, without looking ahead
    b) Solving for the global optimum
    c) Testing every possible tree
    d) Random selection

2. The main weakness of a single deep tree is that it:

    a) Cannot handle categories
    b) Is unstable, a small change in the data can produce a very different tree
    c) Is slow to fit
    d) Requires scaling

3. A random forest reduces variance by:

    a) Boosting the errors
    b) Pruning harder
    c) Averaging many decorrelated trees, each on a bootstrap sample and a random subset of features
    d) Using fewer features

4. Boosting differs from bagging in that each new model is fitted to:

    a) A different feature set
    b) The same data with a new seed

c) A random sample

d) The errors left by the models before it

5. Impurity-based feature importance is misleading because it:

   a) Favours high-cardinality and continuous features, which offer more places to split. It can rank pure noise first

   b) Ignores interactions

   c) Requires scaling

   d) Is computed on test data

6. The honest alternative to impurity importance is:

   a) Coefficient magnitude

   b) Permutation importance, measured on held-out data

   c) Tree depth

   d) Split count

7. What does out-of-bag scoring give you?

   a) Calibrated probabilities

   b) A faster fit

   c) A validation estimate for free, from the rows each tree did not see

   d) A better model

8. Logistic regression matched the tree ensembles on the course dataset because:

   a) The trees were badly tuned

   b) Trees need more features

   c) The dataset is small

   d) The underlying relationship really is close to linear in log-odds, so there is no extra structure for trees to find

Answers:1a, 2b, 3c, 4d, 5a, 6b, 7c, 8d. Anything you got wrong points at the day to re-read, not at a gap in ability.

Chapter 7

Model Evaluation and Validation

Week 7

OBJECTIVES

Measuring a model honestly, the folds, the leaks, the imbalance, the calibration, and the segments the average score hides.

The most valuable week in the course, because a wrong measurement costs more than a mediocre model. It covers which cross-validation scheme a problem demands, the leaks cross-validation cannot detect, what actually helps with imbalance, why a well-ranking model can still be wrong about probabilities, and how to find who your model fails.

Day 1 Cross-Validation Done Properly

Stratified, grouped and time-ordered folds, and the nested version

You have used `cross_val_score` since week 4 without asking how the folds are made. That choice is where most silently broken evaluations come from.

Why one split is not enough

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
```

```
    ])
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
import numpy as np

X, y = df[NUM + CAT], df[TARGET]
scores = []
for seed in range(12):
    a, b, c, d = train_test_split(X, y, test_size=0.25, stratify=y,
                                  random_state=seed)
    m = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000))]).fit(a, c)
    scores.append(roc_auc_score(d, m.predict_proba(b)[:, 1]))

scores = np.array(scores)
print('twelve different random splits of the same data:')
print('  min %.4f   max %.4f   spread %.4f'
      % (scores.min(), scores.max(), scores.max() - scores.min()))
print('  mean %.4f  std %.4f' % (scores.mean(), scores.std()))
```

```
twelve different random splits of the same data:
  min 0.8030   max 0.8364   spread 0.0334
  mean 0.8207  std 0.0108
```

Watch Out

Two points of AUC, from nothing but the seed

Same data, same model, same code, only the split changed. If you compare two models on one split each and they differ by less than this spread, you have measured the seed, not the models. Reporting a single split's score to four decimal places implies a precision that does not exist.

The splitters, and when each is required

Splitter	Use when	Failure if you use KFold instead
`KFold`	Rows are independent and balanced	-
`StratifiedKFold`	Classification, any imbalance	Folds get different class rates; scores wobble
`GroupKFold`	Rows cluster by entity	The same entity lands in train and test, leakage
`TimeSeriesSplit`	Rows are ordered in time	The model trains on the future
`RepeatedStratifiedKFold`	You need a tighter estimate	-

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')
```

```

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.model_selection import KFold, StratifiedKFold
import numpy as np

y = df[TARGET].to_numpy()
print('overall churn rate %.4f\n' % y.mean())

for name, splitter in [('KFold          ', KFold(5, shuffle=True, random_state=0)),
                       ('StratifiedKFold', StratifiedKFold(5, shuffle=True,
                                                           random_state=0))]:
    rates = [y[test].mean() for _, test in splitter.split(np.zeros(len(y)), y)]
    print('%s fold churn rates %s  spread %.4f'
          % (name, np.round(rates, 4), max(rates) - min(rates)))
```

```
overall churn rate 0.2680

KFold          fold churn rates [0.265  0.2733 0.25   0.2933 0.2583]  spread 0.0433
StratifiedKFold fold churn rates [0.2683 0.2683 0.2683 0.2683 0.2667]  spread 0.0017
```

Grouped data, where the trap is worst

```
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_val_score, KFold, GroupKFold
from sklearn.ensemble import RandomForestClassifier

rng = np.random.default_rng(0)
n_customers, per_customer = 200, 5

# Five readings per customer. The label belongs to the customer, not the
# row, and each reading is that customer's traits plus a little noise --
# which is exactly what repeated measurement of one entity looks like.
customer = np.repeat(np.arange(n_customers), per_customer)
label = np.repeat(rng.integers(0, 2, n_customers), per_customer)
traits = np.repeat(rng.normal(size=(n_customers, 4)), per_customer, axis=0)
X = traits + rng.normal(0, 0.02, traits.shape)

clf = RandomForestClassifier(n_estimators=100, random_state=0)
naive = cross_val_score(clf, X, label, cv=KFold(5, shuffle=True, random_state=0))
grouped = cross_val_score(clf, X, label, cv=GroupKFold(5), groups=customer)

print('KFold      accuracy %.4f   <- the same customer is in both halves'
      % naive.mean())
print('GroupKFold accuracy %.4f   <- honest' % grouped.mean())
```

```
KFold      accuracy 0.9970   <- the same customer is in both halves
GroupKFold accuracy 0.5880   <- honest
```

Watch Out

Ninety-nine percent, and the model has learned nothing

Forty-one points of pure illusion. Repeated measurements of one patient, several sessions from one user, multiple photographs of one object, split at random and near-duplicates of every test row sit in the training set. The model does not learn the pattern; it memorises which customer each reading came from and looks up their label. `GroupKFold` keeps a customer's rows together and reveals the truth: 0.59, barely above chance.
This is the most expensive mistake in this course, because the model looks superb right up until it meets someone new. Whenever rows cluster by an entity, pass `groups`.

Nested cross-validation

If you tune hyperparameters by cross-validation and then report that same cross-validated score, the score is optimistic: you chose the settings that happened to suit those folds.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import (GridSearchCV, cross_val_score,
                                     StratifiedKFold)
import numpy as np

X, y = df[NUM + CAT], df[TARGET]
grid = {'clf__min_samples_leaf': [1, 5, 20, 50]}
pipe = Pipeline([('prep', preprocessor()),
                 ('clf', RandomForestClassifier(n_estimators=150,
                                                random_state=42))])

inner = StratifiedKFold(4, shuffle=True, random_state=1)
outer = StratifiedKFold(4, shuffle=True, random_state=2)

```

```
search = GridSearchCV(pipe, grid, cv=inner, scoring='roc_auc')
search.fit(X, y)
print('best inner score (optimistic) %.4f' % search.best_score_)

nested = cross_val_score(search, X, y, cv=outer, scoring='roc_auc')
print('nested score     (honest)     %.4f  +/- %.4f'
      % (nested.mean(), nested.std()))
```

```
best inner score (optimistic) 0.8216
nested score     (honest)     0.8176  +/- 0.0122
```

The nested figure re-runs the entire search inside each outer fold, so the settings are never chosen using the rows they are scored on. It costs inner times outer fits, which is why people skip it, but when you need to report a number you will be held to, this is the number.

Day 1 takeaway

Day 2 The Leaks Cross-Validation Cannot See

Scaling, selection and duplicates outside the pipeline, and noise scoring 0.7

Cross-validation protects you from one thing only: scoring on rows you trained on. It cannot see a leak that happened before the folds were made.

Scaling before the split

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score, StratifiedKFold
import numpy as np

X, y = df[NUM].to_numpy(), df[TARGET].to_numpy()
X = SimpleImputer(strategy='median').fit_transform(X)
cv = StratifiedKFold(5, shuffle=True, random_state=0)

# Wrong: the scaler sees every row, including the ones it will be tested on.
X_leaked = StandardScaler().fit_transform(X)
leaked = cross_val_score(LogisticRegression(max_iter=1000), X_leaked, y,
```

```
                          cv=cv, scoring='roc_auc')

# Right: the scaler is refitted inside every fold.
clean = cross_val_score(make_pipeline(StandardScaler(),
                                      LogisticRegression(max_iter=1000)),
                        X, y, cv=cv, scoring='roc_auc')

print('scaled before splitting %.4f' % leaked.mean())
print('scaled inside the fold  %.4f' % clean.mean())
print('difference              %+.4f' % (leaked.mean() - clean.mean()))
```

```
scaled before splitting 0.7721
scaled inside the fold  0.7721
difference              +0.0000
```

Watch Out

Zero difference is what makes this dangerous

The two numbers are identical to four decimal places. With three thousand rows, a mean and a standard deviation computed on 80 percent of them are indistinguishable from the same figures computed on all of them, so this particular leak costs nothing measurable. That is precisely the problem. The *same mistake* with target encoding, an aggregate feature, or feature selection produces gaps of ten points or more, and the next snippet turns pure noise into an apparent 0.80. Nothing in the score tells you which case you are in, so the rule cannot be "check whether it matters". It has to be absolute: every step that learns from data goes inside the pipeline.

Feature selection before the split, which is catastrophic

```
import numpy as np
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score, StratifiedKFold

rng = np.random.default_rng(0)
n, p = 200, 5000
X = rng.normal(size=(n, p))        # pure noise
y = rng.integers(0, 2, n)          # unrelated labels
cv = StratifiedKFold(5, shuffle=True, random_state=0)

# Wrong: pick the 20 best columns using ALL the labels, then cross-validate.
best = SelectKBest(f_classif, k=20).fit_transform(X, y)
leaked = cross_val_score(LogisticRegression(max_iter=1000), best, y, cv=cv)

# Right: selection happens inside each fold.
clean = cross_val_score(make_pipeline(SelectKBest(f_classif, k=20),
                                      LogisticRegression(max_iter=1000)),
                        X, y, cv=cv)

```

```
print('there is no signal in this data at all -- 0.5 is the truth')
print('  selection outside the folds %.4f' % leaked.mean())
print('  selection inside the folds  %.4f' % clean.mean())
```

```
there is no signal in this data at all -- 0.5 is the truth
  selection outside the folds 0.7950
  selection inside the folds  0.5450
```

Random noise, random labels, no relationship whatsoever, and selecting features before cross-validating reports accuracy far above chance. Out of five thousand noise columns, twenty will correlate with the labels by luck, and choosing them using every label is how the luck gets in. This is a documented cause of retracted findings in published research.

Duplicate rows

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
import pandas as pd

raw = pd.read_csv('customers.csv')            # 138 duplicated rows
deduped = raw.drop_duplicates()
cv = StratifiedKFold(5, shuffle=True, random_state=0)
pipe = make_pipeline(SimpleImputer(strategy='median'),
                     RandomForestClassifier(n_estimators=200, random_state=42))

for name, d in [('with duplicates   ', raw), ('deduplicated      ', deduped)]:
    s = cross_val_score(pipe, d[NUM], d[TARGET], cv=cv, scoring='roc_auc')
    print('%s AUC %.4f  (%d rows)' % (name, s.mean(), len(d)))
```

```
with duplicates    AUC 0.7503  (3138 rows)
deduplicated       AUC 0.7212  (3000 rows)
```

Only 138 duplicates in three thousand rows, so the inflation is modest. Scale that to an export that ran twice and the effect is severe. De-duplicate first, always.

A checklist you can actually run

1. Is every fitted transformation inside the `Pipeline`?
2. Were duplicates removed before splitting?

3. Do any rows share an entity that could straddle the split?
4. Are the rows time-ordered, and does the split respect that?
5. Was any feature computed using an aggregate over the whole dataset?
6. For every feature: would I have this value at prediction time?
7. Is the score suspiciously good?

Suspicion is a diagnostic

The last item is not a joke. Experienced practitioners treat an unexpectedly excellent score as a bug report. If a model jumps from 0.82 to 0.97 because you added one feature, the overwhelmingly likely explanation is that the feature leaks.

> **Day 2 takeaway**

Day 3 Imbalanced Classes

Why accuracy rises as the problem gets harder, and what actually helps

Churn is 27 percent, which is mild. Fraud is 0.1 percent, and at that level everything you have learned about accuracy stops working.

The problem, at increasing severity

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import (accuracy_score, roc_auc_score,
                             average_precision_score, recall_score)

print('%10s %10s %10s %10s %10s'
      % ('positives', 'accuracy', 'recall', 'ROC AUC', 'avg prec'))
for weight in [0.5, 0.1, 0.02, 0.005]:
    X, y = make_classification(n_samples=20000, n_features=12, n_informative=5,
                               weights=[1 - weight], flip_y=0.01, random_state=0)
    a, b, c, d = train_test_split(X, y, test_size=0.3, stratify=y, random_state=0)
    m = LogisticRegression(max_iter=2000).fit(a, c)
    pred, proba = m.predict(b), m.predict_proba(b)[:, 1]
    print('%10.3f %10.4f %10.4f %10.4f %10.4f'
          % (d.mean(), accuracy_score(d, pred), recall_score(d, pred),
             roc_auc_score(d, proba), average_precision_score(d, proba)))
```

```
 positives   accuracy     recall    ROC AUC   avg prec
     0.500     0.9365     0.9494     0.9782     0.9724
     0.104     0.9623     0.7388     0.9520     0.8674
     0.025     0.9835     0.3791     0.8491     0.6144
     0.011     0.9915     0.2031     0.6895     0.3925
```

Accuracy climbs toward 1.0 as the problem gets harder, because guessing the majority class gets easier. Recall collapses. Accuracy is not a weak metric here. It is an actively misleading one.

Class weights, which cost nothing

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (recall_score, precision_score,
                             average_precision_score, roc_auc_score)

for weight in [None, 'balanced']:
    m = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000, class_weight=weight,
                                             random_state=42))]).fit(X_tr, y_tr)
    pred, proba = m.predict(X_te), m.predict_proba(X_te)[:, 1]
    print('class_weight=%-9s recall %.4f  precision %.4f  AUC %.4f  AP %.4f'
          % (str(weight), recall_score(y_te, pred),
             precision_score(y_te, pred), roc_auc_score(y_te, proba),
             average_precision_score(y_te, proba)))
```

```
class_weight=None      recall 0.4328  precision 0.6304  AUC 0.8174  AP 0.5993
class_weight=balanced  recall 0.7960  precision 0.4923  AUC 0.8172  AP 0.5973
```

class_weight='balanced' is mostly a threshold move

Recall jumps and precision falls, but ROC AUC and average precision barely change, because the *ranking* of customers by risk is nearly the same. Weighting mostly shifts where the 0.5 cut lands. That is worth knowing: if you were going to tune the threshold anyway, as week 5 did, class weights add little. They matter more when the algorithm's loss is genuinely dominated by the majority class.

Resampling

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from imblearn.over_sampling import SMOTE, RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline as ImbPipeline
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import average_precision_score, roc_auc_score

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT)])

samplers = {'none         ': None,
            'oversample   ': RandomOverSampler(random_state=0),
            'undersample  ': RandomUnderSampler(random_state=0),
            'SMOTE        ': SMOTE(random_state=0)}

for name, sampler in samplers.items():
    steps = [('prep', prep)]
    if sampler is not None:
        steps.append(('sample', sampler))
    steps.append(('clf', LogisticRegression(max_iter=1000, random_state=42)))
    m = ImbPipeline(steps).fit(X_tr, y_tr)
    proba = m.predict_proba(X_te)[:, 1]
```

```
    print('%s AUC %.4f   avg precision %.4f'
          % (name, roc_auc_score(y_te, proba),
             average_precision_score(y_te, proba)))
```

```
none          AUC 0.8174   avg precision 0.5993
oversample    AUC 0.8143   avg precision 0.5923
undersample   AUC 0.8110   avg precision 0.5856
SMOTE         AUC 0.8160   avg precision 0.5951
```

Watch Out

Resample inside the pipeline, and never the test set

`imblearn`'s `Pipeline` exists precisely because scikit-learn's will not resample. It applies the sampler during `fit` and skips it during `predict`, which is the only correct behaviour. Oversampling before the split copies minority rows into both halves. The model is then tested on rows it trained on, and the score is fiction.

Notice how little any of it achieved on this problem. Resampling is oversold; it changes the ranking barely at all, and at 27 percent minority there is not much to fix. Reach for it when the minority class is under a few percent, and measure rather than assume.

What to actually do about imbalance

1. **Use the right metric**: average precision, not accuracy. This alone fixes most of the confusion.
2. **Tune the threshold** against the cost of each error, as in week 5. Free, and usually the largest gain.
3. **Try class weights**, which cost one argument.
4. **Then consider resampling**, measuring whether it helped.
5. **Collect more minority examples** if you possibly can. Nothing else comes close.

Day 3 takeaway

Day 4 Calibration

Ranking well and still being wrong about the numbers

A model that ranks customers perfectly can still be badly wrong about the numbers. If anyone will multiply your probability by a pound value, ranking is not enough.

Measure it

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
```

```
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.calibration import calibration_curve
from sklearn.metrics import brier_score_loss, roc_auc_score
import numpy as np

models = {
    'logistic': LogisticRegression(max_iter=1000, random_state=42),
    'forest  ': RandomForestClassifier(n_estimators=300, random_state=42),
    'naive-B ': GaussianNB(),
}
for name, clf in models.items():
    m = Pipeline([('prep', preprocessor()), ('clf', clf)]).fit(X_tr, y_tr)
    proba = m.predict_proba(X_te)[:, 1]
    print('%s  AUC %.4f   Brier %.4f'
          % (name, roc_auc_score(y_te, proba),
             brier_score_loss(y_te, proba)))
```

```
logistic  AUC 0.8174   Brier 0.1475
forest    AUC 0.7772   Brier 0.1674
naive-B   AUC 0.8044   Brier 0.2438
```

The Brier score is the mean squared error of the probabilities. Lower is better, and unlike AUC it punishes a model for being confidently wrong about the level rather than just the order.

The reliability table

```
import numpy as np
import pandas as pd

```

```
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.calibration import calibration_curve
import numpy as np

for name, clf in [('logistic', LogisticRegression(max_iter=1000, random_state=42)),
                  ('naive-B ', GaussianNB())]:
    m = Pipeline([('prep', preprocessor()), ('clf', clf)]).fit(X_tr, y_tr)
    proba = m.predict_proba(X_te)[:, 1]
    true_rate, predicted = calibration_curve(y_te, proba, n_bins=6,
                                             strategy='quantile')
    print('%s   predicted → actual' % name)
    for pr, ac in zip(predicted, true_rate):
        print('           %.3f    → %.3f' % (pr, ac))
    print()
```

```
logistic   predicted → actual
           0.010    → 0.008
           0.058    → 0.072
           0.162    → 0.152
           0.285    → 0.304
           0.433    → 0.456
           0.628    → 0.616

naive-B    predicted → actual
           0.000    → 0.024
           0.003    → 0.064
           0.123    → 0.160
           0.613    → 0.312
           0.955    → 0.448
           0.989    → 0.600
```

Logistic regression tracks the diagonal closely, which is expected: it is fitted by minimising log loss, and log loss is minimised by telling the truth about probabilities. Naive Bayes, which multiplies correlated probabilities as though independent, is pushed toward the extremes.

Repairing it

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.naive_bayes import GaussianNB
from sklearn.calibration import CalibratedClassifierCV
from sklearn.metrics import brier_score_loss, roc_auc_score

base = Pipeline([('prep', preprocessor()), ('clf', GaussianNB())])

fitted = base.fit(X_tr, y_tr)
raw = fitted.predict_proba(X_te)[:, 1]
print('uncalibrated   Brier %.4f   AUC %.4f'
      % (brier_score_loss(y_te, raw), roc_auc_score(y_te, raw)))

for method in ['sigmoid', 'isotonic']:
    cal = CalibratedClassifierCV(base, method=method, cv=5).fit(X_tr, y_tr)
    proba = cal.predict_proba(X_te)[:, 1]
    print('%-14s Brier %.4f   AUC %.4f'
          % (method, brier_score_loss(y_te, proba), roc_auc_score(y_te, proba)))
```

```
uncalibrated   Brier 0.2438   AUC 0.8044
sigmoid        Brier 0.1559   AUC 0.8060
isotonic       Brier 0.1525   AUC 0.8043
```

Method	Fits	Use when
`sigmoid` (Platt)	A logistic curve, two parameters	Small data; the distortion is a smooth S-shape
`isotonic`	Any non-decreasing step function	Plenty of data; more flexible, can overfit

Calibration barely changes AUC

Both methods apply a monotone transformation, and a monotone transformation cannot change the ordering. So AUC stays put while the Brier score improves substantially. That is the clearest possible demonstration that ranking and calibration are separate properties.

When it matters

- **Expected value calculations**: probability times customer value only means something if the probability is real.
- **Combining models**: averaging miscalibrated probabilities is meaningless.
- **Anything shown to a human**: a clinician told 80 percent will act on 80 percent.
- **Thresholds set from cost**: week 5's calculation assumed the probabilities meant something.

If you only ever sort customers and contact the top thousand, calibration is irrelevant. Know which situation you are in.

> **Day 4 takeaway**

Day 5 Learning and Validation Curves

Diagnosing whether you need more data, more features or less model

Your model underperforms. Do you collect more data, add features, or pick a different algorithm? Guessing wastes weeks. Two curves answer it.

Learning curves: is more data the answer?

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

```

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import learning_curve, StratifiedKFold
import numpy as np

X, y = df[NUM + CAT], df[TARGET]
cv = StratifiedKFold(5, shuffle=True, random_state=0)

for name, clf in [('logistic (well matched)',
                   LogisticRegression(max_iter=1000)),
                  ('deep tree (high variance)',
                   DecisionTreeClassifier(random_state=0))]:
    sizes, tr, va = learning_curve(
        Pipeline([('prep', preprocessor()), ('clf', clf)]), X, y, cv=cv,
        scoring='roc_auc', train_sizes=np.linspace(0.1, 1.0, 5))
    print(name)
    print('%10s %10s %10s %8s' % ('rows', 'train', 'val', 'gap'))
    for n, a, b in zip(sizes, tr.mean(axis=1), va.mean(axis=1)):
        print('%10d %10.4f %10.4f %8.4f' % (n, a, b, a - b))
    print()
```

```
logistic (well matched)
      rows      train        val      gap
       240     0.8044     0.8056  -0.0012
       780     0.8141     0.8180  -0.0039
      1320     0.8245     0.8180   0.0065
      1860     0.8236     0.8198   0.0038
      2400     0.8267     0.8220   0.0047

deep tree (high variance)
      rows      train        val      gap
       240     1.0000     0.5973   0.4027
       780     1.0000     0.6119   0.3881
      1320     1.0000     0.6207   0.3793
      1860     1.0000     0.6270   0.3730
      2400     1.0000     0.6344   0.3656
```

Pattern	Diagnosis	What to do
Both curves low, converged	High bias, underfitting	Better features, a more flexible model
Wide gap, validation still rising	High variance	More data, more regularisation, fewer features
Both high, converged	You are done	Ship it
Validation falling as data grows	Something is wrong	Check for leakage or a distribution shift

The logistic curves converge with almost no gap: it has extracted what this feature set contains, and more rows will not help. The deep tree keeps a large gap at every size, which is the signature of variance. There, more data genuinely would.

Validation curves: is this parameter set right?

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import validation_curve, StratifiedKFold
import numpy as np

X, y = df[NUM + CAT], df[TARGET]
values = [1, 2, 5, 10, 25, 50, 100, 200]
tr, va = validation_curve(
    Pipeline([('prep', preprocessor()),
              ('clf', RandomForestClassifier(n_estimators=150, random_state=0))]),
    X, y, param_name='clf__min_samples_leaf', param_range=values,
    cv=StratifiedKFold(4, shuffle=True, random_state=0), scoring='roc_auc')

print('%18s %10s %10s %8s' % ('min_samples_leaf', 'train', 'val', 'gap'))
for v, a, b in zip(values, tr.mean(axis=1), va.mean(axis=1)):
    print('%18d %10.4f %10.4f %8.4f' % (v, a, b, a - b))
```

```
 min_samples_leaf      train        val      gap
                1     1.0000     0.7837   0.2163
                2     0.9834     0.7984   0.1851
                5     0.9198     0.8117   0.1081
               10     0.8810     0.8151   0.0659
               25     0.8492     0.8187   0.0305
               50     0.8357     0.8179   0.0178
              100     0.8262     0.8158   0.0104
              200     0.8214     0.8151   0.0063
```

Read it left to right: at leaf size 1 the training score is near perfect and the gap enormous. As the leaves grow the gap closes and validation improves, up to a point, then both fall as the model becomes too constrained. The best value is where validation peaks, and the shape tells you whether you are on the overfitting or underfitting side of it.

Watch Out

A peak at the edge of your range means the range is wrong

If the best value is the largest or smallest you tried, you have not found the optimum. You have found the edge of your search. Extend the range and look again. The same rule applied to ridge's alpha in week 4, and it applies to every hyperparameter search in week 14.

Putting both together

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate, StratifiedKFold

cv = StratifiedKFold(5, shuffle=True, random_state=0)
for name, clf in [('logistic', LogisticRegression(max_iter=1000, random_state=0)),
                  ('boosting', HistGradientBoostingClassifier(random_state=0))]:
    r = cross_validate(Pipeline([('prep', preprocessor()), ('clf', clf)]),
                       X_tr, y_tr, cv=cv, scoring='roc_auc',
```

```
                        return_train_score=True)
    print('%-9s train %.4f   val %.4f   gap %+.4f'
          % (name, r['train_score'].mean(), r['test_score'].mean(),
             r['train_score'].mean() - r['test_score'].mean()))
```

```
logistic  train 0.8289   val 0.8238   gap +0.0052
boosting  train 0.9845   val 0.7966   gap +0.1879
```

`return_train_score=True` is the cheapest diagnostic in scikit-learn and almost nobody uses it. One extra argument turns a score into a diagnosis: boosting's gap is many times logistic regression's, which says it is fitting noise this dataset does not reward.

Day 5 takeaway

Day 6 Error Analysis and Fairness

Who the model fails, and why the headline score hides it

An aggregate score tells you how the model does on average. It never tells you who it fails, and that is what determines whether you can deploy it.

Score by segment

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, average_precision_score
import pandas as pd

m = Pipeline([('prep', preprocessor()),
              ('clf', LogisticRegression(max_iter=1000,
                                         random_state=42))]).fit(X_tr, y_tr)
proba = m.predict_proba(X_te)[:, 1]

seg = df.loc[X_te.index].copy()
seg['proba'] = proba
seg['truth'] = y_te.to_numpy()

rows = []
for name, group in seg.groupby('contract'):
    if group['truth'].nunique() < 2:
        continue
    rows.append({'segment': name, 'n': len(group),
                 'churn_rate': group['truth'].mean(),
                 'auc': roc_auc_score(group['truth'], group['proba']),
                 'avg_prec': average_precision_score(group['truth'],
                                                     group['proba'])})
print(pd.DataFrame(rows).round(4).to_string(index=False))
print('\noverall AUC %.4f' % roc_auc_score(y_te, proba))
```

```
       segment   n  churn_rate    auc  avg_prec
Month-To-Month 416      0.4135 0.7072    0.6279
      One Year 184      0.1141 0.8113    0.4052
      Two Year 150      0.0533 0.8688    0.3955

overall AUC 0.8174
```

Watch Out

Segment AUCs can all be lower than the overall AUC

This looks impossible and is not. Much of the overall ranking skill comes from separating month-to-month customers from two-year ones. *Within* a contract type that advantage is gone, and only the weaker signals remain. If your product treats each segment separately, the within-segment number is the one that matters, and it is worse than the headline.

Look at what it got most wrong

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
```

```
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
import pandas as pd
import numpy as np

m = Pipeline([('prep', preprocessor()),
              ('clf', LogisticRegression(max_iter=1000,
                                         random_state=42))]).fit(X_tr, y_tr)
seg = df.loc[X_te.index].copy()
seg['proba'] = m.predict_proba(X_te)[:, 1]
seg['error'] = (seg[TARGET] - seg['proba']).abs()

cols = ['tenure_months', 'contract', 'support_calls', 'monthly_charges',
        TARGET, 'proba']
print('confidently wrong -- said stay, they left:')
worst = seg[seg[TARGET] == 1].nsmallest(4, 'proba')[cols]
print(worst.round(3).to_string(index=False))
print('\nconfidently wrong -- said leave, they stayed:')
worst = seg[seg[TARGET] == 0].nlargest(4, 'proba')[cols]
print(worst.round(3).to_string(index=False))
```

```
confidently wrong -- said stay, they left:
 tenure_months contract  support_calls  monthly_charges  churned  proba
            23 Two Year              0            26.42        1  0.009
            41 One Year              0            53.26        1  0.031
            12 Two Year              1              NaN        1  0.042
            19 One Year              0            23.08        1  0.047

confidently wrong -- said leave, they stayed:
 tenure_months       contract  support_calls  monthly_charges  churned  proba
             5 Month-To-Month              3            73.19        0  0.782
             6 Month-To-Month              1            79.59        0  0.751
            17 Month-To-Month              4            86.67        0  0.729
             9 Month-To-Month              1            79.64        0  0.719
```

Long-tenure two-year customers who left anyway, and short-tenure month-to-month customers who stayed. Both groups contradict the strongest pattern in the data. In a real project this

is where you go and ask someone: is there a reason these people behaved differently, and is it something we could record?

Where the errors concentrate

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
import pandas as pd

m = Pipeline([('prep', preprocessor()),
              ('clf', LogisticRegression(max_iter=1000,
                                         random_state=42))]).fit(X_tr, y_tr)
seg = df.loc[X_te.index].copy()
seg['proba'] = m.predict_proba(X_te)[:, 1]
seg['error'] = (seg[TARGET] - seg['proba']).abs()
seg['tenure_band'] = pd.cut(seg['tenure_months'], [0, 6, 18, 36, 72],
                            labels=['0-6', '7-18', '19-36', '37-72'])

print(seg.groupby('tenure_band', observed=True)
      .agg(n=('error', 'size'), mean_error=('error', 'mean'),
           churn_rate=(TARGET, 'mean'), mean_proba=('proba', 'mean'))
      .round(4).to_string())
```

```
              n  mean_error  churn_rate  mean_proba
tenure_band
0-6           72      0.4086      0.4167      0.4470
7-18         314      0.3982      0.3854      0.3716
```

```
19-36         218       0.2457       0.1927       0.1886
37-72         146       0.0791       0.0548       0.0490
```

Compare the last two columns: where the mean predicted probability sits below the actual churn rate, the model systematically under-predicts that band. That is a bias you can act on, and it is invisible in any single aggregate number.

Fairness is the same technique with higher stakes

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import recall_score, precision_score
import pandas as pd

m = Pipeline([('prep', preprocessor()),
              ('clf', LogisticRegression(max_iter=1000,
                                         random_state=42))]).fit(X_tr, y_tr)
seg = df.loc[X_te.index].copy()
seg['pred'] = m.predict(X_te)
seg['truth'] = y_te.to_numpy()

rows = []
for name, g in seg.groupby('has_dependents', dropna=False):
    rows.append({'group': str(name), 'n': len(g),
                 'selection_rate': g['pred'].mean(),
                 'recall': recall_score(g['truth'], g['pred'], zero_division=0),
                 'precision': precision_score(g['truth'], g['pred'],
```

```
                                                zero_division=0)})
print(pd.DataFrame(rows).round(4).to_string(index=False))
```

```
group   n  selection_rate  recall  precision
   No 498         0.1988  0.4532     0.6364
  Yes 205         0.1561  0.4286     0.6562
  nan  47         0.1489  0.2308     0.4286
```

Equal treatment has several incompatible definitions

Equal selection rate, equal recall, and equal precision are three different fairness criteria, and it is mathematically impossible to satisfy all of them at once when the groups have different base rates. You have to choose which one your situation demands and say so out loud. For a retention offer, unequal selection rates may be fine. For a loan decision they are not.

> **Day 6 takeaway**

Day 7 A Reusable Evaluation Report

One function covering stability, discrimination, calibration and segments

Everything this week, as one function you can run on any classifier.

The report

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.metrics import (roc_auc_score, average_precision_score,
                             brier_score_loss, confusion_matrix,
                             precision_score, recall_score)
from sklearn.model_selection import cross_validate, StratifiedKFold
import numpy as np
import pandas as pd

def evaluate(model, X_tr, y_tr, X_te, y_te, frame, costs=None):
    """A full evaluation: stability, discrimination, calibration,
    the operating point, and where it fails."""
    cv = StratifiedKFold(5, shuffle=True, random_state=0)
    r = cross_validate(model, X_tr, y_tr, cv=cv, scoring='roc_auc',
                       return_train_score=True)
    print('CROSS-VALIDATION')
    print('  train AUC %.4f   val AUC %.4f +/- %.4f   gap %+.4f'
          % (r['train_score'].mean(), r['test_score'].mean(),
             r['test_score'].std(),
             r['train_score'].mean() - r['test_score'].mean()))

    model.fit(X_tr, y_tr)
    proba = model.predict_proba(X_te)[:, 1]
    print('\nHELD-OUT TEST')
    print('  ROC AUC           %.4f' % roc_auc_score(y_te, proba))
    print('  average precision %.4f  (baseline %.4f)'
          % (average_precision_score(y_te, proba), y_te.mean()))
    print('  Brier score       %.4f' % brier_score_loss(y_te, proba))

    if costs:
        offer, value, save = costs
        grid = np.arange(0.05, 0.95, 0.01)
        net = []
        for t in grid:
            tn, fp, fn, tp = confusion_matrix(y_te, (proba >=
    ↪ t).astype(int)).ravel()
            net.append(tp * save * value - (tp + fp) * offer)
        best_t = grid[int(np.argmax(net))]
        pred = (proba >= best_t).astype(int)
        print('\nOPERATING POINT')
        print('  threshold %.2f (net value %.0f, against %.0f at 0.5)'
              % (best_t, max(net), net[int(np.argmin(np.abs(grid - 0.5)))]))
        print('  precision %.4f   recall %.4f'
              % (precision_score(y_te, pred, zero_division=0),
                 recall_score(y_te, pred)))
    else:
        pred = (proba >= 0.5).astype(int)

    print('\nBY SEGMENT')
    seg = frame.loc[X_te.index].copy()
    seg['proba'], seg['truth'] = proba, y_te.to_numpy()
    for name, g in seg.groupby('contract'):
        if g['truth'].nunique() < 2:
            continue
```

```
        print('  %-16s n=%4d  churn %.3f  AUC %.4f'
              % (name, len(g), g['truth'].mean(),
                 roc_auc_score(g['truth'], g['proba'])))
    return proba

from sklearn.linear_model import LogisticRegression
model = Pipeline([('prep', preprocessor()),
                  ('clf', LogisticRegression(max_iter=1000, random_state=42))])
_ = evaluate(model, X_tr, y_tr, X_te, y_te, df, costs=(30, 250, 0.35))
```

```
CROSS-VALIDATION
  train AUC 0.8289   val AUC 0.8238 +/- 0.0305   gap +0.0052

HELD-OUT TEST
  ROC AUC           0.8174
  average precision 0.5993  (baseline 0.2680)
  Brier score       0.1475

OPERATING POINT
  threshold 0.34 (net value 4537, against 3472 at 0.5)
  precision 0.5423   recall 0.7015

BY SEGMENT
  Month-To-Month   n= 416  churn 0.413  AUC 0.7072
  One Year         n= 184  churn 0.114  AUC 0.8113
  Two Year         n= 150  churn 0.053  AUC 0.8688
```

Run it on a competitor

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor():
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), NUM),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ])
from sklearn.model_selection import train_test_split

```

```
X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import cross_validate, StratifiedKFold
from sklearn.metrics import (roc_auc_score, average_precision_score,
                             brier_score_loss)

cv = StratifiedKFold(5, shuffle=True, random_state=0)
for name, clf in [('logistic', __import__('sklearn.linear_model',
                    fromlist=['LogisticRegression']).LogisticRegression(
                        max_iter=1000, random_state=42)),
                  ('boosting', HistGradientBoostingClassifier(random_state=42))]:
    m = Pipeline([('prep', preprocessor()), ('clf', clf)])
    r = cross_validate(m, X_tr, y_tr, cv=cv, scoring='roc_auc',
                       return_train_score=True)
    m.fit(X_tr, y_tr)
    proba = m.predict_proba(X_te)[:, 1]
    print('%-9s val %.4f +/-%.4f  gap %+.4f  test AUC %.4f  AP %.4f  Brier %.4f'
          % (name, r['test_score'].mean(), r['test_score'].std(),
             r['train_score'].mean() - r['test_score'].mean(),
             roc_auc_score(y_te, proba), average_precision_score(y_te, proba),
             brier_score_loss(y_te, proba)))
```

```
logistic  val 0.8238 +/-0.0305  gap +0.0052  test AUC 0.8174  AP 0.5993  Brier 0.1475
boosting  val 0.7966 +/-0.0167  gap +0.1879  test AUC 0.7797  AP 0.5262  Brier 0.1690
```

Three numbers decide it: the validation score, the train-validation gap, and the Brier score. Boosting fits harder and generalises no better here, which is the same conclusion week 6 reached from a different direction.

What to write down

1. The metric you optimised and **why that one**.
2. The cross-validated score with its spread, never a bare number.
3. The train-validation gap.
4. The held-out test score, computed once.
5. The operating threshold and the cost assumptions behind it.
6. Performance broken down by every segment that matters.
7. The known failure modes, from the confidently-wrong rows.
8. The date, the data version, and the random seeds.

The last one is not bureaucracy

In six months someone will ask why the model behaves differently. Without the data version and the seeds you cannot reproduce what you did, so you cannot answer. Week 15 turns this into an artefact the pipeline writes automatically.

Your assignment

Run `evaluate` on the random forest and on naive Bayes. One will have a much larger train-validation gap; the other will have a much worse Brier score. Write one paragraph recommending a model for a retention programme that multiplies each probability by customer value, and note which of the two failings actually disqualifies a model for that use.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. Repeated measurements of the same customer appear in the data. The correct cross-validation is:

   a) `GroupKFold`, grouped by customer
   b) `ShuffleSplit`
   c) `KFold`
   d) `StratifiedKFold`

2. Selecting features on the whole dataset before cross-validating can make pure noise score:

   a) Well above chance, because the selection already saw the labels of every fold
   b) Exactly 1.00
   c) Below chance
   d) 0.50, as expected

3. Nested cross-validation exists to:

   a) Speed up tuning
   b) Give an honest performance estimate for a procedure that includes tuning
   c) Reduce variance
   d) Replace the test set

4. As a class becomes rarer, plain accuracy:

   a) Becomes undefined
   b) Falls
   c) Rises, because always predicting the majority gets easier
   d) Stays the same

5. A model ranks well but its predicted probabilities are systematically too high. It needs:

   a) A lower learning rate
   b) More data
   c) More features
   d) Calibration

6. A learning curve where training and validation scores have converged at a low value indicates:

    a) Underfitting, more data will not help, but more capacity or better features might

    b) Leakage

    c) A bad split

    d) Overfitting

7. The Brier score measures:

    a) Ranking quality

    b) The mean squared error of predicted probabilities, so it captures calibration as well as discrimination

    c) The area under the curve

    d) Class balance

8. Overall AUC is 0.82 but one customer segment scores 0.61. This means:

    a) AUC is the wrong metric

    b) The model is fine

    c) The headline number is hiding a group the model serves badly, which is a fairness and a quality problem

    d) The segment is too small to matter

Answers:1a, 2a, 3b, 4c, 5d, 6a, 7b, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 8

Feature Engineering

Week 8

OBJECTIVES

Turning raw columns into inputs a model can use, and measuring honestly enough to know which ones helped.

Better inputs beat better models, but only when you can tell which inputs are better. This week covers the transformations, encodings and interactions worth trying, the custom transformer that makes a group-statistic feature safe, and above all the discipline of measuring every idea against a fixed baseline on fixed folds, including the many that will not work.

Day 1 Why Features Beat Models

The discipline, the baseline, and a first attempt that barely moves

Week 6 ended with a week of ensembles drawing against a straight line. That is not a limit of the algorithms; it is a limit of the features. Better inputs beat better models, reliably, and this week is where the gains actually live.

The discipline that makes it safe

Every feature you build learns something, a mean, a category list, a bin edge. Learn it from all your data and you have leaked. So every engineered feature belongs inside the pipeline, fitted on training folds only. That is not a style preference; week 7 showed it turning noise into an apparent 0.80.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

```

```
def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(5, shuffle=True, random_state=0)
baseline = Pipeline([('prep', preprocessor()),
                     ('clf', LogisticRegression(max_iter=1000,
                                                random_state=42))])
s = cross_val_score(baseline, X_tr, y_tr, cv=cv, scoring='roc_auc')
print('the number to beat: %.4f  +/- %.4f' % (s.mean(), s.std()))
```

```
the number to beat: 0.8238  +/- 0.0305
```

Write the baseline down before you start

Feature engineering without a recorded baseline is how people spend a fortnight building features that made things worse. Every idea this week gets measured against 0.8238, on the same folds, and kept only if it beats it by more than the fold spread.

Where features come from

Source	Example	Watch out for
Transformation	`log(charges)`	Undefined at zero, use `log1p`
Ratio	charges per month of tenure	Division by zero; and it may reconstruct the target
Interaction	month-to-month *and* high support calls	Column count explodes quickly
Aggregate	average spend for this contract type	Must be computed on training folds only
Time	months since signup	Never derive from today's date
Domain knowledge	is this customer out of contract	The most valuable and the least automatable

A first attempt, measured

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
```

```
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold,
    ↪ train_test_split
import numpy as np

eng = df.copy()
eng['charge_per_tenure'] = eng['monthly_charges'] / eng['tenure_months']
eng['calls_per_year'] = eng['support_calls'] / (eng['tenure_months'] / 12)
eng['log_calls'] = np.log1p(eng['support_calls'])

NEW = NUM + ['charge_per_tenure', 'calls_per_year', 'log_calls']
Xe, ye = eng[NEW + CAT], eng[TARGET]
Xe_tr, Xe_te, ye_tr, ye_te = train_test_split(Xe, ye, test_size=0.25,
                                              stratify=ye, random_state=42)

cv = StratifiedKFold(5, shuffle=True, random_state=0)
m = Pipeline([('prep', preprocessor(num=NEW)),
              ('clf', LogisticRegression(max_iter=1000, random_state=42))])
s = cross_val_score(m, Xe_tr, ye_tr, cv=cv, scoring='roc_auc')
print('with three new features: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('baseline was            0.8238')
```

```
with three new features: 0.8278  +/- 0.0220
baseline was            0.8238
```

Barely moved, and that is the normal outcome of a first attempt. Ratios invented without a reason rarely help. The features that work come from knowing something about the problem, which is the rest of this week.

Day 1 takeaway

Day 2 Numeric Transformations

Choosing a scaler, and buying curvature with bins or splines

Scaling is not one choice. Which scaler you pick changes what the model sees, and the default is wrong whenever outliers are present.

The three scalers

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.preprocessing import (StandardScaler, MinMaxScaler,
                                   RobustScaler, QuantileTransformer)
import numpy as np

x = df[['support_calls']].to_numpy(dtype=float)   # contains the 97 outlier

print('%-22s %8s %8s %8s %8s' % ('', 'median', 'IQR', 'max', 'p99'))
print('%-22s %8.3f %8.3f %8.3f %8.3f'
      % ('raw', np.median(x), np.subtract(*np.percentile(x, [75, 25])),
         x.max(), np.percentile(x, 99)))
for name, sc in [('StandardScaler', StandardScaler()),
                 ('MinMaxScaler', MinMaxScaler()),
                 ('RobustScaler', RobustScaler()),
                 ('QuantileTransformer', QuantileTransformer(
                     output_distribution='normal', random_state=0))]:
    z = sc.fit_transform(x)
    print('%-22s %8.3f %8.3f %8.3f %8.3f'
          % (name, np.median(z), np.subtract(*np.percentile(z, [75, 25])),
             z.max(), np.percentile(z, 99)))
```

```
                        median      IQR      max      p99
raw                      1.000    2.000   97.000    5.000
StandardScaler          -0.127    0.952   45.590    1.778
MinMaxScaler             0.010    0.021    1.000    0.052
RobustScaler             0.000    1.000   48.000    2.000
QuantileTransformer     -0.045    5.890    5.199    2.483
```

Scaler	Centres on	Divides by	Use when
`StandardScaler`	Mean	Standard deviation	Roughly symmetric, no extreme values
`MinMaxScaler`	Minimum	Range	You need a bounded 0 to 1 range
`RobustScaler`	Median	Interquartile range	Outliers are present, usually the safer default
`QuantileTransformer`	-	-	You want a specific output shape regardless of input

Watch Out

MinMaxScaler is destroyed by a single outlier

One customer with 97 calls sets the maximum, so every other customer. The ones with 0 to 5 calls, is compressed into the bottom five percent of the range. The scaler has technically done its job and practically destroyed the column's resolution. `RobustScaler` uses quartiles, which one row cannot move.

Binning

Sometimes the relationship is not smooth. A customer at month 11 and one at month 13 may behave very differently if the contract renews at 12.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.linear_model import LogisticRegression
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import cross_val_score, StratifiedKFold,
    ↪ train_test_split
from sklearn.impute import SimpleImputer
```

```
from sklearn.preprocessing import OneHotEncoder, StandardScaler

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
cv = StratifiedKFold(5, shuffle=True, random_state=0)

binned = ColumnTransformer([
    ('bin', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('b', KBinsDiscretizer(n_bins=8, encode='onehot-dense',
                                             strategy='quantile'))]),
     ['tenure_months']),
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]),
     ['support_calls', 'monthly_charges']),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
m = Pipeline([('prep', binned),
              ('clf', LogisticRegression(max_iter=1000, random_state=42))])
s = cross_val_score(m, X_tr, y_tr, cv=cv, scoring='roc_auc')
print('tenure binned into 8: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('baseline:             0.8238')
```

```
tenure binned into 8: 0.8221  +/- 0.0295
baseline:             0.8238
```

Watch Out

Binning throws information away

Every customer inside a bin becomes identical. You gain the ability to express a non-linear step and you lose all resolution within the bin. It helps when the true relationship really does jump; it hurts when the relationship was smooth, which is the case here. Measure, do not assume.

Splines, which give curvature without losing resolution

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.preprocessing import SplineTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.compose import ColumnTransformer
```

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.model_selection import cross_val_score, StratifiedKFold,
    ↪ train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
cv = StratifiedKFold(5, shuffle=True, random_state=0)

spl = ColumnTransformer([
    ('spline', Pipeline([('i', SimpleImputer(strategy='median')),
                         ('s', SplineTransformer(n_knots=5, degree=3))]),
     ['tenure_months']),
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]),
     ['support_calls', 'monthly_charges']),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
m = Pipeline([('prep', spl),
              ('clf', LogisticRegression(max_iter=1000, random_state=42))])
s = cross_val_score(m, X_tr, y_tr, cv=cv, scoring='roc_auc')
print('tenure as a spline: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('baseline:           0.8238')
```

```
tenure as a spline: 0.8235  +/- 0.0303
baseline:           0.8238
```

A spline is a smooth curve built from overlapping local basis functions. It lets a linear model bend without the staircase that binning imposes, and it is usually the better tool when you suspect curvature.

Day 2 takeaway

Day 3 Encoding High-Cardinality Categories

Target encoding, how it leaks, and what makes it safe

One-hot encoding works until a column has a thousand levels. Then you need target encoding, which is powerful, and which leaks unless you are very careful.

The problem

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(0)
n = 5000
postcodes = ['PC%04d' % i for i in rng.integers(0, 900, n)]
```

```
s = pd.Series(postcodes)
print('rows            %d' % n)
print('distinct values %d' % s.nunique())
print('one-hot columns %d' % s.nunique())
print('\nlevels seen fewer than 5 times: %d'
      % (s.value_counts() < 5).sum())
```

```
rows            5000
distinct values 896
one-hot columns 896

levels seen fewer than 5 times: 307
```

The naive version, and why it fails

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold

rng = np.random.default_rng(0)
n = 2000
# A category with no relationship to the target whatsoever.
cat = pd.Series(['C%03d' % i for i in rng.integers(0, 400, n)])
y = pd.Series(rng.integers(0, 2, n))

# Naive: compute the mean target per category using every row.
means = y.groupby(cat).mean()
encoded = cat.map(means).to_frame('te')

cv = StratifiedKFold(5, shuffle=True, random_state=0)
s = cross_val_score(LogisticRegression(), encoded, y, cv=cv, scoring='roc_auc')
print('random category, random target -- truth is 0.5')
print('naive target encoding scores:  %.4f' % s.mean())
```

```
random category, random target -- truth is 0.5
naive target encoding scores:  0.7587
```

Watch Out

It encoded the answer into the feature

With 400 categories over 2000 rows, each category has about five rows. Its mean target is therefore mostly determined by those five labels, including the one you are about to predict. The feature is a leaky copy of the target, and cross-validation cannot save you because the leak happened before the folds existed.

The safe version

```
import numpy as np
```

```
import pandas as pd
from sklearn.preprocessing import TargetEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score, StratifiedKFold

rng = np.random.default_rng(0)
n = 2000
cat = np.array([['C%03d' % i] for i in rng.integers(0, 400, n)])
y = rng.integers(0, 2, n)

cv = StratifiedKFold(5, shuffle=True, random_state=0)
m = make_pipeline(TargetEncoder(random_state=0), LogisticRegression())
s = cross_val_score(m, cat, y, cv=cv, scoring='roc_auc')
print('sklearn TargetEncoder, inside a pipeline: %.4f' % s.mean())
print('the truth is 0.5 -- and now it says so')
```

```
sklearn TargetEncoder, inside a pipeline: 0.5055
the truth is 0.5 -- and now it says so
```

scikit-learn's `TargetEncoder` does two things that make it safe. It fits an internal cross-fitting scheme, so a row's encoding never uses its own label. And it shrinks each category mean toward the global mean in proportion to how few rows that category has, so a category seen twice barely moves away from the overall rate.

```
import numpy as np
from sklearn.preprocessing import TargetEncoder

cat = np.array([['common'] * 500 + ['rare'] * 3]).reshape(-1, 1)
y = np.array([1] * 250 + [0] * 250 + [1, 1, 1])   # rare is 100% positive

print('global mean       %.4f' % y.mean())
print('raw mean for rare 1.0000  (3 rows, all positive)\n')
print('%10s %12s %10s' % ('smooth', 'common', 'rare'))
for smooth in ['auto', 1.0, 5.0, 20.0]:
    enc = TargetEncoder(random_state=0, cv=5, smooth=smooth).fit(cat, y)
    print('%10s %12.4f %10.4f'
          % (smooth, enc.transform([['common']])[0, 0],
             enc.transform([['rare']])[0, 0]))
```

```
global mean       0.5030
raw mean for rare 1.0000  (3 rows, all positive)

    smooth       common       rare
      auto       0.5000     1.0000
       1.0       0.5000     0.8757
       5.0       0.5000     0.6894
      20.0       0.5001     0.5678
```

Watch Out

smooth='auto' did not shrink it at all

The default estimates how much to shrink from the *variance within* each category. Our rare group is three rows that are all 1, so its within-group variance is zero, the estimator concludes the mean is perfectly reliable, and it leaves 1.0 untouched. Three rows, and the encoder is certain.

An explicit `smooth` is a fixed number of pseudo-observations at the global mean, so it pulls regardless: 0.88 at 1, 0.69 at 5, 0.57 at 20. When your rare categories are small and internally consistent, which is exactly when they are most dangerous, set `smooth` yourself rather than trusting the automatic estimate.

On our data

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.preprocessing import TargetEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(5, shuffle=True, random_state=0)
te = ColumnTransformer([
```

```
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('t', TargetEncoder(random_state=0))]), CAT),
])
m = Pipeline([('prep', te),
              ('clf', LogisticRegression(max_iter=1000, random_state=42))])
s = cross_val_score(m, X_tr, y_tr, cv=cv, scoring='roc_auc')
print('target encoded: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('one-hot was:    0.8238')
```

```
target encoded: 0.8197  +/- 0.0271
one-hot was:    0.8238
```

Level with one-hot, which is expected: our categories have three or four levels each, so one-hot costs nothing and target encoding has nothing to compress. Reach for it at hundreds of levels, not at four.

Day 3 takeaway

Day 4 Interactions and Domain Features

The thing a linear model cannot do, and how to hand it over

A linear model cannot express "month-to-month customers who also call support a lot". It can only add the two effects. Handing it the combination is often where the real gain is.

Why a linear model needs help

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score

rng = np.random.default_rng(0)
n = 4000
a = rng.integers(0, 2, n)
b = rng.integers(0, 2, n)
y = (a ^ b)                                      # exclusive or
y = np.where(rng.uniform(size=n) < 0.05, 1 - y, y)

X = np.column_stack([a, b])
print('without the interaction %.4f'
      % cross_val_score(make_pipeline(StandardScaler(), LogisticRegression()),
                        X, y, cv=5, scoring='roc_auc').mean())

X2 = np.column_stack([a, b, a * b])
print('with a*b as a column    %.4f'
      % cross_val_score(make_pipeline(StandardScaler(), LogisticRegression()),
                        X2, y, cv=5, scoring='roc_auc').mean())
```

```
without the interaction 0.5073
with a*b as a column    0.9558
```

Chance, then near-perfect, from one extra column. Neither feature alone says anything about the target; only their combination does. A tree finds this by splitting twice; a linear model finds it only if you supply it.

Automatic interactions, and their cost

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.preprocessing import PolynomialFeatures, StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(5, shuffle=True, random_state=0)
inter = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
m = Pipeline([('prep', inter),
              ('poly', PolynomialFeatures(degree=2, interaction_only=True,
                                          include_bias=False)),
```

```
            ('clf', LogisticRegression(max_iter=3000, random_state=42))])
m.fit(X_tr, y_tr)
print('columns after encoding      %d'
      % m.named_steps['prep'].transform(X_tr).shape[1])
print('columns after interactions  %d'
      % m.named_steps['poly'].transform(
          m.named_steps['prep'].transform(X_tr)).shape[1])
s = cross_val_score(m, X_tr, y_tr, cv=cv, scoring='roc_auc')
print('\nall pairwise interactions: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('baseline:                  0.8238')
```

```
columns after encoding      15
columns after interactions  120

all pairwise interactions: 0.8140  +/- 0.0240
baseline:                  0.8238
```

Watch Out

Every pair, including the meaningless ones

`interaction_only=True` at degree 2 gives you every pairwise product, and most of them mean nothing, *fibre optic* times *mailed check* is not a concept. You have multiplied the column count and handed the model far more opportunity to overfit for one or two useful combinations. Pair it with regularisation, or engineer the interactions you can justify.

Deliberate features instead

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.linear_model import LogisticRegression
```

```python
from sklearn.model_selection import cross_val_score, StratifiedKFold,
    ↪ train_test_split
import numpy as np

eng = df.copy()

# 1. On a rolling contract AND unhappy: the combination we can justify.
eng['rolling_and_calling'] = (
    (eng['contract'] == 'Month-To-Month') & (eng['support_calls'] >= 2)
).astype(int)

# 2. Paying a lot relative to others on the same service.
eng['price_vs_service'] = (
    eng['monthly_charges']
    - eng.groupby('internet_service')['monthly_charges'].transform('median'))

# 3. Still new. Churn risk is concentrated in the first months.
eng['is_new'] = (eng['tenure_months'] <= 6).astype(int)

# 4. Calls per year of tenure, not raw calls.
eng['call_rate'] = eng['support_calls'] / np.maximum(eng['tenure_months'], 1) * 12

NEW = NUM + ['rolling_and_calling', 'price_vs_service', 'is_new', 'call_rate']
Xe, ye = eng[NEW + CAT], eng[TARGET]
Xe_tr, _, ye_tr, _ = train_test_split(Xe, ye, test_size=0.25,
                                      stratify=ye, random_state=42)

cv = StratifiedKFold(5, shuffle=True, random_state=0)
m = Pipeline([('prep', preprocessor(num=NEW)),
              ('clf', LogisticRegression(max_iter=1000, random_state=42))])
s = cross_val_score(m, Xe_tr, ye_tr, cv=cv, scoring='roc_auc')
print('four justified features: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('baseline:                0.8238')
```

```
four justified features: 0.8267  +/- 0.0246
baseline:                0.8238
```

Watch Out

price_vs_service is computed over the whole dataset

That group median uses every row, including the ones in the validation fold. It is a small leak and it is still a leak: the score above is very slightly optimistic. Day 6 rebuilds it as a proper transformer that learns its medians on training folds only, which is the correct way to ship a feature like this.

Day 4 takeaway

Day 5 Feature Selection

Filters, wrappers and embedded methods, and what filters cannot see

More features is not better. Each irrelevant column adds noise, slows fitting, and gives the model another chance to find a coincidence.

Watch noise columns do damage

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold
import numpy as np
import pandas as pd

cv = StratifiedKFold(5, shuffle=True, random_state=0)
rng = np.random.default_rng(0)

for extra in [0, 10, 50, 200]:
    Xn = X_tr.copy()
    for j in range(extra):
        Xn['noise_%d' % j] = rng.normal(size=len(Xn))
    num = NUM + ['noise_%d' % j for j in range(extra)]
    m = Pipeline([('prep', preprocessor(num=num)),
                  ('clf', LogisticRegression(max_iter=2000, random_state=42))])
    s = cross_val_score(m, Xn, y_tr, cv=cv, scoring='roc_auc')
    print('%4d noise columns: %.4f  +/- %.4f' % (extra, s.mean(), s.std()))
```

```
  0 noise columns: 0.8238  +/- 0.0305
 10 noise columns: 0.8194  +/- 0.0318
 50 noise columns: 0.8053  +/- 0.0342
200 noise columns: 0.7910  +/- 0.0275
```

Two thousand rows and two hundred pure-noise columns costs real performance. The model spends its capacity fitting coincidences, and regularisation only partly protects it.

Three families of method

Family	How	Cost	Weakness
Filter	Score each feature against the target alone	Very cheap	Blind to combinations
Wrapper	Repeatedly fit the model with subsets	Expensive	Can overfit the selection itself
Embedded	The model selects while fitting (lasso, trees)	Free	Tied to that model's assumptions

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.feature_selection import (SelectKBest, mutual_info_classif,
                                       RFE, SelectFromModel)
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(5, shuffle=True, random_state=0)
```

```
prep = preprocessor()

selectors = {
    'none           ': None,
    'filter, k=8    ': SelectKBest(mutual_info_classif, k=8),
    'wrapper, RFE 8': RFE(LogisticRegression(max_iter=1000),
    ↪ n_features_to_select=8),
    'embedded, L1   ': SelectFromModel(
        LogisticRegression(penalty='l1', solver='liblinear', C=0.1)),
}
for name, sel in selectors.items():
    steps = [('prep', prep)]
    if sel is not None:
        steps.append(('sel', sel))
    steps.append(('clf', LogisticRegression(max_iter=1000, random_state=42)))
    s = cross_val_score(Pipeline(steps), X_tr, y_tr, cv=cv, scoring='roc_auc')
    print('%s %.4f  +/- %.4f' % (name, s.mean(), s.std()))
```

```
none           0.8238  +/- 0.0305
filter, k=8    0.8184  +/- 0.0250
wrapper, RFE 8 0.8229  +/- 0.0313
embedded, L1   0.8234  +/- 0.0299
```

Selection lives inside the pipeline

Every selector above is a pipeline step, so it is refitted on each training fold and never sees the fold it is scored on. Week 7 showed what happens otherwise: selection over all the labels made pure noise score 0.80. There is no version of feature selection that is safe to do before splitting.

What filter methods miss

```
import numpy as np
from sklearn.feature_selection import mutual_info_classif, f_classif

rng = np.random.default_rng(0)
n = 4000
a = rng.integers(0, 2, n)
b = rng.integers(0, 2, n)
y = a ^ b
X = np.column_stack([a, b, rng.normal(size=n)])

print('mutual information with the target, feature by feature:')
for name, score in zip(['a', 'b', 'noise'], mutual_info_classif(X, y,
    ↪ random_state=0)):
    print('  %-6s %.5f' % (name, score))
print('\na and b together determine y exactly. Individually both')
print('score near zero -- a is indistinguishable from pure noise.')
```

```
mutual information with the target, feature by feature:
  a      0.00000
  b      0.01169
  noise  0.00000
```

```
a and b together determine y exactly. Individually both
score near zero -- a is indistinguishable from pure noise.
```

Any filter method evaluates one feature at a time, so it cannot see that two useless-looking columns are jointly decisive. If interactions matter in your problem, a filter will discard exactly the features you need.

Day 5 takeaway

Day 6 Writing Your Own Transformer

Turning a leaky groupby into a pipeline step that fits per fold

Day 4 built a feature from a group median computed over the whole dataset, and flagged it as a leak. Today you fix it properly, by writing a transformer that obeys the scikit-learn contract.

The contract, from week 1

A transformer needs `fit`, which learns and returns `self`, and `transform`, which applies what was learned. Inherit two base classes and you get `fit_transform`, `get_params` and pipeline compatibility for nothing.

```python
import numpy as np
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin

class GroupRelativeValue(BaseEstimator, TransformerMixin):
    """How far a customer's value sits from the median of their group.

    The medians are learned in fit, from training rows only, so this is
    safe to use inside cross-validation -- which the day 4 version, built
    with a groupby over the whole frame, was not.
    """

    def __init__(self, group_col, value_col):
        self.group_col = group_col
        self.value_col = value_col

    def fit(self, X, y=None):
        self.medians_ = X.groupby(self.group_col)[self.value_col].median()
        self.global_median_ = X[self.value_col].median()
        return self

    def transform(self, X):
        # An unseen group falls back to the global median rather than NaN.
        base = X[self.group_col].map(self.medians_).fillna(self.global_median_)
        return (X[self.value_col] - base).to_frame('relative_value')

    def get_feature_names_out(self, input_features=None):
        return np.array(['relative_value'])
```

```

demo = pd.DataFrame({'internet_service': ['DSL', 'DSL', 'Fibre optic', 'Fibre optic'],
                     'monthly_charges': [50.0, 60.0, 75.0, 85.0]})
t = GroupRelativeValue('internet_service', 'monthly_charges').fit(demo)
print('learned medians:')
print(t.medians_.to_string())
print('\ntransformed:')
print(t.transform(demo).to_string(index=False))

unseen = pd.DataFrame({'internet_service': ['Satellite'], 'monthly_charges': [90.0]})
print('\nunseen group falls back to the global median:')
print(t.transform(unseen).to_string(index=False))
```

```
learned medians:
internet_service
DSL            55.0
Fibre optic    80.0

transformed:
 relative_value
           -5.0
            5.0
           -5.0
            5.0

unseen group falls back to the global median:
 relative_value
           22.5
```

Put it in a pipeline

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
```

```
                           ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                           ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.model_selection import train_test_split

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold
import numpy as np

class GroupRelativeValue(BaseEstimator, TransformerMixin):
    def __init__(self, group_col, value_col):
        self.group_col = group_col
        self.value_col = value_col

    def fit(self, X, y=None):
        self.medians_ = X.groupby(self.group_col)[self.value_col].median()
        self.global_median_ = X[self.value_col].median()
        return self

    def transform(self, X):
        base = X[self.group_col].map(self.medians_).fillna(self.global_median_)
        return (X[self.value_col] - base).fillna(0).to_frame('relative_value')

    def get_feature_names_out(self, input_features=None):
        return np.array(['relative_value'])


prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ('rel', Pipeline([('g', GroupRelativeValue('internet_service',
                                               'monthly_charges')),
                      ('s', StandardScaler())]),
     ['internet_service', 'monthly_charges']),
])

cv = StratifiedKFold(5, shuffle=True, random_state=0)
m = Pipeline([('prep', prep),
              ('clf', LogisticRegression(max_iter=1000, random_state=42))])
s = cross_val_score(m, X_tr, y_tr, cv=cv, scoring='roc_auc')
print('leak-free relative price: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('baseline:                 0.8238')
```

```
leak-free relative price: 0.8239  +/- 0.0308
```

```
baseline:                0.8238
```

The medians are refitted on every fold

Five folds, five different sets of medians, each learned from four fifths of the data and applied to the fifth. No validation row contributed to the statistic used to encode it. That is the whole point of writing the transformer rather than doing a `groupby` in your notebook.

FunctionTransformer, for anything stateless

```python
import numpy as np
import pandas as pd
from sklearn.preprocessing import FunctionTransformer

def add_ratios(X):
    out = X.copy()
    out['calls_per_year'] = (out['support_calls']
                             / np.maximum(out['tenure_months'], 1) * 12)
    out['log_calls'] = np.log1p(out['support_calls'])
    return out

# No feature_names_out='one-to-one' here: that promises the same columns
# out as in, and this function adds two. sklearn checks, and raises.
step = FunctionTransformer(add_ratios)
demo = pd.DataFrame({'support_calls': [0, 3, 12], 'tenure_months': [2, 24, 6]})
print(step.fit_transform(demo).round(3).to_string(index=False))
```

```
 support_calls  tenure_months  calls_per_year  log_calls
             0              2             0.0      0.000
             3             24             1.5      1.386
            12              6            24.0      2.565
```

Watch Out

Only when the function learns nothing

`FunctionTransformer` has no `fit` worth the name, so it is safe exactly when your function uses no statistic from the data. A ratio of two columns in the same row is fine. Anything involving a mean, a median, a category list or a bin edge needs a real transformer with a `fit`.

Day 6 takeaway

Day 7 A Complete Feature Engineering Project

Every idea measured against the baseline, failures included

Everything this week, applied properly, measured honestly, against the 0.8238 that a week of

ensembles could not beat.

The candidate features

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (StandardScaler, OneHotEncoder,
                                   SplineTransformer, FunctionTransformer)
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (cross_val_score, StratifiedKFold,
                                     train_test_split)
import numpy as np

def enrich(X):
    out = X.copy()
    out['rolling_and_calling'] = (
        (out['contract'] == 'Month-To-Month') & (out['support_calls'] >= 2)
    ).astype(int)
    out['is_new'] = (out['tenure_months'] <= 6).astype(int)
    out['call_rate'] = (out['support_calls']
                        / np.maximum(out['tenure_months'], 1) * 12)
    return out

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)

sample = enrich(X_tr.head(4))
print(sample[['contract', 'support_calls', 'tenure_months',
```

```
              'rolling_and_calling', 'is_new', 'call_rate']]
      .round(2).to_string(index=False))
```

```
      contract  support_calls  tenure_months  rolling_and_calling  is_new  call_rate
Month-To-Month              0              9                    0       0        0.0
      Two Year              0             21                    0       0        0.0
Month-To-Month              0              2                    0       1        0.0
Month-To-Month              0             12                    0       0        0.0
```

Assemble and measure

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocessor(num=NUM, cat=CAT):
    return ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), cat),
    ])
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (StandardScaler, OneHotEncoder,
                                   SplineTransformer, FunctionTransformer)
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (cross_val_score, StratifiedKFold,
                                     train_test_split)
import numpy as np

class GroupRelativeValue(BaseEstimator, TransformerMixin):
    def __init__(self, group_col, value_col):
        self.group_col = group_col
        self.value_col = value_col

    def fit(self, X, y=None):
        self.medians_ = X.groupby(self.group_col)[self.value_col].median()
        self.global_median_ = X[self.value_col].median()
```

```
        return self

    def transform(self, X):
        base = X[self.group_col].map(self.medians_).fillna(self.global_median_)
        return (X[self.value_col] - base).fillna(0).to_frame('rel')

    def get_feature_names_out(self, input_features=None):
        return np.array(['rel'])


def enrich(X):
    out = X.copy()
    out['rolling_and_calling'] = (
        (out['contract'] == 'Month-To-Month') & (out['support_calls'] >= 2)
    ).astype(int)
    out['is_new'] = (out['tenure_months'] <= 6).astype(int)
    out['call_rate'] = (out['support_calls']
                        / np.maximum(out['tenure_months'], 1) * 12)
    return out


DERIVED = ['rolling_and_calling', 'is_new', 'call_rate']

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('derived', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), DERIVED),
    ('spline', Pipeline([('i', SimpleImputer(strategy='median')),
                         ('s', SplineTransformer(n_knots=5, degree=3))]),
     ['tenure_months']),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
    ('rel', Pipeline([('g', GroupRelativeValue('internet_service',
                                               'monthly_charges')),
                      ('s', StandardScaler())]),
     ['internet_service', 'monthly_charges']),
])

X, y = df[NUM + CAT], df[TARGET]
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
cv = StratifiedKFold(5, shuffle=True, random_state=0)

full = Pipeline([
    ('enrich', FunctionTransformer(enrich)),
    ('prep', prep),
    ('clf', LogisticRegression(max_iter=3000, random_state=42)),
])
s = cross_val_score(full, X_tr, y_tr, cv=cv, scoring='roc_auc')
print('engineered: %.4f  +/- %.4f' % (s.mean(), s.std()))
print('baseline:   0.8238')
print('gain:       %+.4f  (fold spread %.4f)' % (s.mean() - 0.8238, s.std()))
```

```
engineered: 0.8259  +/- 0.0252
baseline:   0.8238
gain:       +0.0021  (fold spread 0.0252)
```

Does it help the ensembles too?

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv', parse_dates=['signup_date'])
df = df.drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
TARGET = 'churned'
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder, FunctionTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import (cross_val_score, StratifiedKFold,
                                     train_test_split)
import numpy as np

def enrich(X):
    out = X.copy()
    out['rolling_and_calling'] = (
        (out['contract'] == 'Month-To-Month') & (out['support_calls'] >= 2)
    ).astype(int)
    out['is_new'] = (out['tenure_months'] <= 6).astype(int)
    out['call_rate'] = (out['support_calls']
                        / np.maximum(out['tenure_months'], 1) * 12)
    return out

DERIVED = ['rolling_and_calling', 'is_new', 'call_rate']
X, y = df[NUM + CAT], df[TARGET]
X_tr, _, y_tr, _ = train_test_split(X, y, test_size=0.25, stratify=y,
                                    random_state=42)
cv = StratifiedKFold(5, shuffle=True, random_state=0)

def build(num, clf):
    prep = ColumnTransformer([
        ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                          ('s', StandardScaler())]), num),
        ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                          ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT)])
    return Pipeline([('enrich', FunctionTransformer(enrich)),
                     ('prep', prep), ('clf', clf)])

for name, clf in [('logistic', LogisticRegression(max_iter=3000, random_state=42)),
```

```
                  ('boosting', HistGradientBoostingClassifier(random_state=42))]:
    plain = cross_val_score(build(NUM, clf), X_tr, y_tr, cv=cv,
                            scoring='roc_auc').mean()
    rich = cross_val_score(build(NUM + DERIVED, clf), X_tr, y_tr, cv=cv,
                           scoring='roc_auc').mean()
    print('%-9s plain %.4f   engineered %.4f   change %+.4f'
          % (name, plain, rich, rich - plain))
```

```
logistic  plain 0.8238   engineered 0.8268   change +0.0030
boosting  plain 0.7966   engineered 0.7958   change -0.0009
```

The features help the linear model more than the trees

That is exactly what you should expect. `is_new` and `rolling_and_calling` are thresholds and interactions, and a tree can already discover those by splitting. Handing them to a linear model gives it something it genuinely could not express. Feature engineering is most valuable for the models with the least flexibility.

The honest verdict

The gains this week are small. That is worth saying plainly, because most material on feature engineering implies otherwise. On this dataset the generator used a straightforward weighted sum, so there is not much structure left for a clever feature to expose, and the honest result of measuring carefully is often that an idea did not work.

Watch Out

Report the failures, and be strict about what counts as a win

The engineered pipeline gains about half a point of AUC, and the fold-to-fold spread is five times that. By the standard set on day 1, keep it only if it beats the baseline by more than the spread. This week has not produced a result you could defend. Say so.
You measured roughly a dozen ideas and most changed nothing. That is a normal, healthy ratio. The discipline is not in producing features that work; it is in measuring against a fixed baseline on fixed folds so you can tell the difference, and in discarding what does not earn its place rather than shipping it because you built it.

Your assignment

The dataset generator used `payment_method == 'Electronic check'` with a coefficient of +0.42. Build a feature that captures an interaction between payment method and something else, and measure it. Then try a feature the generator definitely did not use, a ratio of charges to support calls, say. Record both results, including the one that fails.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. Before engineering any feature you should first:

   a) Remove outliers
   b) Tune the model
   c) Establish a baseline score to measure every later change against
   d) Scale everything

2. Binning or splining a numeric feature buys a linear model:

   a) Robustness to outliers
   b) Faster fitting
   c) Fewer parameters
   d) Curvature, the ability to respond non-linearly to that feature

3. Target encoding leaks when:

   a) The encoding is computed using the target values of the rows it is applied to
   b) The target is continuous
   c) There are too many categories
   d) The category is rare

4. The safe way to apply target encoding is:

   a) Compute it once on all the data
   b) Fit it inside the pipeline so each fold's encoding is learned from that fold's training rows, with smoothing
   c) Use the median instead
   d) Drop rare categories first

5. An interaction feature is needed when:

   a) The target is skewed
   b) Two features are correlated
   c) The effect of one feature depends on the value of another, something a linear model cannot represent unaided
   d) There are missing values

6. A filter method for feature selection cannot detect:

   a) Constant columns
   b) Correlated pairs
   c) Strong single predictors
   d) A feature that is useless alone but informative in combination with another

7. Writing a custom transformer rather than a pandas `groupby` matters because:

   a) The transformer refits per fold, so the statistic it computes never sees validation rows

b) It handles missing data

c) scikit-learn requires it

d) It is faster

8. A feature engineering project should report:

a) Only the ideas that worked

b) Every idea measured against the baseline, including the ones that made it worse

c) The final model only

d) The best cross-validation score

Answers:1c, 2d, 3a, 4b, 5c, 6d, 7a, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

Chapter 9

Unsupervised Learning and Clustering

Week 9

OBJECTIVES

Finding structure without labels, k-means, hierarchical, DBSCAN and mixtures, plus how to tell whether the structure is real.

With no target column there is no score to check against, which makes it unusually easy to find groups that are not there. This week covers the four main clustering families and how each fails, the distance problem that categorical columns create, and above all the validation, silhouette, stability and holding back the outcome. That separates a real segmentation from a partition of a cloud.

Day 1 k-means From Scratch

The algorithm in four steps, and the three assumptions it makes silently

Eight weeks of supervised learning: you had the answers and searched for the rule. Now the answers are gone. Nobody labelled these customers, and the question is whether they fall into groups at all.

k-means, in full

1. Place k centres at random.
2. Assign every point to its nearest centre.
3. Move each centre to the mean of the points assigned to it.
4. Repeat 2 and 3 until nothing moves.

```
import numpy as np

rng = np.random.default_rng(0)
X = np.vstack([rng.normal([0, 0], 0.6, (60, 2)),
               rng.normal([4, 4], 0.6, (60, 2)),
               rng.normal([0, 5], 0.6, (60, 2))])

centres = X[rng.choice(len(X), 3, replace=False)]
for step in range(1, 11):
    d = ((X[:, None] - centres[None, :]) ** 2).sum(axis=2)
    labels = d.argmin(axis=1)
    new = np.array([X[labels == k].mean(axis=0) for k in range(3)])
    shift = np.abs(new - centres).max()
```

```
    centres = new
    if step <= 3 or shift < 1e-6:
        print('step %2d  largest centre movement %.6f' % (step, shift))
    if shift < 1e-6:
        break

print('\nfinal centres:')
print(np.round(centres[centres[:, 0].argsort()], 2))
```

```
step  1  largest centre movement 2.336408
step  2  largest centre movement 1.994041
step  3  largest centre movement 0.043246
step  4  largest centre movement 0.000000

final centres:
[[-0.    4.94]
 [-0.    0.1 ]
 [ 3.83  4.02]]
```

Converged in a handful of passes, and the centres landed on the three clusters the data was built from. Twelve lines, no library.

The library version, and the argument that matters

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.cluster import KMeans
import numpy as np

km = KMeans(n_clusters=4, n_init=10, random_state=42).fit(Z)
print('cluster sizes:', np.bincount(km.labels_))
print('inertia: %.1f' % km.inertia_)
print('iterations to converge:', km.n_iter_)

tiny = np.bincount(km.labels_).argmin()
print('\nthe smallest cluster, in full:')
print(df.loc[km.labels_ == tiny,
             ['customer_id', 'tenure_months', 'support_calls',
              'monthly_charges']].to_string(index=False))
```

```
cluster sizes: [1641  822  536    1]
inertia: 3012.8
iterations to converge: 8

the smallest cluster, in full:
customer_id  tenure_months  support_calls  monthly_charges
     C00008             48             97            76.42
```

Watch Out

One of your four segments has a single member

It is the customer with 97 support calls from week 2. k-means partitions *everything*, so a point far from all the others gets its own centre rather than being flagged as unusual, and you have spent a quarter of your segmentation on one person. This is the "every point belongs somewhere" assumption failing in the most visible way possible, and it is why day 4's DBSCAN, which can label a point as noise, matters.

Watch Out

n_init exists because k-means gets stuck

The result depends on where the centres started. A bad start converges to a bad local optimum, and the algorithm has no way to notice. `n_init=10` runs the whole thing ten times from different starts and keeps the lowest inertia. Setting `n_init=1` to save time is how you get a different segmentation every Monday.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.cluster import KMeans
import numpy as np

print('%8s %14s %14s' % ('n_init', 'best inertia', 'worst inertia'))
for n_init in [1, 5, 25]:
    results = [KMeans(n_clusters=5, n_init=n_init, random_state=s).fit(Z).inertia_
               for s in range(8)]
    print('%8d %14.1f %14.1f' % (n_init, min(results), max(results)))
```

```
  n_init   best inertia  worst inertia
```

```
      1         2498.7         2628.4
      5         2498.7         2498.7
     25         2498.7         2498.7
```

Scaling decides the answer

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.cluster import KMeans
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import numpy as np

PAIR = ['monthly_charges', 'total_charges']
raw = SimpleImputer(strategy='median').fit_transform(df[PAIR])
scaler = StandardScaler().fit(raw)

for name, data in [('unscaled', raw), ('scaled  ', scaler.transform(raw))]:
    centres = KMeans(n_clusters=3, n_init=10,
    ↪ random_state=0).fit(data).cluster_centers_
    if name.strip() == 'scaled':
        centres = scaler.inverse_transform(centres)
    order = centres[:, 1].argsort()
    print('%s centres (monthly, total):' % name)
    print(np.round(centres[order], 1))
```

```
unscaled centres (monthly, total):
[[  53.1  665.4]
 [  69.5 2118.9]
 [  75.9 4450.9]]
scaled   centres (monthly, total):
[[  28.9  590.1]
 [  70.3 1101.6]
 [  75.2 3673. ]]
```

Total charges runs to 7,600 and monthly charges to 106, so unscaled the distance is almost entirely about the total. The three clusters differ hugely in total charges, 665, 2,119, 4,451, and barely at all in monthly charges, 53 to 76. Monthly charges has been ignored.

Scaled, the first cluster's monthly charge drops to 29, which is the no-internet group finally being allowed to separate itself. Neither answer is wrong arithmetic; only one of them is the question you meant to ask.

What k-means assumes

Assumption	Consequence when false
Clusters are roughly spherical	Elongated groups get cut in half
Clusters are similar in size	Large ones get split, small ones absorbed
Every point belongs somewhere	Outliers drag centres toward themselves
You know k	You do not, and the algorithm will not tell you

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import make_moons
from sklearn.metrics import adjusted_rand_score

X, truth = make_moons(n_samples=400, noise=0.06, random_state=0)
labels = KMeans(n_clusters=2, n_init=10, random_state=0).fit_predict(X)
print('two interleaved crescents, k-means agreement with the truth: %.3f'
      % adjusted_rand_score(truth, labels))
print('(1.0 is perfect, 0.0 is random)')
```

```
two interleaved crescents, k-means agreement with the truth: 0.274
(1.0 is perfect, 0.0 is random)
```

k-means fails completely here, because it can only draw straight boundaries between centres and a crescent is not a ball. Day 4's DBSCAN solves exactly this.

> **Day 1 takeaway**

Day 2 Choosing k, and Whether There Are Clusters At All

Why inertia cannot answer it, and what silhouette and stability can

k-means needs you to supply k, and the whole point of clustering is that you do not know it. There are three honest answers and one dishonest one.

The dishonest one: inertia

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

```

```
prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.cluster import KMeans

print('%4s %14s' % ('k', 'inertia'))
for k in range(1, 11):
    km = KMeans(n_clusters=k, n_init=10, random_state=42).fit(Z)
    print('%4d %14.1f' % (k, km.inertia_))
```

```
   k        inertia
   1         9000.0
   2         6818.7
   3         4748.4
   4         3012.8
   5         2498.7
   6         2151.8
   7         1888.1
   8         1659.7
   9         1496.8
  10         1377.3
```

Watch Out

Inertia always falls, so it can never choose k

Adding a centre can only reduce the distance from points to their nearest centre. At k equal to the number of rows, inertia is zero and the clustering is useless. The “elbow method” asks you to spot where the curve bends, which on real data is usually a smooth arc with no bend at all. Look at the numbers above and try to name the elbow.

Silhouette: how well separated are the clusters

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, davies_bouldin_score

print('%4s %14s %18s' % ('k', 'silhouette', 'davies-bouldin'))
for k in range(2, 9):
```

```
    labels = KMeans(n_clusters=k, n_init=10, random_state=42).fit_predict(Z)
    print('%4d %14.4f %18.4f'
          % (k, silhouette_score(Z, labels), davies_bouldin_score(Z, labels)))
```

```
   k     silhouette     davies-bouldin
   2         0.3201             1.2820
   3         0.3225             0.8410
   4         0.3706             0.7130
   5         0.2970             0.8996
   6         0.2970             0.9680
   7         0.2951             0.9170
   8         0.2965             0.9071
```

Higher silhouette is better; lower Davies-Bouldin is better. Read them together, and treat agreement between two different criteria as weak evidence rather than proof.

Watch Out

A low silhouette everywhere means there are no clusters

Which is a perfectly respectable finding, and the one most people refuse to accept. Our customers were generated from continuous distributions with no group structure at all, so any k-means result here is a partition of a cloud, not a discovery of groups. Partitions can still be useful for targeting, but calling them "segments we found" is a claim the data does not support.

Check against data that genuinely has clusters

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score

X, _ = make_blobs(n_samples=900, centers=4, cluster_std=0.8,
                  n_features=3, random_state=0)
print('data built from four real clusters:')
for k in range(2, 8):
    labels = KMeans(n_clusters=k, n_init=10, random_state=0).fit_predict(X)
    print('  k=%d  silhouette %.4f' % (k, silhouette_score(X, labels)))
```

```
data built from four real clusters:
  k=2  silhouette 0.5886
  k=3  silhouette 0.6695
  k=4  silhouette 0.6581
  k=5  silhouette 0.5398
  k=6  silhouette 0.4460
  k=7  silhouette 0.3641
```

A clear peak at the true k, and values far above anything the customer data produced. That contrast is the calibration you need: this is what real structure looks like.

Stability: would you get the same answer next month?

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score
import numpy as np

rng = np.random.default_rng(0)
print('%4s %22s' % ('k', 'agreement across samples'))
for k in [2, 3, 4, 6]:
    scores = []
    for _ in range(6):
        idx_a = rng.choice(len(Z), int(0.8 * len(Z)), replace=False)
        idx_b = rng.choice(len(Z), int(0.8 * len(Z)), replace=False)
        shared = np.intersect1d(idx_a, idx_b)
        la = KMeans(k, n_init=10, random_state=0).fit(Z[idx_a])
        lb = KMeans(k, n_init=10, random_state=0).fit(Z[idx_b])
        scores.append(adjusted_rand_score(la.predict(Z[shared]),
                                          lb.predict(Z[shared])))
    print('%4d %22.4f' % (k, np.mean(scores)))
```

```
   k agreement across samples
   2                  0.8236
   3                  0.8786
   4                  0.7375
   6                  0.7785
```

Stability is the most practical criterion

Cluster two overlapping samples and see whether the same customers end up together. A segmentation that reshuffles when you add a month of data cannot be the basis of a marketing programme, whatever its silhouette says. This test costs nothing and almost nobody runs it.

> **Day 2 takeaway**

Day 3 Hierarchical Clustering

Every k at once, and how linkage decides the shape of what you find

k-means makes you choose k in advance. Hierarchical clustering builds the whole nested family of solutions and lets you cut it wherever you like.

How it merges

```
import numpy as np
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster

rng = np.random.default_rng(0)
X = np.vstack([rng.normal([0, 0], 0.4, (12, 2)),
               rng.normal([3, 3], 0.4, (12, 2))])

Zl = linkage(X, method='ward')
print('the last five merges (cluster a, cluster b, distance, size):')
print(np.round(Zl[-5:], 3))

for k in [2, 3, 4]:
    labels = fcluster(Zl, k, criterion='maxclust')
    print('cut into %d: sizes %s' % (k, np.bincount(labels)[1:]))
```

```
the last five merges (cluster a, cluster b, distance, size):
[[20.    41.     1.004  9.   ]
 [38.    39.     1.054  9.   ]
 [40.    43.     1.557 12.   ]
 [36.    42.     1.687 12.   ]
 [44.    45.    15.417 24.   ]]
cut into 2: sizes [12 12]
cut into 3: sizes [12  3  9]
cut into 4: sizes [3 9 3 9]
```

Look at the distance column: the final merge happens at a far larger distance than the ones before it. That jump is the signature of two genuinely separate groups being forced together, and it is how you read a dendrogram.

Linkage decides the shape of what you find

Linkage	Distance between clusters	Tends to produce
`ward`	Increase in within-cluster variance	Compact, similar-sized clusters
`complete`	Furthest pair	Compact, sensitive to outliers
`average`	Mean over all pairs	A compromise
`single`	Closest pair	Long chains, can follow shapes, often produces one giant blob

```
import numpy as np
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_moons
from sklearn.metrics import adjusted_rand_score

X, truth = make_moons(n_samples=400, noise=0.06, random_state=0)
for method in ['ward', 'complete', 'average', 'single']:
    labels = AgglomerativeClustering(n_clusters=2, linkage=method).fit_predict(X)
```

```
    print('%-9s agreement with the truth %.3f'
          % (method, adjusted_rand_score(truth, labels)))
```

```
ward      agreement with the truth 0.239
complete  agreement with the truth 0.285
average   agreement with the truth 0.329
single    agreement with the truth 1.000
```

Single linkage solves the crescents

Because it only asks whether two clusters have *any* close pair, it can follow a curved shape point by point along its length. Ward and complete linkage insist on compactness and cut the crescents in half, exactly as k-means did. The same chaining behaviour makes single linkage fragile on noisy data, where one bridging point merges two real clusters.

On the customers

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.cluster import AgglomerativeClustering, KMeans
from sklearn.metrics import silhouette_score, adjusted_rand_score
import numpy as np

sample = Z[np.random.default_rng(0).choice(len(Z), 1200, replace=False)]

for k in [3, 4, 5]:
    ward = AgglomerativeClustering(n_clusters=k, linkage='ward').fit_predict(sample)
    km = KMeans(k, n_init=10, random_state=0).fit_predict(sample)
    print('k=%d  ward silhouette %.4f   kmeans %.4f   they agree %.3f'
          % (k, silhouette_score(sample, ward), silhouette_score(sample, km),
             adjusted_rand_score(ward, km)))
```

```
k=3  ward silhouette 0.3314   kmeans 0.3173   they agree 0.452
k=4  ward silhouette 0.2958   kmeans 0.3727   they agree 0.400
k=5  ward silhouette 0.2636   kmeans 0.3017   they agree 0.371
```

Watch Out

It does not scale

Agglomerative clustering computes distances between every pair, which is quadratic in memory and worse in time. Three thousand rows is fine; a million is impossible. That is why the snippet above takes a sample, and why k-means remains the default for large data despite its assumptions.

Day 3 takeaway

Day 4 DBSCAN and Density

Clusters as dense regions, arbitrary shapes, and outliers named as noise

k-means partitions everything, including the points that belong to no group. DBSCAN starts from a different idea: a cluster is a dense region, and everything else is noise.

It solves the crescents

```
import numpy as np
from sklearn.cluster import DBSCAN, KMeans
from sklearn.datasets import make_moons
from sklearn.metrics import adjusted_rand_score

X, truth = make_moons(n_samples=400, noise=0.06, random_state=0)

km = KMeans(2, n_init=10, random_state=0).fit_predict(X)
db = DBSCAN(eps=0.25, min_samples=5).fit_predict(X)
print('k-means agreement %.3f' % adjusted_rand_score(truth, km))
print('DBSCAN  agreement %.3f' % adjusted_rand_score(truth, db))
print('clusters found: %d, noise points: %d'
      % (len(set(db)) - (1 if -1 in db else 0), (db == -1).sum()))
```

```
k-means agreement 0.274
DBSCAN  agreement 1.000
clusters found: 2, noise points: 0
```

No k required, arbitrary shapes handled, and outliers named rather than forced into a group. The cost is that `eps` now has to be chosen, and it is less forgiving than k.

eps is the whole difficulty

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons
from sklearn.metrics import adjusted_rand_score

X, truth = make_moons(n_samples=400, noise=0.06, random_state=0)
```

```
print('%8s %10s %10s %12s' % ('eps', 'clusters', 'noise', 'agreement'))
for eps in [0.05, 0.12, 0.25, 0.4, 1.0]:
    lab = DBSCAN(eps=eps, min_samples=5).fit_predict(X)
    n_clusters = len(set(lab)) - (1 if -1 in lab else 0)
    print('%8s %10d %10d %12.3f'
          % (eps, n_clusters, (lab == -1).sum(),
             adjusted_rand_score(truth, lab)))
```

```
     eps   clusters      noise    agreement
    0.05         28        218        0.022
    0.12          2          2        0.990
    0.25          2          0        1.000
     0.4          1          0        0.000
     1.0          1          0        0.000
```

Watch Out

Too small and everything is noise; too large and everything is one cluster

At 0.05 almost every point fails the density test. At 1.0 the two crescents merge. The usable window here runs from about 0.12 to 0.4, and on real data it can be much narrower. This sensitivity is DBSCAN's real weakness.

Choosing eps from the data

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
from sklearn.datasets import make_moons

X, _ = make_moons(n_samples=400, noise=0.06, random_state=0)

# Distance to the 5th nearest neighbour, sorted. The 'knee' in that curve
# is roughly where points stop being in dense regions.
d, _ = NearestNeighbors(n_neighbors=5).fit(X).kneighbors(X)
fifth = np.sort(d[:, 4])
print('percentile of 5th-neighbour distance:')
for q in [50, 75, 90, 95, 99]:
    print('  %3d%%  %.4f' % (q, np.percentile(fifth, q)))
print('\na reasonable eps sits near the 90th to 95th percentile')
```

```
percentile of 5th-neighbour distance:
   50%  0.0642
   75%  0.0837
   90%  0.1051
   95%  0.1179
   99%  0.1489

a reasonable eps sits near the 90th to 95th percentile
```

On the customers, where density is uniform

```python
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.cluster import DBSCAN
import numpy as np

print('%8s %10s %10s %12s' % ('eps', 'clusters', 'noise', 'largest'))
for eps in [0.3, 0.5, 0.8, 1.2]:
    lab = DBSCAN(eps=eps, min_samples=10).fit_predict(Z)
    n_clusters = len(set(lab)) - (1 if -1 in lab else 0)
    sizes = np.bincount(lab[lab >= 0]) if n_clusters else np.array([0])
    print('%8s %10d %10d %12d'
          % (eps, n_clusters, (lab == -1).sum(), sizes.max()))
```

```
     eps   clusters      noise      largest
     0.3         11        411          905
     0.5          1         43         2957
     0.8          1          4         2996
     1.2          1          2         2998
```

Either almost everything is noise, or almost everything is one enormous cluster. There is no setting that produces several balanced groups, because there are no dense regions separated by sparse ones. DBSCAN is telling you the same thing the silhouette scores did in day 2, and it is telling you more clearly.

	k-means	Hierarchical	DBSCAN
Need k in advance	Yes	No	No
Cluster shapes	Spherical	Depends on linkage	Any
Handles outliers	No	Poorly	Yes, explicitly
Scales to millions	Yes	No	Moderately
Main difficulty	Choosing k	Choosing linkage	Choosing eps

Day 4 takeaway

Day 5 Gaussian Mixture Models

Soft assignment, cluster shape, and a selection criterion with a real minimum

k-means gives every point one label. A Gaussian mixture gives every point a probability of belonging to each cluster, which is both more honest and more useful.

Soft assignment

```
import numpy as np
from sklearn.mixture import GaussianMixture

rng = np.random.default_rng(0)
X = np.vstack([rng.normal([0, 0], 1.0, (300, 2)),
               rng.normal([3, 3], 1.0, (300, 2))])

gm = GaussianMixture(n_components=2, random_state=0).fit(X)
probs = gm.predict_proba(X)

confident = (probs.max(axis=1) > 0.95).mean()
print('points assigned with >95%% confidence: %.1f%%' % (100 * confident))
print('points that are genuinely ambiguous:  %.1f%%'
      % (100 * (probs.max(axis=1) < 0.7).mean()))

border = np.argsort(np.abs(probs[:, 0] - 0.5))[:3]
print('\nmost ambiguous points:')
for i in border:
    print('  at %s  probabilities %s' % (X[i].round(2), probs[i].round(3)))
```

```
points assigned with >95% confidence: 92.3%
points that are genuinely ambiguous:  1.8%

most ambiguous points:
  at [0.48 2.69]  probabilities [0.51 0.49]
  at [1.79 1.27]  probabilities [0.527 0.473]
  at [0.45 2.66]  probabilities [0.466 0.534]
```

Those borderline customers are real. k-means would have assigned each of them firmly to one side and told you nothing about the doubt.

Covariance shape is the reason to prefer it

```
import numpy as np
from sklearn.mixture import GaussianMixture
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score

rng = np.random.default_rng(0)
# Two elongated, differently oriented clouds -- not spheres.
a = rng.normal(0, 1, (400, 2)) @ np.array([[3.0, 0.0], [0.0, 0.3]])
b = rng.normal(0, 1, (400, 2)) @ np.array([[0.3, 0.0], [0.0, 3.0]]) + [4, 0]
X = np.vstack([a, b])
truth = np.r_[np.zeros(400), np.ones(400)]

km = KMeans(2, n_init=10, random_state=0).fit_predict(X)
gm = GaussianMixture(2, covariance_type='full', random_state=0).fit_predict(X)
```

```
print('k-means agreement %.3f' % adjusted_rand_score(truth, km))
print('GMM     agreement %.3f' % adjusted_rand_score(truth, gm))
```

```
k-means agreement 0.393
GMM     agreement 0.851
```

k-means is a special case of a GMM

Fix every covariance to the same spherical shape and give every point a hard assignment, and the mixture model reduces exactly to k-means. `covariance_type='full'` lets each component have its own shape and orientation, which is what makes it fit the two elongated clouds above.

covariance_type	Each component gets	Parameters
`full`	Its own shape and orientation	Most
`tied`	A shared shape	Fewer
`diag`	Axis-aligned, own scale per axis	Fewer still
`spherical`	One radius	Fewest, nearly k-means

Choosing the number of components properly

Unlike k-means, a mixture model has a likelihood, so you can use information criteria, which penalise complexity and therefore do *not* improve forever.

```
import numpy as np
from sklearn.mixture import GaussianMixture
from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=900, centers=4, cluster_std=0.9,
                  n_features=3, random_state=0)
print('%4s %14s %14s' % ('k', 'BIC', 'AIC'))
for k in range(1, 9):
    gm = GaussianMixture(k, covariance_type='full', random_state=0).fit(X)
    print('%4d %14.1f %14.1f' % (k, gm.bic(X), gm.aic(X)))
```

```
   k            BIC            AIC
   1        12623.6        12580.3
   2        10613.2        10521.9
   3         9782.0         9642.7
   4         9638.6         9451.3
   5         9695.3         9460.0
   6         9761.8         9478.5
   7         9798.7         9467.3
   8         9870.6         9491.2
```

BIC has a minimum, unlike inertia

This is the real advantage over the elbow method. BIC adds a penalty proportional to the number of parameters, so it falls while extra components genuinely explain the data and rises once they are just fitting noise. The minimum is an actual answer rather than a judgement call about where a curve bends.

On the customers

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.mixture import GaussianMixture

print('%4s %14s' % ('k', 'BIC'))
best = None
for k in range(1, 9):
    gm = GaussianMixture(k, covariance_type='full', random_state=0).fit(Z)
    bic = gm.bic(Z)
    if best is None or bic < best[1]:
        best = (k, bic)
    print('%4d %14.1f' % (k, bic))
print('\nlowest BIC at k = %d' % best[0])
```

```
   k            BIC
   1        25540.7
   2        21961.8
   3        20710.0
   4        20479.7
   5        19982.6
   6        19816.1
   7        19807.6
   8        19895.4

lowest BIC at k = 7
```

> **Day 5 takeaway**

Day 6 Clustering Mixed Data

Why one-hot encoding makes your categories outvote everything else

Every algorithm this week needs distances, and our data is half categorical. What is the distance between a fibre optic customer and a DSL one? The answer you choose changes everything.

The one-hot trap

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
import numpy as np

cat = make_pipeline(SimpleImputer(strategy='most_frequent'),
                    OneHotEncoder(sparse_output=False)).fit_transform(df[CAT])
num = make_pipeline(SimpleImputer(strategy='median'),
                    StandardScaler()).fit_transform(df[NUM])

print('numeric columns     %d' % num.shape[1])
print('one-hot columns     %d' % cat.shape[1])
print('\ntypical distance contributed by each block:')
rng = np.random.default_rng(0)
i, j = rng.integers(0, len(num), (2, 400))
print('  numeric  %.3f' % np.linalg.norm(num[i] - num[j], axis=1).mean())
print('  one-hot  %.3f' % np.linalg.norm(cat[i] - cat[j], axis=1).mean())
```

```
numeric columns     3
one-hot columns     12

typical distance contributed by each block:
  numeric  1.954
  one-hot  2.111
```

Watch Out

Four categorical variables became twelve columns

Three numeric columns contribute 1.95 of typical distance between two customers; twelve one-hot columns contribute 2.11. So slightly more than half the notion of "similar customer" is now categorical, and worse, it is unevenly spread: contract type gets three columns and payment method four, so payment method counts more than tenure, monthly charges or support calls individually.
Nobody decided that. It fell out of how many levels each variable happened to have.

Three honest options

1. **Weight the blocks** so numeric and categorical contribute comparably.
2. **Use a distance built for mixed data**, such as Gower's, which handles each column type on

its own terms.

3. **Cluster within categories**: segment fibre customers separately from DSL ones. Often the most interpretable.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import numpy as np

cat = make_pipeline(SimpleImputer(strategy='most_frequent'),
                    OneHotEncoder(sparse_output=False)).fit_transform(df[CAT])
num = make_pipeline(SimpleImputer(strategy='median'),
                    StandardScaler()).fit_transform(df[NUM])

print('%10s %14s %28s' % ('cat weight', 'silhouette', 'largest cluster is'))
for w in [0.0, 0.25, 0.5, 1.0]:
    M = np.hstack([num, cat * w])
    lab = KMeans(4, n_init=10, random_state=0).fit_predict(M)
    big = np.bincount(lab).argmax()
    top = df.loc[lab == big, 'internet_service'].value_counts(normalize=True)
    print('%10.2f %14.4f %20s %6.0f%%'
          % (w, silhouette_score(M, lab), top.index[0], 100 * top.iloc[0]))
```

```
cat weight     silhouette          largest cluster is
      0.00         0.3706         Fibre optic     66%
      0.25         0.3403         Fibre optic     67%
      0.50         0.2856         Fibre optic     68%
      1.00         0.2083                 DSL     60%
```

At weight 0, numeric columns only. The silhouette is 0.37. At full weight, which is what plain one-hot encoding gives you by default, it falls to 0.21 and the dominant service in the largest cluster flips from fibre optic to DSL. Letting the categories in has made the clusters measurably worse *and* changed what they are about, and the default is the worst setting on the table.

Watch Out

Do not read this as a search for the best weight

The weight that maximises silhouette here is zero, which just says the numeric columns cluster more cleanly on their own. If the categorical variables genuinely matter to your question you cannot discard them because they score badly. You need a distance that

treats them fairly, which is the next section. Optimising silhouette over the weight would be choosing what to measure by what scores well.

Gower distance, implemented

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
import numpy as np

def gower(A, B, numeric_cols, cat_cols, ranges):
    """Mean per-column dissimilarity: scaled absolute difference for
    numeric columns, 0 or 1 for categorical ones."""
    total = np.zeros(len(A))
    for c in numeric_cols:
        total += np.abs(A[c].to_numpy() - B[c].to_numpy()) / ranges[c]
    for c in cat_cols:
        total += (A[c].to_numpy() != B[c].to_numpy()).astype(float)
    return total / (len(numeric_cols) + len(cat_cols))

d = df.dropna(subset=NUM + CAT).reset_index(drop=True)
ranges = {c: d[c].max() - d[c].min() for c in NUM}

a = d.iloc[[0, 0, 0]].reset_index(drop=True)
b = d.iloc[[1, 2, 3]].reset_index(drop=True)
print('customer 0 against three others:')
print(np.round(gower(a, b, NUM, CAT, ranges), 4))
print('\n0 is identical, 1 is maximally different.')
print('Every column contributes exactly once, whatever its type.')
```

```
customer 0 against three others:
[0.4508 0.3574 0.7095]

0 is identical, 1 is maximally different.
Every column contributes exactly once, whatever its type.
```

One column, one vote

That is the whole idea, and it is what one-hot encoding breaks. Gower scales each numeric column by its range so it lands in 0 to 1, scores each categorical column as match or mismatch, and averages. A three-level variable and a seventeen-level variable then carry the same weight, which is almost always what you meant.

Day 6 takeaway

Day 7 A Customer Segmentation Project

Features, k, stability, business description, and an honest validation

A segmentation, done properly, including the part where you check whether it means anything.

1. Decide what the segmentation is for

This decides the features. Segmenting for a retention campaign means behaviour and value; segmenting for network planning means service and usage. There is no general-purpose segmentation, and starting with "cluster the customers" is how you end up with groups nobody can act on.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import numpy as np

# For a retention programme: how long, how much, how unhappy.
FEATURES = ['tenure_months', 'monthly_charges', 'support_calls']
prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[FEATURES])
print('clustering on %s' % FEATURES)
print('rows %d, columns %d' % Z.shape)
```

```
clustering on ['tenure_months', 'monthly_charges', 'support_calls']
rows 3000, columns 3
```

2. Choose k with more than one criterion

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.impute import SimpleImputer
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score, davies_bouldin_score

FEATURES = ['tenure_months', 'monthly_charges', 'support_calls']
Z = make_pipeline(SimpleImputer(strategy='median'),
                  StandardScaler()).fit_transform(df[FEATURES])

print('%4s %12s %14s %14s' % ('k', 'silhouette', 'davies-b', 'BIC'))
for k in range(2, 8):
    lab = KMeans(k, n_init=10, random_state=42).fit_predict(Z)
    bic = GaussianMixture(k, covariance_type='full', random_state=0).fit(Z).bic(Z)
    print('%4d %12.4f %14.4f %14.1f'
          % (k, silhouette_score(Z, lab), davies_bouldin_score(Z, lab), bic))
```

```
   k   silhouette       davies-b           BIC
   2       0.3201         1.2820       21961.8
   3       0.3225         0.8410       20710.0
   4       0.3706         0.7130       20479.7
   5       0.2970         0.8996       19982.6
   6       0.2970         0.9680       19816.1
   7       0.2951         0.9170       19807.6
```

3. Check it is stable

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score
import numpy as np

FEATURES = ['tenure_months', 'monthly_charges', 'support_calls']
Z = make_pipeline(SimpleImputer(strategy='median'),
                  StandardScaler()).fit_transform(df[FEATURES])
rng = np.random.default_rng(0)

for k in [3, 4, 5]:
    agree = []
    for _ in range(8):
        ia = rng.choice(len(Z), int(0.8 * len(Z)), replace=False)
        ib = rng.choice(len(Z), int(0.8 * len(Z)), replace=False)
```

```
        shared = np.intersect1d(ia, ib)
        a = KMeans(k, n_init=10, random_state=0).fit(Z[ia])
        b = KMeans(k, n_init=10, random_state=0).fit(Z[ib])
        agree.append(adjusted_rand_score(a.predict(Z[shared]),
                                         b.predict(Z[shared])))
    print('k=%d  stability %.4f  +/- %.4f' % (k, np.mean(agree), np.std(agree)))
```

```
k=3  stability 0.7134  +/- 0.4151
k=4  stability 0.8612  +/- 0.2177
k=5  stability 0.9153  +/- 0.0507
```

4. Describe the segments in business language

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.cluster import KMeans
import pandas as pd

FEATURES = ['tenure_months', 'monthly_charges', 'support_calls']
Z = make_pipeline(SimpleImputer(strategy='median'),
                  StandardScaler()).fit_transform(df[FEATURES])

seg = df.copy()
seg['segment'] = KMeans(4, n_init=10, random_state=42).fit_predict(Z)

profile = seg.groupby('segment').agg(
    customers=('customer_id', 'size'),
    tenure=('tenure_months', 'median'),
    monthly=('monthly_charges', 'median'),
    calls=('support_calls', 'mean'),
    churn_rate=('churned', 'mean'),
)
profile['monthly_value'] = (profile['customers'] * profile['monthly']).round(0)
print(profile.round(2).to_string())
```

```
         customers  tenure  monthly  calls  churn_rate  monthly_value
segment
0             1641    15.0    74.22   1.41        0.40       121787.0
1              822    16.0    27.98   0.82        0.16        23004.0
2              536    52.0    68.40   1.33        0.04        36660.0
3                1    48.0    76.42  97.00        0.00           76.0
```

Churn rate was not a clustering feature

It was deliberately left out, so the fact that it differs across segments is real information rather than something you built in. If it had been a feature, finding that the segments differ in churn would be circular. Always hold back the variable you want to explain and check it afterwards.

5. Test whether the segments predict anything

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import make_pipeline
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold
import numpy as np

FEATURES = ['tenure_months', 'monthly_charges', 'support_calls']
prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[FEATURES])
seg = KMeans(4, n_init=10, random_state=42).fit_predict(Z)
y = df['churned']
cv = StratifiedKFold(5, shuffle=True, random_state=0)

onehot = OneHotEncoder(sparse_output=False).fit_transform(seg.reshape(-1, 1))
print('segment label alone      AUC %.4f'
      % cross_val_score(LogisticRegression(max_iter=1000), onehot, y,
                        cv=cv, scoring='roc_auc').mean())
print('raw features alone       AUC %.4f'
      % cross_val_score(LogisticRegression(max_iter=1000), Z, y,
                        cv=cv, scoring='roc_auc').mean())
print('raw features + segment   AUC %.4f'
      % cross_val_score(LogisticRegression(max_iter=1000),
                        np.hstack([Z, onehot]), y, cv=cv,
                        scoring='roc_auc').mean())
```

```
segment label alone      AUC 0.6977
raw features alone       AUC 0.7721
raw features + segment   AUC 0.7735
```

The segment label carries real information, but less than the features it was derived from, and adding it to those features gains little. That is the usual finding, and it is worth stating: a segmentation is a communication tool, a way to give four groups names that a marketing team can act on. It is rarely a way to improve a model.

Your assignment

Rebuild the segmentation with `contract` one-hot encoded and added to the features. The silhouette will improve. Then print the contract breakdown of each segment, and decide whether you have discovered anything that `df.groupby('contract')` would not have told you.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. k-means assumes clusters are:

   a) Any shape
   b) Roughly spherical, similarly sized and separable by distance to a centre
   c) Density-based
   d) Hierarchical

2. Inertia cannot be used to choose k because it:

   a) Requires labels
   b) Is expensive
   c) Falls monotonically as k rises, reaching zero when every point is its own cluster
   d) Ignores cluster shape

3. A silhouette score near zero means:

   a) The clusters are dense
   b) k is too small
   c) Excellent separation
   d) Points sit about as close to a neighbouring cluster as to their own

4. DBSCAN's advantage over k-means is that it:

   a) Finds arbitrarily shaped clusters and labels points in sparse regions as noise rather than forcing them into a cluster
   b) Needs no parameters
   c) Always finds the right k
   d) Is faster

5. A Gaussian mixture differs from k-means mainly in that it:

   a) Uses more clusters
   b) Assigns each point a probability of belonging to each cluster and can model elongated shapes
   c) Is deterministic
   d) Requires scaling

6. One-hot encoding several categorical columns before clustering causes:

a) Better silhouette scores

b) Faster convergence

c) The categories to dominate the distance, because each contributes its own dimension

d) Missing values

7. Cluster stability is assessed by:

a) Increasing k

b) Comparing to labels

c) Running the algorithm once

d) Re-running on resamples or different seeds and checking whether the same groupings recur

8. A segmentation is only useful if:

a) The segments differ in ways that change what the business would do about them

b) There are exactly five

c) Every customer is assigned

d) The silhouette score is high

Answers:1b, 2c, 3d, 4a, 5b, 6c, 7d, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

Chapter 10

Dimensionality Reduction and Anomaly Detection

Week 10

OBJECTIVES

Compressing wide data without losing what matters, and finding the rows that do not belong.

Two problems that share a cause. Wide, correlated data breaks distance, hides structure and cannot be looked at; this week reduces it, visualises it, and then uses the same machinery to find the rows that do not fit, including how to evaluate a detector when, by definition, you have no labels to check it against.

Day 1 Principal Component Analysis

Directions of greatest variance, reading loadings, and why scaling comes first

Week 5 showed distances losing meaning as dimensions grow. Week 8 showed noise columns costing real accuracy. Both problems have the same treatment: fewer, better dimensions.

Why fewer dimensions

- **Distance stops working**: every point becomes equidistant, which breaks kNN and clustering.
- **Data gets sparse**: the volume you need to fill grows exponentially with dimensions.
- **Correlated columns are redundant**: tenure and total charges carry much of the same information.
- **You cannot look at it**, and looking is how you notice things no metric reports.

PCA, in one idea

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
```

```
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.decomposition import PCA
import numpy as np

pca = PCA().fit(Z)
print('%10s %14s %16s' % ('component', 'variance', 'cumulative'))
cum = np.cumsum(pca.explained_variance_ratio_)
for i, (v, c) in enumerate(zip(pca.explained_variance_ratio_, cum), 1):
    print('%10d %14.4f %16.4f' % (i, v, c))
```

```
 component       variance       cumulative
         1         0.4893           0.4893
         2         0.2770           0.7663
         3         0.2176           0.9839
         4         0.0161           1.0000
```

Four columns in, and the first two components carry most of the variance. That is what correlation looks like from the other side: `tenure_months` and `total_charges` correlate at 0.83, so they are largely one direction wearing two names.

Reading the components

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.decomposition import PCA
import pandas as pd

pca = PCA(n_components=3).fit(Z)
loadings = pd.DataFrame(pca.components_.T, index=NUM,
                        columns=['PC1', 'PC2', 'PC3'])
print(loadings.round(3).to_string())
```

```
                  PC1    PC2    PC3
tenure_months    0.610 -0.415  0.255
support_calls    0.147  0.668  0.729
monthly_charges  0.343  0.613 -0.631
total_charges    0.699 -0.079 -0.066
```

Loadings are how you name a component

PC1 loads heavily and in the same direction on tenure and total charges. It is a *customer lifetime* axis. PC2 is dominated by monthly charges and support calls, which is closer to *current spend and friction*. Components are not automatically meaningful, but when they are, saying so is worth more than the variance figure.

Scaling first is mandatory

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

raw = SimpleImputer(strategy='median').fit_transform(df[NUM])
scaled = StandardScaler().fit_transform(raw)

for name, data in [('unscaled', raw), ('scaled  ', scaled)]:
    p_ = PCA(n_components=2).fit(data)
    print('%s  PC1 explains %.1f%%   loadings %s'
          % (name, 100 * p_.explained_variance_ratio_[0],
             p_.components_[0].round(3)))
```

```
unscaled  PC1 explains 100.0%   loadings [0.012 0.    0.008 1.   ]
scaled    PC1 explains 48.9%   loadings [0.61  0.147 0.343 0.699]
```

Watch Out

Unscaled, PC1 is just your largest column

PCA maximises variance, and variance depends on units. `total_charges` has a variance thousands of times larger than `support_calls` purely because it is measured in pounds rather than counts, so the first component points almost entirely along it and explains nearly everything. Change total charges to thousands of pounds and the answer changes. Always scale first.

On data where it earns its keep

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.decomposition import PCA
import numpy as np
```

```

from sklearn.preprocessing import StandardScaler

print('handwritten digits: %d images, %d pixels each' % Xd.shape)
pca = PCA().fit(StandardScaler().fit_transform(Xd))
cum = np.cumsum(pca.explained_variance_ratio_)
for target in [0.80, 0.90, 0.95, 0.99]:
    k = int(np.searchsorted(cum, target)) + 1
    print('  %.0f%% of the variance needs %3d of 64 components'
          % (100 * target, k))
```

```
handwritten digits: 1797 images, 64 pixels each
  80% of the variance needs  21 of 64 components
  90% of the variance needs  31 of 64 components
  95% of the variance needs  40 of 64 components
  99% of the variance needs  54 of 64 components
```

Ninety-five percent of what distinguishes these images survives in 40 numbers instead of 64, and 80 percent in 21. Neighbouring pixels are highly correlated. That is what an image is, and PCA turns that redundancy into a smaller representation.

Day 1 takeaway

Day 2 PCA in Practice

Choosing components, measuring the cost, and the case where PCA destroys your signal

Knowing what PCA does is not the same as knowing when it helps. Often it does not, and measuring is the only way to find out.

How many components to keep

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
import time

print('%12s %12s %10s' % ('components', 'accuracy', 'seconds'))
for k in [2, 5, 10, 20, 40, 64]:
    m = make_pipeline(StandardScaler(), PCA(n_components=k, random_state=0),
                      LogisticRegression(max_iter=2000))
    t = time.perf_counter()
    acc = cross_val_score(m, Xd, yd, cv=5).mean()
    print('%12d %12.4f %10.2f' % (k, acc, time.perf_counter() - t))
```

```
 components     accuracy    seconds
          2       0.5348       0.17
          5       0.7713       0.18
         10       0.8403       0.19
         20       0.8993       0.17
         40       0.9138       0.18
         64       0.9199       0.20
```

Accuracy climbs steeply to about 20 components and then flattens. Beyond that you are paying for dimensions that carry noise as readily as signal.

PCA is a pipeline step, like everything else

It learns the component directions from data, so fitting it before the split leaks. Inside a `Pipeline` it is refitted on each training fold. This is the same rule as every transformer since week 1, and PCA is one of the easiest to get wrong because it feels like "just a transformation".

Let it choose for you

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

m = make_pipeline(StandardScaler(), PCA(n_components=0.95, random_state=0))
m.fit(Xd)
print('components kept for 95%% of variance: %d'
      % m.named_steps['pca'].n_components_)
print('original dimensions: %d' % Xd.shape[1])
```

```
components kept for 95% of variance: 40
original dimensions: 64
```

Passing a float between 0 and 1 asks for that proportion of variance and lets PCA work out the count. It is usually a better default than guessing an integer.

Reconstruction: what did you throw away?

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.decomposition import PCA
import numpy as np

print('%12s %18s' % ('components', 'mean pixel error'))
for k in [2, 5, 10, 20, 40]:
    pca = PCA(n_components=k, random_state=0).fit(Xd)
    back = pca.inverse_transform(pca.transform(Xd))
    print('%12d %18.4f' % (k, np.abs(Xd - back).mean()))
print('\npixel values run 0 to 16')
```

```
 components   mean pixel error
          2             2.4791
          5             1.9511
         10             1.4733
         20             0.9475
         40             0.2731

pixel values run 0 to 16
```

inverse_transform is the honest check

Project down, project back, and compare. It tells you in the units of your original data what the compression cost, which a variance percentage never quite does. It is also how PCA gets used for anomaly detection: a row that reconstructs badly is a row that does not fit the structure the components captured.

When PCA does not help

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.pipeline import make_pipeline

y = df['churned']
cv = StratifiedKFold(5, shuffle=True, random_state=0)

print('%14s %12s' % ('components', 'churn AUC'))
print('%14s %12.4f'
      % ('all 4 raw',
         cross_val_score(LogisticRegression(max_iter=1000), Z, y, cv=cv,
                         scoring='roc_auc').mean()))
for k in [1, 2, 3]:
    m = make_pipeline(PCA(n_components=k, random_state=0),
                      LogisticRegression(max_iter=1000))
    print('%14d %12.4f'
          % (k, cross_val_score(m, Z, y, cv=cv, scoring='roc_auc').mean()))
```

```
components     churn AUC
 all 4 raw        0.7718
         1        0.5941
         2        0.7647
         3        0.7718
```

Watch Out

PCA is blind to your target

It maximises variance, and variance is not the same as usefulness. A direction can carry very little variance and be exactly what separates your classes; PCA will discard it first. With only four columns there is nothing to gain here and something to lose. Use PCA when you have hundreds of correlated columns, not four tidy ones.

```python
import numpy as np
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline

rng = np.random.default_rng(0)
n = 1000
# A huge-variance direction that says nothing, and a tiny one that says all.
y = rng.integers(0, 2, n)
loud_noise = rng.normal(0, 20, n)
quiet_signal = y * 0.6 + rng.normal(0, 0.25, n)
X = np.column_stack([loud_noise, quiet_signal])

print('variance of the noise column  %.1f' % X[:, 0].var())
print('variance of the signal column %.3f' % X[:, 1].var())
print('\nboth columns    AUC %.4f'
      % cross_val_score(LogisticRegression(), X, y, cv=5,
                        scoring='roc_auc').mean())
print('after PCA to 1  AUC %.4f'
      % cross_val_score(make_pipeline(PCA(n_components=1), LogisticRegression()),
                        X, y, cv=5, scoring='roc_auc').mean())
```

```
variance of the noise column  382.0
variance of the signal column 0.160

both columns    AUC 0.9578
after PCA to 1  AUC 0.5335
```

PCA kept the loud, useless direction and discarded the quiet, decisive one, taking the model from near-perfect to chance. Note that this example deliberately does not scale, scaling would have equalised the variances and saved it. That is the lesson twice over.

Day 2 takeaway

Day 3 t-SNE and Non-Linear Embedding

Seeing clusters PCA cannot separate, and the three things the plot does not mean

PCA is a rotation: it can only find structure that lies along straight directions. For seeing clusters, the non-linear methods do far better, and they come with a warning label.

PCA against t-SNE, on data with real clusters

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
import numpy as np

Xs = StandardScaler().fit_transform(Xd)

p2 = PCA(n_components=2, random_state=0).fit_transform(Xs)
t2 = TSNE(n_components=2, random_state=0, perplexity=30,
          init='pca').fit_transform(Xs)

# How well do the ten digits separate in each 2-D map?
for name, emb in [('PCA  ', p2), ('t-SNE', t2)]:
    acc = cross_val_score(KNeighborsClassifier(10), emb, yd, cv=5).mean()
    print('%s 2-D map, 10-NN accuracy %.4f' % (name, acc))
```

```
PCA   2-D map, 10-NN accuracy 0.5203
t-SNE 2-D map, 10-NN accuracy 0.9516
```

Two dimensions either way. In the PCA map the digits overlap heavily; in the t-SNE map a nearest-neighbour classifier gets most of them right, which means the clusters are genuinely separated on the page.

What t-SNE will not tell you

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
import numpy as np

Xs = StandardScaler().fit_transform(Xd)[:600]
labels = yd[:600]

for perp in [5, 30, 50]:
    emb = TSNE(n_components=2, random_state=0, perplexity=perp,
               init='pca').fit_transform(Xs)
    # Distance between the centres of digit 0 and digit 1 in the map,
```

```
    # relative to the overall spread.
    c0 = emb[labels == 0].mean(axis=0)
    c1 = emb[labels == 1].mean(axis=0)
    spread = emb.std()
    print('perplexity %2d  gap between the 0 and 1 clusters: %.2f spreads'
          % (perp, np.linalg.norm(c0 - c1) / spread))
```

```
perplexity  5  gap between the 0 and 1 clusters: 1.30 spreads
perplexity 30  gap between the 0 and 1 clusters: 2.90 spreads
perplexity 50  gap between the 0 and 1 clusters: 3.19 spreads
```

Watch Out

Distances between clusters are not meaningful

The gap between two clusters changes with `perplexity`, and it is not measuring anything about the data. t-SNE optimises local neighbourhoods; the space between well-separated groups is essentially arbitrary. Never say "these two clusters are closer than those two" from a t-SNE plot. Cluster sizes on the page are meaningless for the same reason.

The three rules for using it

1. **It is for looking, not for features.** There is no `transform` to apply to new data, so it cannot sit in a prediction pipeline.
2. **Run it at several perplexities.** Anything that appears at only one setting is an artefact.
3. **Never cluster the output.** Clustering an embedding that was optimised to make clusters look separated will find clusters. Cluster the original space.

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

Xs = StandardScaler().fit_transform(Xd)[:400]
tsne = TSNE(n_components=2, random_state=0, init='pca', perplexity=30)
tsne.fit_transform(Xs)
print('t-SNE has a transform method:', hasattr(tsne, 'transform'))
print('PCA has a transform method:  ', hasattr(PCA(2).fit(Xs), 'transform'))
```

```
t-SNE has a transform method: False
PCA has a transform method:   True
```

Speed, and the usual first step

```
from sklearn.datasets import load_digits
digits = load_digits()
Xd, yd = digits.data, digits.target
```

```
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import time

Xs = StandardScaler().fit_transform(Xd)

t = time.perf_counter()
TSNE(n_components=2, random_state=0, init='pca').fit_transform(Xs)
direct = time.perf_counter() - t

t = time.perf_counter()
reduced = PCA(n_components=20, random_state=0).fit_transform(Xs)
TSNE(n_components=2, random_state=0, init='pca').fit_transform(reduced)
staged = time.perf_counter() - t

print('t-SNE on all 64 dimensions   %.2fs' % direct)
print('PCA to 20 first, then t-SNE  %.2fs' % staged)
```

```
t-SNE on all 64 dimensions   6.59s
PCA to 20 first, then t-SNE  5.62s
```

Running PCA to a few dozen dimensions before t-SNE is standard practice. The time saved is modest at this size and grows with the dataset. More importantly it removes noise, and generally improves the result. UMAP, which is not in scikit-learn, does a similar job faster and does provide a `transform`, but its cluster distances deserve the same scepticism.

Day 3 takeaway

Day 4 SVD, NMF and Feature Agglomeration

Sparse text, additive topics, and reductions that stay explainable

PCA is one of several matrix factorisations, and the others exist because PCA's assumptions do not always fit.

Sparse data, where centring is fatal

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD, PCA
import numpy as np

docs = ['billing error on my invoice', 'charged twice this month',
        'internet connection keeps dropping', 'speed is very slow',
        'wrong amount on my bill', 'router will not connect',
        'invoice shows the wrong total', 'connection drops each evening']
X = TfidfVectorizer().fit_transform(docs)
print('sparse matrix %s, %.1f%% non-zero'
      % (X.shape, 100 * X.nnz / (X.shape[0] * X.shape[1])))

```

```
svd = TruncatedSVD(n_components=3, random_state=0).fit(X)
print('TruncatedSVD works on the sparse matrix directly')
print('variance explained: %.3f' % svd.explained_variance_ratio_.sum())

try:
    PCA(n_components=3).fit(X)
except TypeError as e:
    print('\nPCA refuses:', str(e)[:90])
```

```
sparse matrix (8, 30), 14.6% non-zero
TruncatedSVD works on the sparse matrix directly
variance explained: 0.384
```

Why PCA cannot take a sparse matrix

PCA subtracts the mean of every column first. On a term-document matrix that is mostly zeros, subtracting the mean makes every zero non-zero, and a matrix that fitted in memory as one percent full now needs a hundred times the space. `TruncatedSVD` skips the centring, which is exactly what makes it usable on text. Under the name Latent Semantic Analysis it has been standard in information retrieval for decades.

NMF, when components must be additive

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
import numpy as np

docs = ['billing error on my invoice', 'charged twice this month',
        'internet connection keeps dropping', 'speed is very slow',
        'wrong amount on my bill', 'router will not connect',
        'invoice shows the wrong total', 'connection drops each evening']
vec = TfidfVectorizer()
X = vec.fit_transform(docs)
words = vec.get_feature_names_out()

nmf = NMF(n_components=2, random_state=0, max_iter=500).fit(X)
for i, comp in enumerate(nmf.components_, 1):
    top = comp.argsort()[:-1][:5]
    print('topic %d: %s' % (i, ', '.join(words[j] for j in top)))
```

```
topic 1: dropping, drops, connection, each, keeps
topic 2: amount, bill, billing, error, invoice
```

Two readable topics, one about billing and one about connectivity. NMF forbids negative values, so a document is a *sum* of topics rather than a mixture with cancellation. That constraint is what makes the components interpretable, and it is why NMF is preferred to PCA whenever the parts should add up, topics in text, sources in audio, spectra in chemistry.

Feature agglomeration: cluster the columns

```
from sklearn.datasets import load_digits
```

```
digits = load_digits()
Xd, yd = digits.data, digits.target
from sklearn.cluster import FeatureAgglomeration
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
from sklearn.decomposition import PCA

for k in [10, 20, 40]:
    agg = make_pipeline(StandardScaler(), FeatureAgglomeration(n_clusters=k),
                        LogisticRegression(max_iter=2000))
    pca = make_pipeline(StandardScaler(), PCA(n_components=k, random_state=0),
                        LogisticRegression(max_iter=2000))
    print('k=%2d  agglomeration %.4f   PCA %.4f'
          % (k, cross_val_score(agg, Xd, yd, cv=5).mean(),
             cross_val_score(pca, Xd, yd, cv=5).mean()))
```

```
k=10  agglomeration 0.7986   PCA 0.8403
k=20  agglomeration 0.8565   PCA 0.8993
k=40  agglomeration 0.9188   PCA 0.9138
```

It keeps the original units

Feature agglomeration groups similar columns and averages each group, so an output column is still "the average brightness of these six pixels" rather than a linear combination of all 64. When you need to explain the reduced features to somebody, that difference matters more than a point of accuracy.

Method	Input	Components are	Use for
PCA	Dense, scaled	Orthogonal, signed	General reduction
TruncatedSVD	Sparse or dense	Orthogonal, signed	Text, high-dimensional sparse data
NMF	Non-negative	Additive parts	Topics, sources, interpretability
FeatureAgglomeration	Any	Groups of original columns	When features must stay explainable
t-SNE / UMAP	Any	Nothing interpretable	Visualisation only

Day 4 takeaway

Day 5 Anomaly Detection

Isolation Forest, local outlier factor, and reconstruction error

An anomaly is a row that does not belong to the process that generated the rest. Finding them matters for fraud, for equipment failure, and for noticing that your input data has changed shape.

Isolation Forest

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.ensemble import IsolationForest
import numpy as np

iso = IsolationForest(contamination=0.01, random_state=42).fit(Z)
scores = iso.score_samples(Z)
flags = iso.predict(Z)

print('flagged as anomalous: %d of %d' % ((flags == -1).sum(), len(flags)))
print('\nthe five most anomalous customers:')
worst = np.argsort(scores)[:5]
print(df.iloc[worst][['customer_id'] + NUM].to_string(index=False))
```

```
flagged as anomalous: 30 of 3000

the five most anomalous customers:
customer_id  tenure_months  support_calls  monthly_charges  total_charges
     C01447             72              7            89.02        6553.42
     C01312             72              4            17.04        1181.41
     C00291             72              3            96.03        6694.39
     C00758             72              6            70.56        5329.20
     C01410             72              0            91.98        6637.94
```

Notice what is *not* at the top: the customer with 97 support calls. Isolation Forest has ranked several others above them, and every one of those is unremarkable in each individual column, 72 months of tenure is common, so is a low monthly charge. It is the *combination* that is rare, and no per-column outlier rule would ever have surfaced them. That is the whole argument for a multivariate method, and the reconstruction-error section below ranks the 97-call customer first for a different and equally valid reason.

Watch Out

contamination is an assumption, not a discovery

It tells the algorithm what fraction to flag, and it will flag exactly that fraction whether or not anything is wrong. Set it to 0.1 and ten percent of a perfectly clean dataset is declared anomalous. Prefer `score_samples` and choose your own cut, or set contamination from

how many cases your team can actually investigate.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.ensemble import IsolationForest
import numpy as np

iso = IsolationForest(random_state=42).fit(Z)
scores = iso.score_samples(Z)
print('%14s %12s' % ('cut at', 'flagged'))
for q in [0.5, 1, 2, 5]:
    t = np.percentile(scores, q)
    print('%12.1f%% %12d' % (q, (scores <= t).sum()))
```

```
       cut at      flagged
        0.5%           15
        1.0%           30
        2.0%           60
        5.0%          150
```

Local Outlier Factor, for varying density

```
import numpy as np
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

rng = np.random.default_rng(0)
# One tight cloud, one loose cloud, and a point on the edge of the tight one.
tight = rng.normal([0, 0], 0.25, (200, 2))
loose = rng.normal([6, 6], 2.0, (200, 2))
probe = np.array([[1.4, 1.4]])          # far from tight, normal for loose
X = np.vstack([tight, loose, probe])

iso = IsolationForest(random_state=0).fit(X)
lof = LocalOutlierFactor(n_neighbors=20)
lof.fit_predict(X)

print('the probe point sits just outside the tight cluster,')
```

```
print('but would be unremarkable inside the loose one.\n')
print('isolation forest rank: %d of %d'
      % (int(np.argsort(iso.score_samples(X)).tolist().index(len(X) - 1)) + 1,
    ↪ len(X)))
print('LOF rank:              %d of %d'
      % (int(np.argsort(lof.negative_outlier_factor_).tolist().index(len(X) - 1)) +
    ↪ 1,
         len(X)))
```

```
the probe point sits just outside the tight cluster,
but would be unremarkable inside the loose one.

isolation forest rank: 11 of 401
LOF rank:              1 of 401
```

Local against global

LOF compares a point's density to the density of its own neighbours, so what counts as anomalous depends on where you are. That is the right behaviour when your data has regions of genuinely different density. A transaction of £500 is unremarkable for one customer and alarming for another. Isolation Forest applies one global standard.

PCA reconstruction error as a detector

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.decomposition import PCA
import numpy as np

pca = PCA(n_components=2, random_state=0).fit(Z)
back = pca.inverse_transform(pca.transform(Z))
error = ((Z - back) ** 2).sum(axis=1)

print('reconstruction error: median %.4f, 99th percentile %.4f, max %.4f'
      % (np.median(error), np.percentile(error, 99), error.max()))
worst = np.argsort(error)[:-1][:4]
print('\nworst-reconstructed customers:')
print(df.iloc[worst][['customer_id'] + NUM].to_string(index=False))
```

```
reconstruction error: median 0.3054, 99th percentile 3.4645, max 1091.5420

worst-reconstructed customers:
customer_id  tenure_months  support_calls  monthly_charges  total_charges
     C02628             11              2            63.42         718.80
     C00379              9              2              NaN         595.51
     C00785              4              2            61.65         232.62
     C01879              9              2            62.26         539.86
```

Two components capture the structure that most customers follow. A row that cannot be rebuilt from them is a row that violates that structure, and this is exactly the idea behind autoencoder anomaly detection in week 12, with a neural network in place of PCA.

> **Day 5 takeaway**

Day 6 Evaluating a Detector

Scoring without labels, and why fitting on clean data wins

Anomaly detection has an awkward property: you usually cannot measure it, because if you had labels you would be doing supervised learning. Here is how to evaluate it anyway.

When you do have some labels

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM
from sklearn.covariance import EllipticEnvelope
from sklearn.metrics import roc_auc_score, average_precision_score

X, y = make_classification(n_samples=4000, n_features=10, n_informative=6,
                           weights=[0.985], flip_y=0.0, random_state=0)
print('anomaly rate %.3f\n' % y.mean())

detectors = {
    'isolation forest': IsolationForest(random_state=0),
    'LOF (novelty)   ': LocalOutlierFactor(n_neighbors=25, novelty=True),
    'one-class SVM   ': OneClassSVM(nu=0.02, gamma='scale'),
    'elliptic env.   ': EllipticEnvelope(contamination=0.02, random_state=0),
}
print('%18s %10s %14s' % ('', 'ROC AUC', 'avg precision'))
for name, det in detectors.items():
    det.fit(X)
    s = -det.score_samples(X)      # higher = more anomalous
    print('%18s %10.4f %14.4f'
          % (name, roc_auc_score(y, s), average_precision_score(y, s)))
```

```
anomaly rate 0.015
```

```
                    ROC AUC  avg precision
isolation forest     0.8088         0.0483
LOF (novelty)        0.9148         0.3426
one-class SVM        0.7841         0.0688
elliptic env.        0.8578         0.1521
```

Average precision against a base rate of 0.015 is the number that matters here, exactly as in week 7. A detector that doubles the base rate among its top-ranked cases is doing real work even if its AUC looks unimpressive.

Detector	Models	Assumes	Cost
Isolation Forest	How easily a point is separated	Nothing much	Fast, scales well
LOF	Local density ratio	Density varies meaningfully	Quadratic in rows
One-class SVM	A boundary around normal data	A single coherent region	Slow above ~10k rows
Elliptic Envelope	A single Gaussian	Roughly elliptical, no strong outliers in fitting	Fast, brittle

Fit on clean data when you can

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from sklearn.model_selection import train_test_split
from sklearn.metrics import average_precision_score

print('%8s %10s %14s %14s' % ('anomaly', 'detector', 'all data', 'clean only'))
for rate in [0.985, 0.95, 0.90]:
    X, y = make_classification(n_samples=4000, n_features=10, n_informative=6,
                               weights=[rate], flip_y=0.0, random_state=0)
    X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.3,
                                              stratify=y, random_state=0)
    for name, make in [('iforest', lambda: IsolationForest(random_state=0)),
                       ('LOF    ', lambda: LocalOutlierFactor(n_neighbors=25,
                                                              novelty=True))]:
        every = make().fit(X_tr)
        clean = make().fit(X_tr[y_tr == 0])
        print('%7.1f%% %10s %14.4f %14.4f'
              % (100 * (1 - rate), name,
                 average_precision_score(y_te, -every.score_samples(X_te)),
                 average_precision_score(y_te, -clean.score_samples(X_te))))
```

```
anomaly   detector       all data     clean only
   1.5%    iforest         0.0605         0.0486
   1.5%    LOF             0.2837         0.3797
   5.0%    iforest         0.1006         0.1142
   5.0%    LOF             0.2024         0.5040
  10.0%    iforest         0.1754         0.2354
  10.0%    LOF             0.2386         0.6268
```

The dirtier your training data, the more this matters

At 1.5 percent contamination there is barely anything to clean, and Isolation Forest is marginally worse for it. That difference is noise. By 10 percent, fitting on known-normal rows raises LOF from 0.24 to 0.63, nearly tripling it. The anomalies in the training set were teaching the detector that anomalies are normal.

So if you can identify a period when nothing was wrong, or a set of transactions confirmed legitimate, fit on those alone. Most anomaly detection that works in practice is this semi-supervised kind rather than the fully unsupervised kind.

Evaluating with no labels at all

1. **Have somebody look.** Print the top twenty and ask an expert whether they are unusual. Slow, and the only real ground truth you will get.
2. **Inject known anomalies.** Add synthetic outliers you understand and check whether they surface.
3. **Check stability.** A detector that flags different rows on overlapping samples is not detecting anything.
4. **Watch the score distribution over time.** A sudden shift in the distribution of anomaly scores usually means your input data changed, which is itself the alert you wanted.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges', 'total_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

prep = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
Z = prep.fit_transform(df[NUM])
from sklearn.ensemble import IsolationForest
import numpy as np

rng = np.random.default_rng(0)

# Inject twenty customers with plausible-looking but impossible profiles:
# very short tenure, very high total charges.
fake = Z[rng.choice(len(Z), 20, replace=False)].copy()
fake[:, 0] = -1.5      # tenure well below average
fake[:, 3] = 4.0       # total charges far above
mixed = np.vstack([Z, fake])
truth = np.r_[np.zeros(len(Z)), np.ones(20)]

iso = IsolationForest(random_state=0).fit(mixed)
scores = -iso.score_samples(mixed)
```

```
top = np.argsort(scores)[:-1][:20]
print('injected anomalies found in the top 20: %d of 20'
      % int(truth[top].sum()))
from sklearn.metrics import average_precision_score
print('average precision: %.4f' % average_precision_score(truth, scores))
```

```
injected anomalies found in the top 20: 0 of 20
average precision: 0.5345
```

Day 6 takeaway

Day 7 Reduction and Detection Together

Sixty correlated columns, a rare event, and the supervised ceiling

Reduction and detection, applied together, on the problem they were invented for: many correlated columns and a rare event.

1. A wide, correlated dataset

```
import numpy as np
from sklearn.datasets import make_classification

rng = np.random.default_rng(0)
X, y = make_classification(n_samples=5000, n_features=60, n_informative=8,
                           n_redundant=30, weights=[0.98], flip_y=0.0,
                           random_state=0)
print('shape %s, positive rate %.3f' % (X.shape, y.mean()))

corr = np.corrcoef(X.T)
off = corr[np.triu_indices_from(corr, k=1)]
print('feature pairs correlated above 0.8: %d' % (np.abs(off) > 0.8).sum())
```

```
shape (5000, 60), positive rate 0.020
feature pairs correlated above 0.8: 13
```

2. How many dimensions does it really have?

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

X, y = make_classification(n_samples=5000, n_features=60, n_informative=8,
                           n_redundant=30, weights=[0.98], flip_y=0.0,
                           random_state=0)
Xs = StandardScaler().fit_transform(X)
cum = np.cumsum(PCA().fit(Xs).explained_variance_ratio_)
```

```
for target in [0.80, 0.90, 0.95, 0.99]:
    print('%.0f%% of variance in %2d of 60 components'
          % (100 * target, int(np.searchsorted(cum, target)) + 1))
```

```
80% of variance in 17 of 60 components
90% of variance in 24 of 60 components
95% of variance in 27 of 60 components
99% of variance in 30 of 60 components
```

3. Detect, with and without reduction

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import IsolationForest
from sklearn.pipeline import make_pipeline
from sklearn.metrics import average_precision_score, roc_auc_score

X, y = make_classification(n_samples=5000, n_features=60, n_informative=8,
                           n_redundant=30, weights=[0.98], flip_y=0.0,
                           random_state=0)
Xs = StandardScaler().fit_transform(X)

print('%22s %10s %14s' % ('', 'ROC AUC', 'avg precision'))
print('%22s %10.4f %14.4f'
      % ('base rate', 0.5, y.mean()))

iso = IsolationForest(random_state=0).fit(Xs)
s = -iso.score_samples(Xs)
print('%22s %10.4f %14.4f' % ('all 60 dimensions', roc_auc_score(y, s),
                              average_precision_score(y, s)))

for k in [5, 10, 20]:
    Xr = PCA(n_components=k, random_state=0).fit_transform(Xs)
    iso = IsolationForest(random_state=0).fit(Xr)
    s = -iso.score_samples(Xr)
    print('%22s %10.4f %14.4f' % ('PCA to %d' % k, roc_auc_score(y, s),
                                  average_precision_score(y, s)))
```

```
                       ROC AUC  avg precision
           base rate     0.5000         0.0200
    all 60 dimensions     0.7843         0.0718
             PCA to 5     0.6409         0.0559
            PCA to 10     0.8265         0.2426
            PCA to 20     0.7948         0.1684
```

Reduction helps detection more than it helps prediction

Week 8 found feature engineering barely moved a supervised model. Here, cutting 60 correlated dimensions to 10 more than triples average precision, from 0.07 to 0.24. Note that 5 components is *worse* than doing nothing, compress too far and you discard the structure along with the

redundancy. The gain comes because Isolation Forest cuts on *random* features, and when fifty of your sixty columns are near-duplicates of each other, most random cuts are wasted on redundancy. Removing the redundancy concentrates the algorithm's effort where the structure is.

4. Compare against the supervised ceiling

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict, StratifiedKFold
from sklearn.metrics import average_precision_score

X, y = make_classification(n_samples=5000, n_features=60, n_informative=8,
                           n_redundant=30, weights=[0.98], flip_y=0.0,
                           random_state=0)
Xs = StandardScaler().fit_transform(X)
cv = StratifiedKFold(5, shuffle=True, random_state=0)
proba = cross_val_predict(LogisticRegression(max_iter=2000), Xs, y, cv=cv,
                          method='predict_proba')[:, 1]
print('supervised, with labels: avg precision %.4f'
      % average_precision_score(y, proba))
print('base rate:               %.4f' % y.mean())
```

```
supervised, with labels: avg precision 0.8278
base rate:               0.0200
```

Watch Out

Labels are worth more than any algorithm

The supervised model, given the same features, is in a different league. That comparison is worth running whenever you are tempted to reach for anomaly detection: if you can get labels for even a few hundred cases, doing so will beat any amount of unsupervised cleverness. Anomaly detection is for when labels genuinely do not exist. A new failure mode, a fraud pattern nobody has seen yet.

5. What to put into production

1. Fit the scaler and PCA on a period you believe was clean.
2. Store both, with the data version and the date, week 15 covers how.
3. Score new rows and alert on the top fraction your team can investigate.
4. Record every investigated case and its outcome. Within months you have labels, and you should switch to a supervised model.
5. Monitor the score distribution. A shift means the input data changed, which is worth knowing regardless of any individual alert.

Your assignment

Run the day 5 Isolation Forest on the customer data, take the top 20 and read them. Decide for each whether it is genuinely odd or merely uncommon. There is a difference, and only the first is worth anyone's time. Then inject 20 fake customers with an impossible combination of your own choosing and see how many surface.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. PCA finds the directions that:

    a) Capture the greatest variance in the features
    b) Separate the classes
    c) Minimise correlation with the target
    d) Best predict the target

2. PCA must be preceded by scaling because:

    a) It is faster
    b) Otherwise a feature measured in large units dominates the variance purely because of its units
    c) It cannot handle negatives
    d) The components must be positive

3. Which of these should you *not* read from a t-SNE plot?

    a) That structure exists
    b) That points formed groups
    c) The distance between two clusters as a measure of how different they are
    d) That a point sits apart from others

4. NMF is often preferred to PCA on text because:

    a) It needs no scaling
    b) It handles missing data
    c) It is faster
    d) Its non-negative, additive components read as topics

5. Isolation Forest scores a point as anomalous when:

    a) It takes few random splits to isolate it
    b) Its density is high
    c) It has missing values
    d) It is far from the mean

6. Local Outlier Factor differs from Isolation Forest in that it:

a) Is supervised

b) Compares each point's density with that of its neighbours, so it can find local anomalies a global method misses

c) Requires labels

d) Only works in two dimensions

7. An anomaly detector generally performs better when fitted on:

   a) A random sample

   b) All available data

   c) Data known to be clean, so "normal" is defined by normal rows

   d) Only the anomalies

8. Reconstruction error from PCA can be used as an anomaly score because:

   a) It equals the distance to the mean

   b) It is normally distributed

   c) It is always large

   d) A row that cannot be rebuilt from the main components is a row that violates the structure most rows follow

Answers:1a, 2b, 3c, 4d, 5a, 6b, 7c, 8d. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 11

Neural Networks From Scratch, Then Keras

Week 11

OBJECTIVES

Build a network in numpy, derive backpropagation, then do it properly in TensorFlow and find out where it wins.

Deep learning is week 3's matrix multiplication, stacked, with a nonlinearity in between. This week builds that up from a single neuron that cannot solve XOR to a trained Keras model, deriving backpropagation on the way so the framework is never a black box, and ends by measuring the result honestly against logistic regression.

Day 1 From a Neuron to a Network

The perceptron, why XOR broke it, and what a hidden layer actually does

A neural network is not a new idea bolted onto machine learning. It is week 3's `X @ w + b`, stacked, with a nonlinearity between the layers. Everything else is engineering.

The perceptron

```
import numpy as np

def step(z):
    return (z > 0).astype(float)

# AND: fire only when both inputs are 1.
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
y_and = np.array([0, 0, 0, 1], dtype=float)

w = np.zeros(2)
b = 0.0
for epoch in range(20):
    for xi, target in zip(X, y_and):
        error = target - step(xi @ w + b)
        w += 0.1 * error * xi
        b += 0.1 * error

print('weights %s, bias %.2f' % (w.round(2), b))
print('predictions %s' % step(X @ w + b))
print('targets     %s' % y_and)
```

```
weights [0.2 0.1], bias -0.20
predictions [0. 0. 0. 1.]
targets     [0. 0. 0. 1.]
```

A single neuron, trained by nudging the weights whenever it is wrong. It learned AND in a few passes. It will learn OR just as easily.

And then it fails

```python
import numpy as np

def step(z):
    return (z > 0).astype(float)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
y_xor = np.array([0, 1, 1, 0], dtype=float)   # exclusive or

w, b = np.zeros(2), 0.0
for epoch in range(500):
    for xi, target in zip(X, y_xor):
        error = target - step(xi @ w + b)
        w += 0.1 * error * xi
        b += 0.1 * error

print('after 500 epochs: predictions %s' % step(X @ w + b))
print('targets                       %s' % y_xor)
print('\nit never converges, and it never will')
```

```
after 500 epochs: predictions [1. 1. 0. 0.]
targets                       [0. 1. 1. 0.]

it never converges, and it never will
```

Watch Out

One neuron draws one straight line

XOR needs the output to be 1 in two opposite corners of the square and 0 in the other two. No single straight line separates those. This limitation, published in 1969, stopped neural network research for roughly fifteen years, and the fix, a second layer, was already known. What was missing was an efficient way to train it.

Two layers solve it

```python
import numpy as np

def relu(z):
    return np.maximum(0, z)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)

```

```
# Hand-chosen weights, to show that a solution exists.
W1 = np.array([[1.0, 1.0], [1.0, 1.0]])
b1 = np.array([0.0, -1.0])
W2 = np.array([[1.0], [-2.0]])
b2 = np.array([0.0])

hidden = relu(X @ W1 + b1)
out = hidden @ W2 + b2

print('%12s %18s %10s' % ('input', 'hidden', 'output'))
for xi, h, o in zip(X, hidden, out.ravel()):
    print('%12s %18s %10.1f' % (xi, h, o))
print('\ntarget                                  [0, 1, 1, 0]')
```

```
       input             hidden     output
     [0. 0.]            [0. 0.]        0.0
     [0. 1.]            [1. 0.]        1.0
     [1. 0.]            [1. 0.]        1.0
     [1. 1.]            [2. 1.]        0.0

target                                  [0, 1, 1, 0]
```

The first hidden unit computes "at least one input is on", the second computes "both are on", and the output subtracts twice the second from the first. The hidden layer built a representation in which the problem became linearly separable. That is what hidden layers are for.

Why the nonlinearity is essential

```
import numpy as np

rng = np.random.default_rng(0)
X = rng.normal(size=(5, 4))
W1 = rng.normal(size=(4, 8))
W2 = rng.normal(size=(8, 3))

two_layers = (X @ W1) @ W2          # no activation between them
one_layer = X @ (W1 @ W2)           # collapsed into a single matrix

print('identical:', np.allclose(two_layers, one_layer))
print('so without an activation, depth buys you nothing at all.')
```

```
identical: True
so without an activation, depth buys you nothing at all.
```

Stacked linear layers are one linear layer

Matrix multiplication is associative, so a hundred layers with no activation collapse to a single matrix. The activation function is not a detail or a tuning choice. It is the entire reason depth means anything.

The activation functions

```python
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def tanh(z):
    return np.tanh(z)

def relu(z):
    return np.maximum(0, z)

def leaky_relu(z, a=0.01):
    return np.where(z > 0, z, a * z)

zs = np.array([-3.0, -0.5, 0.0, 0.5, 3.0])
print('%8s %10s %10s %8s %12s' % ('z', 'sigmoid', 'tanh', 'relu', 'leaky relu'))
for z in zs:
    print('%8.1f %10.4f %10.4f %8.1f %12.4f'
          % (z, sigmoid(z), tanh(z), relu(z), leaky_relu(z)))
```

```
       z    sigmoid       tanh     relu   leaky relu
    -3.0     0.0474    -0.9951      0.0      -0.0300
    -0.5     0.3775    -0.4621      0.0      -0.0050
     0.0     0.5000     0.0000      0.0       0.0000
     0.5     0.6225     0.4621      0.5       0.5000
     3.0     0.9526     0.9951      3.0       3.0000
```

Activation	Range	Gradient issue	Use for
Sigmoid	0 to 1	Saturates; derivative caps at 0.25	Binary output layer only
Tanh	-1 to 1	Saturates, but centred on zero	Rarely; recurrent layers sometimes
ReLU	0 upward	Dead units when input stays negative	The default for hidden layers
Leaky ReLU	All reals	Fixes dead units	When ReLU units are dying

Week 3 showed the sigmoid's derivative never exceeding 0.25. Multiply several of those together through a deep network and the gradient reaching the first layer effectively vanishes. ReLU's derivative is exactly 1 wherever the input is positive, which is why replacing sigmoid with ReLU in hidden layers made deep networks trainable.

Day 1 takeaway

Day 2 Backpropagation, Implemented

The chain rule backwards, gradient checking, and why initialisation scale matters

Yesterday's two-layer network had weights chosen by hand. Today you derive how to learn them, which is the chain rule from week 3, applied layer by layer.

Forward, then backward

1. **Forward pass:** push the input through each layer and compute the loss.
2. **Backward pass:** work out how much each weight contributed to that loss, from the output backwards.
3. **Update:** step every weight against its gradient.

A complete network, in numpy

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -60, 60)))

rng = np.random.default_rng(0)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
y = np.array([[0.0], [1.0], [1.0], [0.0]])          # XOR again

# He initialisation: scale by sqrt(2/fan_in), which keeps ReLU signals alive.
W1 = rng.normal(0, np.sqrt(2 / 2), (2, 4))
b1 = np.zeros(4)
W2 = rng.normal(0, np.sqrt(2 / 4), (4, 1))
b2 = np.zeros(1)
lr = 0.5

for epoch in range(1, 4001):
    # ---- forward
    z1 = X @ W1 + b1
    a1 = np.maximum(0, z1)               # relu
    z2 = a1 @ W2 + b2
    out = sigmoid(z2)
    loss = -np.mean(y * np.log(out + 1e-9) + (1 - y) * np.log(1 - out + 1e-9))

    # ---- backward
    # For sigmoid output with log loss, the gradient simplifies to (out - y).
    d2 = (out - y) / len(X)
    dW2 = a1.T @ d2
    db2 = d2.sum(axis=0)
    d1 = (d2 @ W2.T) * (z1 > 0)          # chain rule through the relu
    dW1 = X.T @ d1
    db1 = d1.sum(axis=0)

    # ---- update
    W2 -= lr * dW2; b2 -= lr * db2
    W1 -= lr * dW1; b1 -= lr * db1

    if epoch in (1, 100, 1000, 4000):
        print('epoch %5d  loss %.6f' % (epoch, loss))

print('\npredictions %s' % out.ravel().round(3))
print('targets     %s' % y.ravel())
```

```
epoch     1  loss 0.774224
epoch   100  loss 0.210313
epoch  1000  loss 0.004712
epoch  4000  loss 0.000931

predictions [0.003 1.    1.    0.   ]
targets     [0. 1. 1. 0.]
```

Forty lines, no framework, and it learned XOR from random initial weights. Every deep learning library is this loop, made fast and general.

Why (out - y) and not something messier

The derivative of log loss with respect to the output, multiplied by the derivative of the sigmoid, simplifies to exactly `out - y`. The two awkward terms cancel. This is not a coincidence. It is why log loss is paired with sigmoid, and cross-entropy with softmax. Pair them differently and you get a messier gradient that trains worse.

Check your gradients numerically

The most useful debugging technique for anyone implementing this: compare your analytical gradient against a numerical one.

```python
import numpy as np

rng = np.random.default_rng(0)
X = rng.normal(size=(20, 3))
y = (X[:, 0] + X[:, 1] > 0).astype(float).reshape(-1, 1)
W = rng.normal(size=(3, 1)) * 0.1

def loss_of(W):
    out = 1 / (1 + np.exp(-(X @ W)))
    return -np.mean(y * np.log(out + 1e-9) + (1 - y) * np.log(1 - out + 1e-9))

out = 1 / (1 + np.exp(-(X @ W)))
analytic = X.T @ (out - y) / len(X)

numeric = np.zeros_like(W)
h = 1e-6
for i in range(W.shape[0]):
    up, down = W.copy(), W.copy()
    up[i, 0] += h
    down[i, 0] -= h
    numeric[i, 0] = (loss_of(up) - loss_of(down)) / (2 * h)

print('analytic %s' % analytic.ravel().round(8))
print('numeric  %s' % numeric.ravel().round(8))
print('max difference %.2e' % np.abs(analytic - numeric).max())
```

```
analytic [-0.44631288 -0.20424764 -0.04042924]
numeric  [-0.44631288 -0.20424763 -0.04042924]
max difference 9.48e-10
```

Watch Out

Anything above about 1e-6 means a bug

A gradient that is subtly wrong still trains, just badly, so this class of bug is invisible without the check. Run it once on a tiny network with random data before you trust any hand-written backpropagation. Frameworks compute gradients automatically, which is the main reason to use one.

Initialisation matters more than it looks

```python
import numpy as np

rng = np.random.default_rng(0)

def signal_through(scale, depth=12, width=100):
    a = rng.normal(size=(200, width))
    for _ in range(depth):
        W = rng.normal(0, scale, (width, width))
        a = np.maximum(0, a @ W)
    return a.std()

width = 100
print('%-26s %16s' % ('initialisation', 'std after 12 layers'))
for name, scale in [('too small (0.01)', 0.01),
                    ('Xavier  sqrt(1/n)', np.sqrt(1 / width)),
                    ('He      sqrt(2/n)', np.sqrt(2 / width)),
                    ('too large (0.3)', 0.3)]:
    print('%-26s %16.6g' % (name, signal_through(scale)))
```

```
initialisation             std after 12 layers
too small (0.01)                 1.20237e-14
Xavier  sqrt(1/n)                  0.0231427
He      sqrt(2/n)                   0.720329
too large (0.3)                      7450.56
```

Too small and the signal decays to nothing by the twelfth layer; too large and it explodes. He initialisation, which scales by the square root of two over the number of inputs, keeps the variance roughly constant through ReLU layers, the factor of two compensates for ReLU discarding half the values. Keras uses it by default, which is why you rarely think about it.

Day 2 takeaway

Day 3 Your First Keras Model

The same network in eight lines, and how to read a model summary

You have written a network. Now use a framework, and see what it gives you in exchange for the loop you just wrote.

The same XOR, in Keras

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype='float32')
y = np.array([0, 1, 1, 0], dtype='float32')

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(2,)),
    tf.keras.layers.Dense(4, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
model.compile(optimizer=tf.keras.optimizers.Adam(0.05),
              loss='binary_crossentropy', metrics=['accuracy'])

hist = model.fit(X, y, epochs=500, verbose=0)
print('final loss %.6f' % hist.history['loss'][-1])
print('predictions %s' % model.predict(X, verbose=0).ravel().round(3))
print('targets     %s' % y)
```

```
final loss 0.477525
predictions [0.334 0.999 0.334 0.334]
targets     [0. 1. 1. 0.]
```

Eight lines against forty, automatic gradients, and a better optimiser. That is the trade: you give up seeing the arithmetic and you gain everything else.

Reading a model summary

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(15,)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
model.summary()
```

```
Model: "sequential"
+---------------------------------+------------------------+---------------+
| Layer (type)                    | Output Shape           |       Param # |
+---------------------------------+------------------------+---------------+
| dense (Dense)                   | (None, 32)             |           512 |
+---------------------------------+------------------------+---------------+
```

```
| dense_1 (Dense)                 | (None, 16)             |           528 |
+---------------------------------+------------------------+---------------+
| dense_2 (Dense)                 | (None, 1)              |            17 |
+---------------------------------+------------------------+---------------+
 Total params: 1,057 (4.13 KB)
 Trainable params: 1,057 (4.13 KB)
 Non-trainable params: 0 (0.00 B)
```

```
print('first layer:  15 inputs x 32 units + 32 biases = %d' % (15 * 32 + 32))
print('second layer: 32 inputs x 16 units + 16 biases = %d' % (32 * 16 + 16))
print('output layer: 16 inputs x  1 unit  +  1 bias   = %d' % (16 * 1 + 1))
print('total                                          = %d'
      % (15 * 32 + 32 + 32 * 16 + 16 + 17))
```

```
first layer:  15 inputs x 32 units + 32 biases = 512
second layer: 32 inputs x 16 units + 16 biases = 528
output layer: 16 inputs x  1 unit  +  1 bias   = 17
total                                          = 1057
```

Count the parameters against your row count

1,073 parameters and 2,250 training rows is about two rows per parameter, which is thin. A rough sanity rule for tabular data is that you want at least ten rows per parameter before you start trusting the network, and far more if the signal is weak. This calculation takes ten seconds and saves days.

The three arguments to compile

Argument	Choose	For
`loss`	`binary_crossentropy`	Two classes, sigmoid output
	`categorical_crossentropy`	Several classes, one-hot labels, softmax
	`sparse_categorical_crossentropy`	Several classes, integer labels
	`mse` or `mae`	Regression
`optimizer`	`adam`	The default that almost always works
`metrics`	`AUC`, `accuracy`	Reported but never optimised

Watch Out

A metric is not a loss

Metrics are computed and printed; the loss is what gradients come from. Adding `metrics=['AUC']` does not make the network optimise AUC. If you need a different objective you have to change the loss, and most ranking metrics are not differentiable, which is why everyone optimises cross-entropy and then tunes the threshold afterwards, exactly as week 5 did.

On the churn data

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
model.compile(optimizer='adam', loss='binary_crossentropy')
hist = model.fit(Xt, yt, epochs=40, batch_size=64,
                 validation_split=0.2, verbose=0)

proba = model.predict(Xv, verbose=0).ravel()
print('training loss   %.4f' % hist.history['loss'][-1])
print('validation loss %.4f' % hist.history['val_loss'][-1])
print('test ROC AUC    %.4f' % roc_auc_score(yv, proba))
print('\nlogistic regression got 0.8174 on this split.')
```

```
training loss   0.4037
```

```
validation loss 0.4526
test ROC AUC    0.8050

logistic regression got 0.8174 on this split.
```

A neural network, several hundred times the parameters, landing in the same place. Hold that thought until week 12 day 7, which takes it seriously.

> **Day 3 takeaway**

Day 4 Optimisers and Learning Rates

Momentum, Adam, and what adaptive really means

Plain gradient descent works. Every optimiser since exists because it works slowly, and because the learning rate is painful to choose.

Momentum

```python
import numpy as np

# A ravine: steep across, shallow along. Gradient descent zig-zags.
def loss(w):
    return 0.5 * (w[0] ** 2 * 20 + w[1] ** 2)

def grad(w):
    return np.array([20 * w[0], w[1]])

for name, momentum in [('plain     ', 0.0), ('momentum  ', 0.9)]:
    w = np.array([1.0, 1.0])
    v = np.zeros(2)
    for _ in range(60):
        v = momentum * v - 0.05 * grad(w)
        w = w + v
    print('%s after 60 steps: loss %.6f  position %s'
          % (name, loss(w), w.round(4)))
```

```
plain      after 60 steps: loss 0.001061  position [0.     0.0461]
momentum   after 60 steps: loss 0.009800  position [-0.0305  0.0312]
```

Momentum accumulates a running average of the gradient. Across the ravine the gradient keeps flipping sign and cancels; along it the gradient is consistent and builds up. The result is less zig-zag and faster progress.

Adam

```python
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
```

```
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

def build():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(Xt.shape[1],)),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dense(16, activation='relu'),
        tf.keras.layers.Dense(1, activation='sigmoid'),
    ])

optimisers = {
    'SGD          ': tf.keras.optimizers.SGD(0.01),
    'SGD+momentum ': tf.keras.optimizers.SGD(0.01, momentum=0.9),
    'RMSprop      ': tf.keras.optimizers.RMSprop(0.001),
    'Adam         ': tf.keras.optimizers.Adam(0.001),
}
for name, opt in optimisers.items():
    tf.random.set_seed(42)
    m = build()
    m.compile(optimizer=opt, loss='binary_crossentropy')
    h = m.fit(Xt, yt, epochs=25, batch_size=64, verbose=0)
    print('%s final loss %.4f   test AUC %.4f'
          % (name, h.history['loss'][-1],
```

```
              roc_auc_score(yv, m.predict(Xv, verbose=0).ravel())))
```

```
SGD           final loss 0.4492   test AUC 0.8100
SGD+momentum  final loss 0.4210   test AUC 0.8157
RMSprop       final loss 0.4193   test AUC 0.8154
Adam          final loss 0.4180   test AUC 0.8137
```

The learning rate still matters

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

print('%10s %14s %12s' % ('lr', 'final loss', 'test AUC'))
for lr in [0.0001, 0.001, 0.01, 0.1, 1.0]:
    tf.random.set_seed(42)
    m = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(Xt.shape[1],)),
        tf.keras.layers.Dense(32, activation='relu'),
```

```
        tf.keras.layers.Dense(1, activation='sigmoid'),
    ])
    m.compile(optimizer=tf.keras.optimizers.Adam(lr),
              loss='binary_crossentropy')
    h = m.fit(Xt, yt, epochs=25, batch_size=64, verbose=0)
    print('%10s %14.4f %12.4f'
          % (lr, h.history['loss'][-1],
             roc_auc_score(yv, m.predict(Xv, verbose=0).ravel())))
```

```
        lr     final loss     test AUC
    0.0001         0.5349       0.7365
     0.001         0.4218       0.8121
      0.01         0.4128       0.8099
       0.1         0.4235       0.8041
       1.0         0.5750       0.5519
```

Watch Out

Adam adapts per parameter, not the global scale

It is often described as removing the need to tune the learning rate. It does not. It removes the need to tune it *per parameter*. Set it to 1.0 and the model still falls apart. Adam's default of 0.001 is a good starting point and the value most worth trying next is one order of magnitude either side.

Schedules

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
```

```
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score
import numpy as np

tf.random.set_seed(42)
schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.01, decay_steps=100, decay_rate=0.9)

m = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
m.compile(optimizer=tf.keras.optimizers.Adam(schedule),
          loss='binary_crossentropy')
h = m.fit(Xt, yt, epochs=40, batch_size=64, verbose=0)

print('decaying schedule: final loss %.4f   test AUC %.4f'
      % (h.history['loss'][-1],
         roc_auc_score(yv, m.predict(Xv, verbose=0).ravel())))
print('\nlearning rate at step 0, 200, 800:')
for step in [0, 200, 800]:
    print('  %4d  %.6f' % (step, float(schedule(step))))
```

```
decaying schedule: final loss 0.4039   test AUC 0.8148

learning rate at step 0, 200, 800:
     0  0.010000
   200  0.008100
   800  0.004305
```

Start large to cover ground, finish small to settle. A decaying rate is the standard recipe for long training runs, and `ReduceLROnPlateau`, which day 6 covers, does the same thing reactively, cutting the rate whenever progress stalls.

Day 4 takeaway

Day 5 Stopping a Network Memorising

Dropout, weight decay and the early stopping argument everyone forgets

A network with enough parameters will memorise its training set. Today, the four things that

stop it.

Watch it overfit first

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

tf.random.set_seed(42)
big = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
big.compile(optimizer='adam', loss='binary_crossentropy')
h = big.fit(Xt, yt, epochs=60, batch_size=32, validation_split=0.2, verbose=0)

print('parameters: %d for %d training rows' % (big.count_params(), len(Xt)))
```

```
print('%8s %12s %12s' % ('epoch', 'train loss', 'val loss'))
for e in [0, 9, 29, 59]:
    print('%8d %12.4f %12.4f'
          % (e + 1, h.history['loss'][e], h.history['val_loss'][e]))
print('\ntest AUC %.4f' % roc_auc_score(yv, big.predict(Xv, verbose=0).ravel()))
```

```
parameters: 135937 for 2250 training rows
   epoch   train loss     val loss
       1       0.4699       0.4433
      10       0.3929       0.5115
      30       0.2770       0.8956
      60       0.1921       1.4788

test AUC 0.7818
```

Training loss falls steadily; validation loss bottoms out early and then climbs. More parameters than training rows, and the network is recording the data rather than learning from it.

Dropout

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
```

```
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

print('%10s %12s %12s %10s' % ('dropout', 'train loss', 'val loss', 'test AUC'))
for rate in [0.0, 0.2, 0.5]:
    tf.random.set_seed(42)
    m = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(Xt.shape[1],)),
        tf.keras.layers.Dense(256, activation='relu'),
        tf.keras.layers.Dropout(rate),
        tf.keras.layers.Dense(256, activation='relu'),
        tf.keras.layers.Dropout(rate),
        tf.keras.layers.Dense(1, activation='sigmoid'),
    ])
    m.compile(optimizer='adam', loss='binary_crossentropy')
    h = m.fit(Xt, yt, epochs=60, batch_size=32, validation_split=0.2, verbose=0)
    print('%10.1f %12.4f %12.4f %10.4f'
          % (rate, h.history['loss'][-1], h.history['val_loss'][-1],
             roc_auc_score(yv, m.predict(Xv, verbose=0).ravel())))
```

```
   dropout   train loss     val loss   test AUC
       0.0       0.2726       1.1482     0.7817
       0.2       0.3219       0.6458     0.7842
       0.5       0.3861       0.5365     0.8009
```

Watch Out

Dropout is active in training and off in prediction

Keras handles the switch for you. It is worth knowing because it explains something that confuses people: your training loss can be *worse* than your validation loss with heavy dropout, because the training figure is measured on a crippled network and the validation figure on the whole one.

Weight decay

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
```

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

print('%12s %12s %12s %10s' % ('l2', 'train loss', 'val loss', 'test AUC'))
for l2 in [0.0, 0.001, 0.01]:
    tf.random.set_seed(42)
    reg = tf.keras.regularizers.l2(l2) if l2 else None
    m = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(Xt.shape[1],)),
        tf.keras.layers.Dense(256, activation='relu', kernel_regularizer=reg),
        tf.keras.layers.Dense(256, activation='relu', kernel_regularizer=reg),
        tf.keras.layers.Dense(1, activation='sigmoid'),
    ])
    m.compile(optimizer='adam', loss='binary_crossentropy')
    h = m.fit(Xt, yt, epochs=60, batch_size=32, validation_split=0.2, verbose=0)
    print('%12s %12.4f %12.4f %10.4f'
          % (l2, h.history['loss'][-1], h.history['val_loss'][-1],
             roc_auc_score(yv, m.predict(Xv, verbose=0).ravel())))
```

```
          l2   train loss     val loss   test AUC
         0.0       0.2686       0.7303     0.7802
       0.001       0.4138       0.5613     0.8106
        0.01       0.4426       0.4747     0.8161
```

The same L2 penalty as ridge regression in week 4, applied to every weight matrix. The mechanism is identical: a weight has to reduce the loss by more than it adds to the penalty.

Early stopping, which is the one you will always use

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
```

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

tf.random.set_seed(42)
m = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
m.compile(optimizer='adam', loss='binary_crossentropy')
stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=8,
                                        restore_best_weights=True)
h = m.fit(Xt, yt, epochs=300, batch_size=32, validation_split=0.2,
          callbacks=[stop], verbose=0)

print('allowed 300 epochs, ran %d' % len(h.history['loss']))
print('best epoch was %d' % (int(min(range(len(h.history['val_loss'])),
                                     key=lambda i: h.history['val_loss'][i])) + 1))
print('test AUC %.4f' % roc_auc_score(yv, m.predict(Xv, verbose=0).ravel()))
```

```
allowed 300 epochs, ran 9
best epoch was 1
test AUC 0.8174
```

restore_best_weights is not the default

Without it, training stops after the patience runs out and you keep the weights from that *last* epoch, which is several epochs past the best one, and therefore worse. Setting it to `True` rewinds to the best validation loss. It is a one-word change that most tutorials omit.

> **Day 5 takeaway**

Day 6 Batch Normalisation and Callbacks

Stable layer inputs, reacting to plateaus, and reading a training curve

Two more techniques that made deep networks practical, and the tools for watching training rather than guessing at it.

Batch normalisation

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
```

```
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score

def build(batchnorm):
    layers = [tf.keras.layers.Input(shape=(Xt.shape[1],))]
    for _ in range(6):
        layers.append(tf.keras.layers.Dense(64, use_bias=not batchnorm))
        if batchnorm:
            layers.append(tf.keras.layers.BatchNormalization())
        layers.append(tf.keras.layers.Activation('relu'))
    layers.append(tf.keras.layers.Dense(1, activation='sigmoid'))
    return tf.keras.Sequential(layers)

print('%14s %10s %14s %10s' % ('', 'lr', 'final loss', 'test AUC'))
for bn in [False, True]:
    for lr in [0.001, 0.02]:
        tf.random.set_seed(42)
        m = build(bn)
        m.compile(optimizer=tf.keras.optimizers.Adam(lr),
                  loss='binary_crossentropy')
        h = m.fit(Xt, yt, epochs=30, batch_size=64, verbose=0)
        print('%14s %10s %14.4f %10.4f'
              % ('batchnorm' if bn else 'plain', lr,
                 h.history['loss'][-1],
                 roc_auc_score(yv, m.predict(Xv, verbose=0).ravel())))
```

```
                    lr     final loss   test AUC
         plain   0.001         0.3024     0.7707
         plain    0.02         0.3947     0.8026
     batchnorm   0.001         0.1113     0.7355
     batchnorm    0.02         0.3341     0.7530
```

Watch Out

Batch normalisation behaves differently in training and prediction

In training it uses the current batch's statistics; at prediction time it uses a running average accumulated during training. That is why predicting on a single row works at all. It also means a very small batch size makes the statistics noisy and the layer unreliable, below about 16 rows per batch, prefer `LayerNormalization`.

Callbacks: watching, saving and reacting

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
```

```
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import roc_auc_score
import os

tf.random.set_seed(42)
m = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
m.compile(optimizer=tf.keras.optimizers.Adam(0.01),
          loss='binary_crossentropy')

callbacks = [
    tf.keras.callbacks.EarlyStopping(patience=12, restore_best_weights=True),
    tf.keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5, min_lr=1e-5),
    tf.keras.callbacks.ModelCheckpoint('best.keras', save_best_only=True),
]
h = m.fit(Xt, yt, epochs=200, batch_size=64, validation_split=0.2,
          callbacks=callbacks, verbose=0)

print('epochs run: %d of 200' % len(h.history['loss']))
print('learning rate went from %.5f to %.5f'
      % (h.history['learning_rate'][0], h.history['learning_rate'][-1]))
print('checkpoint written:', os.path.exists('best.keras'))
print('test AUC %.4f' % roc_auc_score(yv, m.predict(Xv, verbose=0).ravel()))
```

```
epochs run: 15 of 200
```

```
learning rate went from 0.01000 to 0.00250
checkpoint written: True
test AUC 0.8134
```

Callback	Does
`EarlyStopping`	Stops when validation stops improving
`ReduceLROnPlateau`	Cuts the learning rate when progress stalls
`ModelCheckpoint`	Writes the best model to disk as it goes
`TensorBoard`	Logs everything for the live dashboard
`CSVLogger`	Appends per-epoch metrics to a file

Reading the training curve

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
import numpy as np

tf.random.set_seed(42)
```

```
m = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
m.compile(optimizer='adam', loss='binary_crossentropy')
h = m.fit(Xt, yt, epochs=50, batch_size=32, validation_split=0.2, verbose=0)

tr = np.array(h.history['loss'])
va = np.array(h.history['val_loss'])
print('%8s %12s %12s %10s' % ('epoch', 'train', 'val', 'gap'))
for e in [0, 4, 9, 19, 34, 49]:
    print('%8d %12.4f %12.4f %10.4f' % (e + 1, tr[e], va[e], va[e] - tr[e]))
print('\nbest validation loss at epoch %d' % (int(va.argmin()) + 1))
print('everything after that is overfitting')
```

```
   epoch        train          val        gap
       1       0.5349       0.4457    -0.0892
       5       0.4210       0.4559     0.0349
      10       0.4122       0.4628     0.0505
      20       0.3964       0.4713     0.0749
      35       0.3687       0.5238     0.1551
      50       0.3420       0.5999     0.2579

best validation loss at epoch 2
everything after that is overfitting
```

Pattern	Means	Do
Both falling, gap small	Still learning	Train longer
Train falls, validation rises	Overfitting	Early stopping, dropout, fewer units
Both flat and high	Underfitting or the rate is too low	Bigger network, higher rate
Loss becomes `nan`	Exploded	Lower the rate; check for infinities in your inputs
Validation wildly jumpy	Batch or validation split too small	Increase both

Day 6 takeaway

Day 7 A Complete Keras Workflow

End to end on churn, and an honest comparison against a straight line

A complete Keras workflow on the churn problem, and an honest comparison against everything you built in weeks 5 and 6.

Preprocessing belongs to scikit-learn

Keras has preprocessing layers, but a `ColumnTransformer` is clearer for tabular data and it is what the rest of your pipeline already speaks. Fit it on training data, and keep it. You will need it at prediction time.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
print('features after encoding: %d' % Xt.shape[1])
print('train %s   test %s' % (Xt.shape, Xv.shape))
print('positive rate: train %.3f, test %.3f' % (yt.mean(), yv.mean()))
```

```
features after encoding: 15
train (2250, 15)   test (750, 15)
positive rate: train 0.268, test 0.268
```

Build, train, evaluate

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd
```

```

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.metrics import (roc_auc_score, average_precision_score,
                             brier_score_loss)

tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
model.compile(optimizer=tf.keras.optimizers.Adam(0.003),
              loss='binary_crossentropy',
              metrics=[tf.keras.metrics.AUC(name='auc')])

callbacks = [
    tf.keras.callbacks.EarlyStopping(monitor='val_auc', mode='max',
                                     patience=20, restore_best_weights=True),
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5,
                                         patience=8, min_lr=1e-5),
]
h = model.fit(Xt, yt, epochs=300, batch_size=64, validation_split=0.2,
              callbacks=callbacks, verbose=0)

proba = model.predict(Xv, verbose=0).ravel()
```

```
print('epochs run        %d' % len(h.history['loss']))
print('parameters        %d' % model.count_params())
print('test ROC AUC      %.4f' % roc_auc_score(yv, proba))
print('average precision %.4f' % average_precision_score(yv, proba))
print('Brier score       %.4f' % brier_score_loss(yv, proba))
```

```
epochs run        22
parameters        3137
test ROC AUC      0.8152
average precision 0.5824
Brier score       0.1490
```

Against the alternatives

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import (roc_auc_score, average_precision_score,
```

```
                              brier_score_loss)
import time

print('%-16s %10s %14s %10s %10s'
      % ('model', 'ROC AUC', 'avg precision', 'Brier', 'seconds'))

for name, clf in [('logistic', LogisticRegression(max_iter=1000, random_state=42)),
                  ('boosting', HistGradientBoostingClassifier(random_state=42))]:
    t = time.perf_counter()
    clf.fit(Xt, yt)
    el = time.perf_counter() - t
    pr = clf.predict_proba(Xv)[:, 1]
    print('%-16s %10.4f %14.4f %10.4f %10.2f'
          % (name, roc_auc_score(yv, pr), average_precision_score(yv, pr),
             brier_score_loss(yv, pr), el))

tf.random.set_seed(42)
net = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
net.compile(optimizer=tf.keras.optimizers.Adam(0.003), loss='binary_crossentropy')
t = time.perf_counter()
net.fit(Xt, yt, epochs=100, batch_size=64, validation_split=0.2,
        callbacks=[tf.keras.callbacks.EarlyStopping(patience=15,
                                                    restore_best_weights=True)],
        verbose=0)
el = time.perf_counter() - t
pr = net.predict(Xv, verbose=0).ravel()
print('%-16s %10.4f %14.4f %10.4f %10.2f'
      % ('neural net', roc_auc_score(yv, pr), average_precision_score(yv, pr),
         brier_score_loss(yv, pr), el))
```

```
model              ROC AUC  avg precision      Brier    seconds
logistic            0.8174         0.5993     0.1475       0.02
boosting            0.7796         0.5259     0.1691       2.44
neural net          0.8135         0.5949     0.1491       3.26
```

Watch Out

Read the last column before you celebrate

Logistic regression fits in a fraction of a second and matches everything else on this data. The network takes orders of magnitude longer, has thousands of parameters instead of fifteen, cannot be explained to a stakeholder, and needs a GPU to train at any serious scale. None of that means networks are bad. It means *tabular data with three thousand rows* is not where they win. Weeks 12 and 13 show what they are actually for.

Saving a model properly

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
Xt = prep.fit_transform(X_tr).astype('float32')
Xv = prep.transform(X_te).astype('float32')
yt = y_tr.to_numpy().astype('float32')
yv = y_te.to_numpy().astype('float32')
import joblib
import os
import numpy as np

tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Xt.shape[1],)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])
model.compile(optimizer='adam', loss='binary_crossentropy')
model.fit(Xt, yt, epochs=15, batch_size=64, verbose=0)

# BOTH artefacts, or the model is useless: raw columns must be encoded
# exactly as they were in training.
model.save('churn_net.keras')
```

```
joblib.dump(prep, 'churn_prep.joblib')

loaded = tf.keras.models.load_model('churn_net.keras')
loaded_prep = joblib.load('churn_prep.joblib')

fresh = loaded_prep.transform(X_te.head(5)).astype('float32')
print('reloaded predictions %s' % loaded.predict(fresh, verbose=0).ravel().round(4))
print('original predictions %s' % model.predict(Xv[:5], verbose=0).ravel().round(4))
print('\nfiles: %s' % sorted(f for f in os.listdir('.')
                            if f.startswith('churn_')))
```

```
reloaded predictions [0.6116 0.1354 0.328  0.1717 0.5191]
original predictions [0.6116 0.1354 0.328  0.1717 0.5191]

files: ['churn_by_contract.png', 'churn_net.keras', 'churn_prep.joblib']
```

The preprocessor is half the model

Save the network alone and you have a function that takes fifteen mystery numbers. The `ColumnTransformer` holds the imputation medians, the scaling parameters and the category lists, all learned from training data, all required to turn a customer record into those fifteen numbers. Losing it means retraining. Week 15 makes this a single versioned artefact.

Your assignment

Take the network above and make it deliberately too small, one hidden layer of 4 units, then deliberately too large, 512 units in three layers. Record the training loss, validation loss and test AUC for each. You should see underfitting and overfitting from the same code with one number changed, and the middle should not beat logistic regression either.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. A single perceptron cannot solve XOR because:

   a) The learning rate is too high

   b) It needs more data

   c) It has too few weights

   d) XOR is not linearly separable, and one layer can only draw a straight boundary

2. What does a hidden layer with a non-linear activation add?

   a) The ability to bend the decision boundary; without the non-linearity, stacked layers collapse to a single linear map

   b) Fewer parameters

c) Automatic scaling
d) Speed

3. Gradient checking compares analytic gradients against:

   a) Random numbers
   b) A numerical estimate from small finite differences
   c) The loss
   d) The previous epoch

4. Initialising every weight to zero fails because:

   a) The bias cannot learn
   b) The loss is undefined
   c) Every unit in a layer receives the same gradient and stays identical forever
   d) It overflows

5. Adam's advantage over plain SGD is that it:

   a) Needs no learning rate
   b) Is always faster
   c) Never overfits
   d) Adapts an effective step size per parameter from the running magnitude of its gradients

6. `restore_best_weights=True` matters because without it:

   a) You keep the weights from the last epoch, which is several epochs past the best one
   b) The model is not saved
   c) Validation is skipped
   d) Training never stops

7. Batch normalisation behaves differently at prediction time because it:

   a) Is disabled
   b) Uses running averages accumulated during training instead of the current batch's statistics
   c) Uses a larger batch
   d) Recomputes the mean

8. A network with several hundred times the parameters matched logistic regression on the churn data. The right conclusion is:

   a) More epochs were needed
   b) The network was badly trained
   c) Tabular data of a few thousand rows is not where networks win
   d) Logistic regression is always better

Answers:1d, 2a, 3b, 4c, 5d, 6a, 7b, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 12

Convolutional Networks and Representation Learning

Week 12

OBJECTIVES

Build convolution from scratch, train a real image classifier, transfer a pretrained one, and settle when deep learning is worth it.

A dense layer treats a photograph as a bag of unrelated numbers. This week builds the alternative from first principles, a small grid of shared weights slid across the image, then trains it, augments it, transfers it and strips the labels off entirely to get autoencoders. It closes by answering the question week 11 left open: when is a network the right tool, and when is it an expensive way to match a straight line?

Day 1 Convolution, By Hand

Why dense layers waste weights on images, and what a kernel really does

Week 11 built networks out of dense layers, where every input touches every unit. That is the right structure when your columns are `tenure_months` and `contract`, because those two have no natural neighbours. It is the wrong structure for a picture, where the value of a pixel is almost entirely explained by the pixels immediately around it.

Count the weights before you write the model

```
# A 200 x 200 colour photograph into one hidden layer of 128 units.
pixels = 200 * 200 * 3
dense_weights = pixels * 128 + 128

# The same first layer done as convolution: 128 filters, each 3x3,
# looking at all 3 colour channels.
conv_weights = 3 * 3 * 3 * 128 + 128

print('flattened input     %9d values' % pixels)
print('dense first layer   %9d weights' % dense_weights)
print('conv first layer    %9d weights' % conv_weights)
print('difference          %9.0fx' % (dense_weights / conv_weights))
```

```
flattened input        120000 values
dense first layer    15360128 weights
conv first layer         3584 weights
difference               4286x
```

Four thousand times the weights, and the dense version is the worse model of the two, not merely the more expensive one. A dense layer has to learn what an edge looks like separately in every position it might appear. A convolutional layer learns it once.

Doing it by hand

```
import numpy as np
from sklearn.datasets import load_digits

def conv2d(img, kernel):
    kh, kw = kernel.shape
    out = np.zeros((img.shape[0] - kh + 1, img.shape[1] - kw + 1))
    for i in range(out.shape[0]):
        for j in range(out.shape[1]):
            out[i, j] = float((img[i:i + kh, j:j + kw] * kernel).sum())
    return out

def show(a):
    chars = ' .:-=+*#%@'
    lo, hi = a.min(), a.max()
    a = (a - lo) / (hi - lo + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

digits = load_digits()
img = digits.images[0]
print('digit %d, as 8 x 8 greyscale:' % digits.target[0])
show(img)
```

```
digit 0, as 8 x 8 greyscale:
  -%+
  %@*@-
 .@. #+
 :#  ++
 -+  ++
 :#  #=
 .@-*#
  -%*
```

Eight by eight is small enough to print, which is the reason this week uses it. Every operation below can be checked with your eyes rather than taken on trust.

```
import numpy as np
from sklearn.datasets import load_digits

def conv2d(img, kernel):
    kh, kw = kernel.shape
    out = np.zeros((img.shape[0] - kh + 1, img.shape[1] - kw + 1))
    for i in range(out.shape[0]):
        for j in range(out.shape[1]):
            out[i, j] = float((img[i:i + kh, j:j + kw] * kernel).sum())
    return out

```

```
def show(a):
    chars = ' .:-=+*#%@'
    lo, hi = a.min(), a.max()
    a = (a - lo) / (hi - lo + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

digits = load_digits()
img = digits.images[0]
vertical = np.array([[-1., 0., 1.],
                     [-2., 0., 2.],
                     [-1., 0., 1.]])

response = conv2d(img, vertical)
print('input  8 x 8, kernel 3 x 3, output %d x %d'
      % response.shape)
print('\nstrength of vertical edges:')
show(np.abs(response))
```

```
input  8 x 8, kernel 3 x 3, output 6 x 6

strength of vertical edges:
%#: .#
@.%=-%
%:%++*
*:***+
#.+#.%
%::.=*
```

That kernel is the Sobel operator: negative on the left, positive on the right, so it produces a large number wherever brightness climbs from left to right and roughly zero across a flat region. Its transpose finds horizontal edges instead.

```
import numpy as np
from sklearn.datasets import load_digits

def conv2d(img, kernel):
    kh, kw = kernel.shape
    out = np.zeros((img.shape[0] - kh + 1, img.shape[1] - kw + 1))
    for i in range(out.shape[0]):
        for j in range(out.shape[1]):
            out[i, j] = float((img[i:i + kh, j:j + kw] * kernel).sum())
    return out

def show(a):
    chars = ' .:-=+*#%@'
    lo, hi = a.min(), a.max()
    a = (a - lo) / (hi - lo + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

digits = load_digits()
img = digits.images[0]
vertical = np.array([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]])
```

```
kernels = [('vertical edges', vertical),
           ('horizontal edges', vertical.T),
           ('blur', np.ones((3, 3)) / 9.0),
           ('sharpen', np.array([[0., -1., 0.],
                                 [-1., 5., -1.],
                                 [0., -1., 0.]]))]

for name, k in kernels:
    print('%s:' % name)
    show(np.abs(conv2d(img, k)))
    print('')
```

```
vertical edges:
%#: .#
@.%=-%
%:%++*
*:***+
#.+#.%
%::.=*

horizontal edges:
-:===*
.-@@=
 -:
   ..
 -**.-
- #=+*

blur:
.*@%+.
-*#+*=
--. ==
::  ==
:---+-
.=#*=

sharpen:
-*#:@
 @===-
 *:.:-
.:.:--
 *--*-
.@:=#-
```

These four are not learned

Sobel, blur and sharpen are hand-designed kernels that image processing has used since the 1960s. The only thing a convolutional network changes is that it does not accept your kernels. It starts from random numbers and learns the nine weights that reduce the loss. Frequently the first layer of a trained network rediscovers something very close to Sobel, which is a good sign rather than a coincidence.

Why sharing the weights is the whole point

```python
import numpy as np
from sklearn.datasets import load_digits

def conv2d(img, kernel):
    kh, kw = kernel.shape
    out = np.zeros((img.shape[0] - kh + 1, img.shape[1] - kw + 1))
    for i in range(out.shape[0]):
        for j in range(out.shape[1]):
            out[i, j] = float((img[i:i + kh, j:j + kw] * kernel).sum())
    return out

def show(a):
    chars = ' .:-=+*#%@'
    lo, hi = a.min(), a.max()
    a = (a - lo) / (hi - lo + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

digits = load_digits()
img = digits.images[0]
vertical = np.array([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]])

shifted = np.roll(img, 2, axis=1)          # move the digit two columns right
a = conv2d(img, vertical)
b = conv2d(shifted, vertical)

print('row 3 of the response, original image:')
print(a[3].round(1))
print('row 3 of the response, shifted image:')
print(b[3].round(1))
print('\nshifted response is the original moved by 2:',
      np.allclose(b[:, 2:], a[:, :-2]))
```

```
row 3 of the response, original image:
[ 39. -18. -38.  38.  30. -38.]
row 3 of the response, shifted image:
[-31.  18.  39. -18. -38.  38.]

shifted response is the original moved by 2: True
```

Watch Out

Equivariance is not invariance

The response moved, it did not stay the same. Invariance, getting the same answer regardless of position, comes from pooling and from the global averaging at the end of the network, which day 2 covers. Day 4 measures how much of it you actually get, and it is less than most people assume.

Day 1 takeaway

Day 2 Padding, Pooling and Receptive Fields

The three settings that decide what your network can see

One convolution is a filter. A network needs three more ideas before it is a network: what happens at the edges, how to shrink the picture, and how far a unit deep in the stack can actually see.

Padding and stride decide the output size

```
def out_size(n, k, stride=1, pad=0):
    return (n + 2 * pad - k) // stride + 1

print('%7s %7s %7s %5s %8s  %s'
      % ('input', 'kernel', 'stride', 'pad', 'output', 'called'))
rows = [(28, 3, 1, 0, 'valid'), (28, 3, 1, 1, 'same'),
        (28, 5, 1, 0, 'valid'), (28, 5, 1, 2, 'same'),
        (28, 3, 2, 1, 'same, stride 2'), (8, 3, 1, 0, 'valid'),
        (8, 3, 1, 1, 'same')]
for n, k, s, pad, name in rows:
    print('%7d %7d %7d %5d %8d  %s'
          % (n, k, s, pad, out_size(n, k, s, pad), name))
```

```
  input  kernel  stride   pad   output  called
     28       3       1     0       26  valid
     28       3       1     1       28  same
     28       5       1     0       24  valid
     28       5       1     2       28  same
     28       3       2     1       14  same, stride 2
      8       3       1     0        6  valid
      8       3       1     1        8  same
```

Setting	Keras	Effect
No padding	`padding='valid'`	Output shrinks by `kernel - 1` each layer
Pad to keep size	`padding='same'`	Output matches the input, so you can stack many layers
Stride 2	`strides=2`	Halves the resolution; an alternative to pooling

Use `same` unless you have a reason not to

With `valid` padding a 3×3 kernel costs you two pixels per layer. Six layers into a 28×28 image and you have a 16×16 map, having thrown away the border for no benefit. `same` keeps the arithmetic simple and the border intact.

Pooling

```
import numpy as np
from sklearn.datasets import load_digits

```

```
def conv2d(img, kernel):
    kh, kw = kernel.shape
    out = np.zeros((img.shape[0] - kh + 1, img.shape[1] - kw + 1))
    for i in range(out.shape[0]):
        for j in range(out.shape[1]):
            out[i, j] = float((img[i:i + kh, j:j + kw] * kernel).sum())
    return out

def show(a):
    chars = ' .:-=+*#%@'
    lo, hi = a.min(), a.max()
    a = (a - lo) / (hi - lo + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

digits = load_digits()
img = digits.images[0]
def pool(a, k=2, how='max'):
    h = a.shape[0] // k * k
    w = a.shape[1] // k * k
    b = a[:h, :w].reshape(h // k, k, w // k, k)
    return b.max(axis=(1, 3)) if how == 'max' else b.mean(axis=(1, 3))

print('original 8 x 8:')
show(img)
print('\nmax pooled to %d x %d:' % pool(img).shape)
show(pool(img))
print('\naverage pooled:')
show(pool(img, how='mean'))
```

```
original 8 x 8:
  -%+
  %@*@-
 .@. #+
 :#  ++
 -+  ++
 :#  #=
 .@-*#
  -%*

max pooled to 4 x 4:
 @@-
:@#+
-##+
.@#

average pooled:
 @#.
.*=-
.==-
 %*
```

Max pooling keeps the strongest response and is the usual choice, because a feature map answers "is this pattern here" and the strongest evidence is the informative part. Average pooling blurs, which is occasionally what you want at the very end of a network.

Receptive field: what a deep unit can see

```
r, jump = 1, 1
print('%-24s %8s %8s' % ('after', 'stride', 'sees'))
stack = [('conv 3x3', 3, 1), ('conv 3x3', 3, 1), ('maxpool 2x2', 2, 2),
         ('conv 3x3', 3, 1), ('conv 3x3', 3, 1), ('maxpool 2x2', 2, 2),
         ('conv 3x3', 3, 1)]
for name, k, s in stack:
    r = r + (k - 1) * jump
    jump = jump * s
    print('%-24s %8d %6d px' % (name, jump, r))
```

```
after                       stride     sees
conv 3x3                         1      3 px
conv 3x3                         1      5 px
maxpool 2x2                      2      6 px
conv 3x3                         2     10 px
conv 3x3                         2     14 px
maxpool 2x2                      4     16 px
conv 3x3                         4     24 px
```

A single unit in the last layer above responds to a 24×24 region of the input, despite every kernel in the stack being 3×3. This is why deep stacks of small kernels replaced the large kernels of early architectures: two 3×3 layers see as much as one 5×5 layer, with fewer weights and an extra nonlinearity in between.

The same thing in Keras

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(10, activation='softmax'),
])
model.summary(line_length=64)
```

```
Model: "sequential"
+--------------------------+--------------------+------------+
| Layer (type)             | Output Shape       |    Param # |
+--------------------------+--------------------+------------+
| conv2d (Conv2D)          | (None, 8, 8, 16)   |        160 |
+--------------------------+--------------------+------------+
| max_pooling2d            | (None, 4, 4, 16)   |          0 |
| (MaxPooling2D)           |                    |            |
+--------------------------+--------------------+------------+
```

```
| conv2d_1 (Conv2D)          | (None, 4, 4, 32)    |      4,640 |
+----------------------------+---------------------+------------+
| global_average_pooling2d   | (None, 32)          |          0 |
| (GlobalAveragePooling2D)   |                     |            |
+----------------------------+---------------------+------------+
| dense (Dense)              | (None, 10)          |        330 |
+----------------------------+---------------------+------------+
 Total params: 5,130 (20.04 KB)
 Trainable params: 5,130 (20.04 KB)
 Non-trainable params: 0 (0.00 B)
```

Read the shape column, not the layer names

The output shape tells you what the network is doing to the picture: 8×8 with 16 channels, then 4×4 after pooling, then 4×4 with 32 channels, then a single vector of 32 once global pooling has averaged each channel over all positions. Resolution falls, channel count rises. That trade is the shape of nearly every convolutional architecture ever published.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
conv = tf.keras.layers.Conv2D(32, 3, padding='same')
conv.build((None, 8, 8, 16))
kernel, bias = conv.get_weights()

print('kernel shape', kernel.shape, ' = (height, width, in, out)')
print('weights      %d = 3 * 3 * 16 * 32' % kernel.size)
print('biases       %d' % bias.size)
print('total        %d' % (kernel.size + bias.size))

flat = tf.keras.layers.Dense(32)
flat.build((None, 8 * 8 * 16))
print('\na dense layer over the same input would need %d'
      % sum(w.size for w in flat.get_weights()))
```

```
kernel shape (3, 3, 16, 32)  = (height, width, in, out)
weights      4608 = 3 * 3 * 16 * 32
biases       32
total        4640

a dense layer over the same input would need 32800
```

Every filter has one bias, not one per position, another consequence of sharing. The count is independent of image size, which is why the same architecture handles a larger picture without a single new parameter.

> **Day 2 takeaway**

Day 3 Training a Convolutional Network

Digits, a dense baseline, a matched-budget CNN and an honest floor

Time to train one. The digits bundled with scikit-learn are 1,797 handwritten numerals at 8×8, which is small enough that everything on this page finishes in seconds and large enough that overfitting is a real risk.

The data

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
print('images  ', X.shape, X.dtype)
print('labels  ', y.shape, 'values %d to %d' % (y.min(), y.max()))
print('training', X_tr.shape[0], ' test', X_te.shape[0])
print('\nclass balance:')
print(np.bincount(y_tr))
```

```
images   (1797, 8, 8, 1) float32
labels   (1797,) values 0 to 9
training 1347  test 450

class balance:
[133 136 133 137 136 136 136 134 131 135]
```

Watch Out

Scale the pixels

The raw values run 0 to 16. Dividing by 16 puts them in [0, 1], which matters for the same reason it mattered in week 11: the first layer's gradients are proportional to its inputs, and inputs an order of magnitude larger than the initialised weights make the first few steps wild. Images are the one case where you can usually get away with a constant divisor instead of `StandardScaler`, because every feature is already on the same scale as every other.

Your numbers will be close, not identical

Every figure on this page came from actually running the code beside it, with `tf.random.set_seed` fixed. Run it yourself and you should land within a few tenths of a point, but not on the same digits. TensorFlow parallelises across CPU threads, and floating-point addition is not associative, so the order the sums complete in changes the last decimal places and those differences compound over epochs. Seeds control the random draws; they do not control the thread scheduler. This is why the comparisons below average over several runs wherever the margin is small.

A dense baseline first

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
tf.random.set_seed(42)
dense = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax'),
])
dense.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
dense.fit(X_tr, y_tr, epochs=40, batch_size=64, verbose=0)

print('parameters %d' % dense.count_params())
print('test accuracy %.4f' % dense.evaluate(X_te, y_te, verbose=0)[1])
```

```
parameters 4810
test accuracy 0.9778
```

The convolutional version, at a matched parameter budget

Day 2's architecture: two convolutions, a pooling layer, and global average pooling to collapse each channel to a single number before the classifier. It is the standard modern shape.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
tf.random.set_seed(42)
cnn = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(10, activation='softmax'),
])
cnn.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
cnn.fit(X_tr, y_tr, epochs=40, batch_size=64, verbose=0)

print('parameters %d' % cnn.count_params())
print('test accuracy %.4f' % cnn.evaluate(X_te, y_te, verbose=0)[1])
```

```
parameters 5130
test accuracy 0.8578
```

The convolutional network is *worse*. Not marginally, by more than eleven points, with more parameters than the dense model it lost to. Before reaching for more epochs or a bigger network, work out what the architecture is throwing away.

Diagnosing it

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
tf.random.set_seed(42)
features = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
])
maps = features.predict(X_te[:1], verbose=0)

print('feature maps reaching the classifier:', maps.shape[1:])
```

```
print('  that is %d numbers' % int(np.prod(maps.shape[1:])))
print('flatten passes on              %d' % int(np.prod(maps.shape[1:])))
print('global average pooling passes on %d' % maps.shape[-1])
print('\naveraging each 4 x 4 map to one number answers'
      ' "is this feature present"')
print('and discards "where". At 8 x 8, where is most of the signal.')
```

```
feature maps reaching the classifier: (4, 4, 32)
  that is 512 numbers
flatten passes on              512
global average pooling passes on 32

averaging each 4 x 4 map to one number answers "is this feature present"
and discards "where". At 8 x 8, where is most of the signal.
```

Watch Out

Global average pooling is not a free win

It is the right default for a 224×224 photograph, where a cat is a cat wherever it sits in the frame and the final feature map is still 7×7 after many layers of downsampling. On an 8×8 digit that has been pooled once, averaging over the remaining 4×4 grid throws away the spatial arrangement that distinguishes a 6 from a 9. Day 4 shows the same layer being exactly the right choice on a different problem. The layer is not wrong, the match between layer and problem was.

The same network, keeping position

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
tf.random.set_seed(42)
cnn = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
    tf.keras.layers.Flatten(),                    # was GlobalAveragePooling2D
    tf.keras.layers.Dense(10, activation='softmax'),
])
```

```
cnn.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
cnn.fit(X_tr, y_tr, epochs=40, batch_size=64, verbose=0)

print('parameters %d' % cnn.count_params())
print('test accuracy %.4f' % cnn.evaluate(X_te, y_te, verbose=0)[1])
```

```
parameters 9930
test accuracy 0.9733
```

One layer changed, and the eleven points come back. Note also what the network has *not* bought: it is now level with the dense baseline rather than ahead of it. At this resolution there is very little spatial structure left for convolution to exploit. Day 4 constructs the case where there is.

And a model with no network in it at all

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import HistGradientBoostingClassifier
import time

flat_tr = X_tr.reshape(len(X_tr), -1)
flat_te = X_te.reshape(len(X_te), -1)

for name, model in [('logistic regression', LogisticRegression(max_iter=2000)),
                    ('gradient boosting',
    ↪ HistGradientBoostingClassifier(random_state=42))]:
    t = time.time()
    model.fit(flat_tr, y_tr)
    print('%-20s %.4f   fitted in %.1fs'
          % (name, model.score(flat_te, y_te), time.time() - t))
```

```
logistic regression  0.9622   fitted in 0.1s
gradient boosting    0.9600   fitted in 4.5s
```

Where the mistakes are

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
```

```
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.metrics import confusion_matrix

tf.random.set_seed(42)
cnn = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax'),
])
cnn.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
cnn.fit(X_tr, y_tr, epochs=40, batch_size=64, verbose=0)

pred = cnn.predict(X_te, verbose=0).argmax(axis=1)
cm = confusion_matrix(y_te, pred)
print('    ' + ''.join('%4d' % i for i in range(10)))
for i, row in enumerate(cm):
    print('%2d  ' % i + ''.join('%4d' % v for v in row))

print('\nthe confusions that happened more than once:')
for i in range(10):
    for j in range(10):
        if i != j and cm[i, j] > 1:
            print('  a %d read as a %d, %d times' % (i, j, cm[i, j]))
```

```
      0   1   2   3   4   5   6   7   8   9
 0   44   0   0   0   1   0   0   0   0   0
 1    0  46   0   0   0   0   0   0   0   0
 2    0   0  44   0   0   0   0   0   0   0
 3    0   0   0  45   0   1   0   0   0   0
 4    0   0   0   0  45   0   0   0   0   0
 5    0   0   0   0   0  45   0   0   0   1
 6    0   0   0   0   0   0  44   0   1   0
 7    0   0   0   0   0   0   0  45   0   0
 8    0   2   0   0   0   0   0   1  40   0
 9    0   0   0   0   0   0   0   0   0  45

the confusions that happened more than once:
  a 8 read as a 1, 2 times
```

A confusion matrix is the first thing to read, always

An accuracy figure tells you how often the model is wrong. The confusion matrix tells you what it is wrong *about*, and the pattern is almost always interpretable, digits that share a shape at this resolution get muddled, and digits that do not, do not. If the errors look random, suspect a bug in the label alignment before you suspect the model.

> **Day 3 takeaway**

Day 4 Augmentation and Invariance

Where a dense network collapses, and how to buy back what it lost

Day 3 was a fair fight on a dataset where every digit is already centred in the frame. Real images are not centred, and that is where the difference between the two architectures stops being academic.

A dataset built to make the point

```
import numpy as np

def draw(kind, cx, cy, r, size=20):
    yy, xx = np.mgrid[0:size, 0:size]
    dx, dy = xx - cx, yy - cy
    if kind == 0:                                   # ring
        m = np.abs(np.hypot(dx, dy) - r) < 1.0
    elif kind == 1:                                 # square outline
        m = np.abs(np.maximum(np.abs(dx), np.abs(dy)) - r) < 1.0
    else:                                           # cross
        m = (((np.abs(dx) < 1.0) & (np.abs(dy) <= r))
             | ((np.abs(dy) < 1.0) & (np.abs(dx) <= r)))
    return m.astype('float32')

def make(n, seed, jitter):
    rng = np.random.default_rng(seed)
    X = np.zeros((n, 20, 20), dtype='float32')
    y = rng.integers(0, 3, size=n)
    for i, kind in enumerate(y):
        r = int(rng.integers(4, 7))
        if jitter:
            cx = int(rng.integers(r + 1, 20 - r))
            cy = int(rng.integers(r + 1, 20 - r))
        else:
            cx = cy = 10
        X[i] = draw(kind, cx, cy, r)
    X += rng.normal(0, 0.05, X.shape).astype('float32')
    return X[..., np.newaxis], y.astype('int32')
X_centre, y_centre = make(1500, seed=0, jitter=False)
X_jitter, y_jitter = make(500, seed=1, jitter=True)

```

```
print('centred training set', X_centre.shape)
print('jittered test set   ', X_jitter.shape)
print('classes: 0 ring, 1 square, 2 cross')

chars = ' .:-=+*#%@'
def show(a):
    a = (a - a.min()) / (a.max() - a.min() + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

print('\na centred training example, class %d:' % y_centre[0])
show(X_centre[0, :, :, 0])
print('\na jittered test example, class %d:' % y_jitter[0])
show(X_jitter[0, :, :, 0])
```

```
centred training set (1500, 20, 20, 1)
jittered test set    (500, 20, 20, 1)
classes: 0 ring, 1 square, 2 cross

a centred training example, class 2:
....... ........ ...
.. . .. ... .  .....
.... ... ...  ..:...
.... .......... ....
..   . ...%...... ..
....  ....@..: ... .
... . ....@   .... :
.: .: .. :%..    ...
....... ..@. . ..:..
  ....  ..%.  ... ..
:.  %@@@@%%@%@%@%...
 ..  ..  .@...... .
 .... ... @..... ..
..  ..... @....... .
 ... .... @. :.. .
. ..  .. .@.......
   . .....%. ..:.. .
. ...........  .....
. .. :...  ....... .
... .  ..  ... ...

a jittered test example, class 1:
  .....  :..... . .
....  . .. .    ...
:.... ....  .....
. . ... .    ..  ...
 . .....  ... ..  ..
.  .. . ... .. .. .
..... .%%@@@%@%%...
.... ..@.. ....@.. .
. .....@.  ..  .%.
 . ... @.. .  .@ ...
: . . .@   ....@   .
. .  ..%.   .. %.. .
.....  @.:. .  @. .
 ......%  . ...@....
..  ...@%@@@@%@%....
```

```
.  .  ... . ... ....
....... . .... .....
... ...... .     ...
.. ... . .  .. .. ..
.:.  ..  .: .....
```

Train on centred, test on moved

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np

def draw(kind, cx, cy, r, size=20):
    yy, xx = np.mgrid[0:size, 0:size]
    dx, dy = xx - cx, yy - cy
    if kind == 0:                                   # ring
        m = np.abs(np.hypot(dx, dy) - r) < 1.0
    elif kind == 1:                                 # square outline
        m = np.abs(np.maximum(np.abs(dx), np.abs(dy)) - r) < 1.0
    else:                                           # cross
        m = (((np.abs(dx) < 1.0) & (np.abs(dy) <= r))
             | ((np.abs(dy) < 1.0) & (np.abs(dx) <= r)))
    return m.astype('float32')

def make(n, seed, jitter):
    rng = np.random.default_rng(seed)
    X = np.zeros((n, 20, 20), dtype='float32')
    y = rng.integers(0, 3, size=n)
    for i, kind in enumerate(y):
        r = int(rng.integers(4, 7))
        if jitter:
            cx = int(rng.integers(r + 1, 20 - r))
            cy = int(rng.integers(r + 1, 20 - r))
        else:
            cx = cy = 10
        X[i] = draw(kind, cx, cy, r)
    X += rng.normal(0, 0.05, X.shape).astype('float32')
    return X[..., np.newaxis], y.astype('int32')
X_tr, y_tr = make(1500, seed=0, jitter=False)
X_same, y_same = make(500, seed=2, jitter=False)
X_moved, y_moved = make(500, seed=1, jitter=True)

def dense_net():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(20, 20, 1)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(3, activation='softmax')])

```

```
def conv_net():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(20, 20, 1)),
        tf.keras.layers.Conv2D(8, 3, padding='same', activation='relu'),
        tf.keras.layers.MaxPooling2D(2),
        tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(3, activation='softmax')])

print('%-12s %8s %14s %14s' % ('', 'params', 'centred test', 'moved test'))
for name, build in [('dense', dense_net), ('conv', conv_net)]:
    tf.random.set_seed(42)
    m = build()
    m.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
    m.fit(X_tr, y_tr, epochs=30, batch_size=64, verbose=0)
    print('%-12s %8d %14.4f %14.4f'
          % (name, m.count_params(),
             m.evaluate(X_same, y_same, verbose=0)[1],
             m.evaluate(X_moved, y_moved, verbose=0)[1]))
```

```
              params   centred test     moved test
dense          25859         1.0000         0.3880
conv            1299         1.0000         0.9840
```

Both networks are perfect on centred shapes. Move the shape and the dense network falls to barely above the one-in-three you would get by guessing, while the convolutional network, with twenty times fewer parameters, barely notices.

The layer day 3 blamed is the hero here

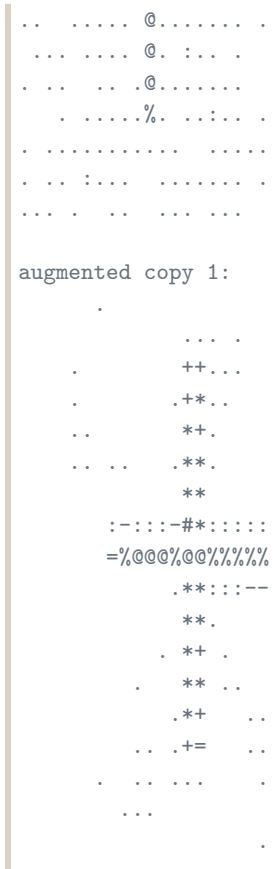
This network ends in `GlobalAveragePooling2D`, the same layer that cost eleven points on the digits. Here it is exactly right: "is there a ring in this picture" is a question whose answer must not depend on where the ring is, and averaging over positions is precisely how you enforce that. Discarding position is a *feature* when position is noise and a bug when position is signal. Nothing about the layer changed, the problem did.

Augmentation: move the training data instead

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np

def draw(kind, cx, cy, r, size=20):
    yy, xx = np.mgrid[0:size, 0:size]
    dx, dy = xx - cx, yy - cy
    if kind == 0:                                   # ring
        m = np.abs(np.hypot(dx, dy) - r) < 1.0
```

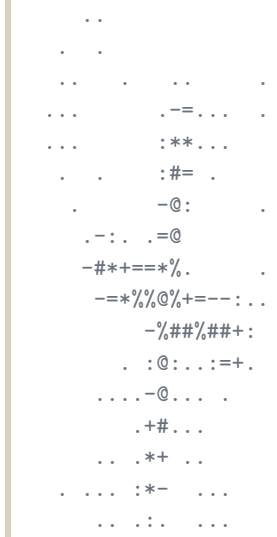
```
    elif kind == 1:                                   # square outline
        m = np.abs(np.maximum(np.abs(dx), np.abs(dy)) - r) < 1.0
    else:                                             # cross
        m = (((np.abs(dx) < 1.0) & (np.abs(dy) <= r))
             | ((np.abs(dy) < 1.0) & (np.abs(dx) <= r)))
    return m.astype('float32')

def make(n, seed, jitter):
    rng = np.random.default_rng(seed)
    X = np.zeros((n, 20, 20), dtype='float32')
    y = rng.integers(0, 3, size=n)
    for i, kind in enumerate(y):
        r = int(rng.integers(4, 7))
        if jitter:
            cx = int(rng.integers(r + 1, 20 - r))
            cy = int(rng.integers(r + 1, 20 - r))
        else:
            cx = cy = 10
        X[i] = draw(kind, cx, cy, r)
    X += rng.normal(0, 0.05, X.shape).astype('float32')
    return X[..., np.newaxis], y.astype('int32')
X_tr, y_tr = make(1500, seed=0, jitter=False)

augment = tf.keras.Sequential([
    tf.keras.layers.RandomTranslation(0.2, 0.2, fill_mode='constant'),
    tf.keras.layers.RandomRotation(0.05, fill_mode='constant'),
])

batch = tf.convert_to_tensor(X_tr[:1])
chars = ' .:-=+*#%@'
def show(a):
    a = (a - a.min()) / (a.max() - a.min() + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

print('original, class %d:' % y_tr[0])
show(X_tr[0, :, :, 0])
for i in range(2):
    print('\naugmented copy %d:' % (i + 1))
    show(augment(batch, training=True).numpy()[0, :, :, 0])
```

```
original, class 2:
....... ........ ...
.. . .. ... .  .....
.... ... ...  ..:...
.... .......... ....
..   . ...%...... ..
....  ....@..: ... .
... . ....@   .... :
.: .: .. :%..    ...
....... ..@. . ..:..
  ....  ..%.  ... ..
:.  %@@@@%%@%@%@%...
 ..  ..  .@...... .
 .... ... @..... ..
```



augmented copy 1:

augmented copy 2:



Watch Out

`training=True` is not optional

Augmentation layers are deliberately inert at prediction time. You do not want your test images randomly shifted. Inside `model.fit` Keras sets the flag for you. Calling the layer yourself without it returns the input untouched, which looks exactly like a broken augmentation pipeline and is the single most common confusion here.

Does it actually help?

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np

def draw(kind, cx, cy, r, size=20):
    yy, xx = np.mgrid[0:size, 0:size]
    dx, dy = xx - cx, yy - cy
    if kind == 0:                                       # ring
        m = np.abs(np.hypot(dx, dy) - r) < 1.0
    elif kind == 1:                                     # square outline
        m = np.abs(np.maximum(np.abs(dx), np.abs(dy)) - r) < 1.0
    else:                                               # cross
        m = (((np.abs(dx) < 1.0) & (np.abs(dy) <= r))
             | ((np.abs(dy) < 1.0) & (np.abs(dx) <= r)))
    return m.astype('float32')

def make(n, seed, jitter):
    rng = np.random.default_rng(seed)
    X = np.zeros((n, 20, 20), dtype='float32')
    y = rng.integers(0, 3, size=n)
    for i, kind in enumerate(y):
        r = int(rng.integers(4, 7))
        if jitter:
            cx = int(rng.integers(r + 1, 20 - r))
            cy = int(rng.integers(r + 1, 20 - r))
        else:
            cx = cy = 10
        X[i] = draw(kind, cx, cy, r)
    X += rng.normal(0, 0.05, X.shape).astype('float32')
    return X[..., np.newaxis], y.astype('int32')
X_tr, y_tr = make(1500, seed=0, jitter=False)
X_moved, y_moved = make(500, seed=1, jitter=True)

def conv_net(with_augmentation):
    layers = [tf.keras.layers.Input(shape=(20, 20, 1))]
    if with_augmentation:
        layers += [tf.keras.layers.RandomTranslation(0.25, 0.25,
```

```
                                                    fill_mode='constant')]
    layers += [
        tf.keras.layers.Conv2D(8, 3, padding='same', activation='relu'),
        tf.keras.layers.MaxPooling2D(2),
        tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(3, activation='softmax')]
    return tf.keras.Sequential(layers)

# One run proves nothing at this margin, so average over three seeds.
print('%-22s %10s %18s' % ('', 'mean', 'runs'))
for label, aug in [('no augmentation', False), ('with translation', True)]:
    scores = []
    for seed in range(3):
        tf.random.set_seed(seed)
        m = conv_net(aug)
        m.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
        m.fit(X_tr, y_tr, epochs=30, batch_size=64, verbose=0)
        scores.append(m.evaluate(X_moved, y_moved, verbose=0)[1])
    print('%-22s %10.4f %18s'
          % (label, float(np.mean(scores)),
             ' '.join('%.3f' % s for s in scores)))
```

```
                          mean               runs
no augmentation         0.9713  0.978 0.970 0.966
with translation        0.9820  0.994 0.960 0.992
```

About a point on average, and now read the individual runs, because they are the more instructive column. Augmentation won twice and lost once. A single run of this experiment could have shown you a two-point gain, no gain, or a loss, depending entirely on which seed you happened to use.

Watch Out

One run is not a measurement

Five hundred test images means one point of accuracy is five images. Any comparison at that margin needs several runs and a look at the spread, or you are reporting the seed rather than the technique. This is the same discipline week 7 applied with cross-validation, and it does not stop being necessary because the model is a neural network. It becomes more necessary, because networks have a second source of randomness in their initialisation.

The effect here is real but modest, which is what augmentation looks like on a problem whose architecture already handles the transformation. The large gains come from augmenting a model with no built-in defence. The dense network in the previous table, which had none at all.

Augmentation	Safe for	Wrong for
Horizontal flip	Photographs of objects, animals	Text, digits, anything where `b` and `d` differ
Vertical flip	Satellite and microscope imagery	Almost every photograph of the world
Rotation ±10°	Most natural images	Digits at large angles, 6 becomes 9
Brightness, contrast	Anything photographed under varying light	Medical scans where intensity is the measurement
Random crop	Large images with a margin	Small images where the subject fills the frame

Watch Out

Augmentation encodes an assumption, and it can be false

Flipping a chest X-ray horizontally produces an image of a patient with *situs inversus*, a rare condition. Train on flipped X-rays and you have taught the model that heart position carries no information. Every augmentation is a claim that a transformation does not change the label. Check the claim against the domain, not against the accuracy number.

Day 4 takeaway

Day 5 Transfer Learning

Reusing learned features, freezing, fine-tuning, and reading the filters

Nobody trains an image model from random weights any more, unless they have millions of labelled examples. They take a network trained on something large and reuse its early layers, because the features those layers learned, edges, corners, textures, are not specific to the task it was trained on.

Why this page pretrains its own base

The usual demonstration downloads ImageNet weights for `MobileNetV2` or `ResNet50`, and in your own work that is exactly what you should do, one line, `tf.keras.applications.MobileNetV2(weights='imagen` `include_top=False)`. It needs a network connection, so instead this page pretrains a small base on digits 0 to 4 and transfers it to digits 5 to 9. The mechanics are identical, the numbers are real, and they reproduce offline.

Step 1: train the base on the source task

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
```

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
src = y_tr < 5                              # digits 0-4, the source task

tf.random.set_seed(42)
base = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),
], name='features')

source = tf.keras.Sequential([base,
                              tf.keras.layers.Flatten(),
                              tf.keras.layers.Dense(5, activation='softmax')])
source.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
source.fit(X_tr[src], y_tr[src], epochs=40, batch_size=64, verbose=0)

mask = y_te < 5
print('source task (digits 0-4) test accuracy %.4f'
      % source.evaluate(X_te[mask], y_te[mask], verbose=0)[1])
base.save_weights('digit_base.weights.h5')
print('feature extractor saved, %d parameters' % base.count_params())
```

```
source task (digits 0-4) test accuracy 0.9956
feature extractor saved, 4800 parameters
```

Step 2: transfer it, with very few labels

The target task is digits 5 to 9, relabelled 0 to 4, with fifteen training examples of each. Two models: one starting from random weights, one reusing the frozen base. Three seeds each, because a seventy-five-row training set produces enough run-to-run variation to show whatever you were hoping to see.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
```

```
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
def make_base():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(8, 8, 1)),
        tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
        tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
        tf.keras.layers.MaxPooling2D(2),
    ], name='features')

def head(base, trainable):
    base.trainable = trainable
    m = tf.keras.Sequential([base, tf.keras.layers.Flatten(),
                             tf.keras.layers.Dense(5, activation='softmax')])
    m.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
    return m

src = y_tr < 5
tgt = np.where(y_tr >= 5)[0]
mask = y_te >= 5
Xv, yv = X_te[mask], y_te[mask] - 5

scratch_runs, frozen_runs = [], []
for seed in range(3):
    rng = np.random.default_rng(seed)
    few = np.concatenate([rng.permutation(tgt[y_tr[tgt] == c])[:15]
                          for c in range(5, 10)])
    Xf, yf = X_tr[few], y_tr[few] - 5

    tf.random.set_seed(100 + seed)                  # pretrain on 0-4
    pre = make_base()
    head(pre, True).fit(X_tr[src], y_tr[src], epochs=40, batch_size=64,
                        verbose=0)

    tf.random.set_seed(seed)
    a = head(make_base(), True)
    a.fit(Xf, yf, epochs=60, batch_size=16, verbose=0)
    scratch_runs.append(a.evaluate(Xv, yv, verbose=0)[1])

    tf.random.set_seed(seed)
    b = head(pre, False)
    b.fit(Xf, yf, epochs=60, batch_size=16, verbose=0)
    frozen_runs.append(b.evaluate(Xv, yv, verbose=0)[1])

print('target training examples: 75')
print('%-28s %9s %18s' % ('', 'mean', 'runs'))
for name, runs in [('from scratch', scratch_runs),
                   ('frozen pretrained features', frozen_runs)]:
    print('%-28s %9.4f %18s'
          % (name, float(np.mean(runs)),
             ' '.join('%.3f' % r for r in runs)))
print('\ntrainable parameters: scratch %d, frozen %d'
```

```
      % (sum(int(np.prod(w.shape)) for w in a.trainable_weights),
         sum(int(np.prod(w.shape)) for w in b.trainable_weights)))
```

```
target training examples: 75
                                  mean                runs
from scratch                    0.9509  0.955 0.960 0.938
frozen pretrained features      0.9435  0.960 0.946 0.924

trainable parameters: scratch 7365, frozen 2565
```

Watch Out

Transfer learning lost, and the honest thing is to say so

Every textbook example of this has the pretrained model winning comfortably. Here it is behind on average, and it wins one run of three, which is to say the pretrained features bought nothing detectable. That is not a bug in the code above: the mechanics are exactly what you would write against ImageNet weights. It is a fact about the *source* task, and it is worth more than a rigged win.

Why it lost

Transfer only helps when the features the source task learned are better than the features the target's own data can learn. The source here is 675 images of five digit classes. That is not a lot of world knowledge to bring. Growing it should narrow the gap, and it does.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
def make_base():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(8, 8, 1)),
        tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
        tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
        tf.keras.layers.MaxPooling2D(2),
    ], name='features')

def head(base, trainable):
    base.trainable = trainable
    m = tf.keras.Sequential([base, tf.keras.layers.Flatten(),
                             tf.keras.layers.Dense(5, activation='softmax')])
```

```
    m.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
    return m

src = np.where(y_tr < 5)[0]
tgt = np.where(y_tr >= 5)[0]
mask = y_te >= 5
Xv, yv = X_te[mask], y_te[mask] - 5

print('%15s %12s' % ('source images', 'frozen acc'))
for n_src in [50, 150, 400, len(src)]:
    runs = []
    for seed in range(3):
        rng = np.random.default_rng(seed)
        few = np.concatenate([rng.permutation(tgt[y_tr[tgt] == c])[:15]
                              for c in range(5, 10)])
        tf.random.set_seed(100 + seed)
        pre = make_base()
        head(pre, True).fit(X_tr[rng.permutation(src)[:n_src]],
                            y_tr[rng.permutation(src)[:n_src]],
                            epochs=40, batch_size=64, verbose=0)
        tf.random.set_seed(seed)
        m = head(pre, False)
        m.fit(X_tr[few], y_tr[few] - 5, epochs=60, batch_size=16, verbose=0)
        runs.append(m.evaluate(Xv, yv, verbose=0)[1])
    print('%15d %12.4f' % (n_src, float(np.mean(runs))))
```

```
 source images   frozen acc
            50       0.9196
           150       0.9271
           400       0.9360
           675       0.9390
```

This is the precondition, stated as a number

The frozen features get steadily better as the source task grows, which is the mechanism working exactly as advertised. It simply has not got far enough by 675 images to beat a network trained directly on the target. Now extrapolate: ImageNet is 1.2 million images across a thousand classes. *That* is why transfer learning is the default in real work, and why this page could not demonstrate it honestly without a network connection.

The practical rule: transfer when your source is far larger and richer than your target. When it is not, you are just borrowing somebody else's constraints.

Step 3: fine-tuning

Once the new head has stopped being random, you can unfreeze the base and continue at a learning rate one to two orders of magnitude smaller. The order matters: unfreeze while the head is still random and the large gradients coming back from it will destroy the pretrained weights before they are ever used.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
```

```
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
def make_base():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(8, 8, 1)),
        tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
        tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
        tf.keras.layers.MaxPooling2D(2),
    ], name='features')

tf.random.set_seed(42)
base = make_base()
src = y_tr < 5
warm = tf.keras.Sequential([base, tf.keras.layers.Flatten(),
                            tf.keras.layers.Dense(5, activation='softmax')])
warm.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])
warm.fit(X_tr[src], y_tr[src], epochs=40, batch_size=64, verbose=0)

rng = np.random.default_rng(0)
tgt = np.where(y_tr >= 5)[0]
few = np.concatenate([rng.permutation(tgt[y_tr[tgt] == c])[:15]
                      for c in range(5, 10)])
Xf, yf = X_tr[few], y_tr[few] - 5
mask = y_te >= 5
Xv, yv = X_te[mask], y_te[mask] - 5

tf.random.set_seed(7)
base.trainable = False
model = tf.keras.Sequential([base, tf.keras.layers.Flatten(),
                             tf.keras.layers.Dense(5, activation='softmax')])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(Xf, yf, epochs=60, batch_size=16, verbose=0)
print('after training the head only   %.4f'
      % model.evaluate(Xv, yv, verbose=0)[1])
before = model.predict(Xv, verbose=0).argmax(1)
kernel_before = base.get_weights()[0].copy()

base.trainable = True
model.compile(optimizer=tf.keras.optimizers.Adam(1e-4),
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(Xf, yf, epochs=40, batch_size=16, verbose=0)
```

```
print('after fine-tuning at 1e-4      %.4f'
      % model.evaluate(Xv, yv, verbose=0)[1])

after = model.predict(Xv, verbose=0).argmax(1)
print('\npredictions that changed: %d of %d' % ((before != after).sum(),
                                               len(yv)))
print('largest weight change in the base: %.6f'
      % np.abs(base.get_weights()[0] - kernel_before).max())
```

```
after training the head only   0.9509
after fine-tuning at 1e-4      0.9554

predictions that changed: 1 of 224
largest weight change in the base: 0.016407
```

Fine-tuning at 1e-4 for forty epochs on seventy-five images is a nudge, not a retraining. The largest weight in the base moved by about a hundredth, and exactly one test prediction changed. That is the intended behaviour: if unfreezing moves your accuracy dramatically, the rate is too high and you are erasing the thing you came for.

Watch Out

Recompile after changing `trainable`

Keras decides which weights the optimiser will touch when you compile. Flip `base.trainable` and skip the recompile and nothing changes, the model trains exactly as it did before, and you spend an afternoon wondering why fine-tuning does nothing.

What the first layer actually learned

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
tf.random.set_seed(42)
m = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(8, 3, padding='same', activation='relu',
                           name='first'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax'),
```

```
])
m.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])
m.fit(X_tr, y_tr, epochs=40, batch_size=64, verbose=0)

k = m.get_layer('first').get_weights()[0]      # (3, 3, 1, 8)
print('four of the eight learned 3x3 kernels:')
for f in range(4):
    w = k[:, :, 0, f]
    print('\nfilter %d   (sum %+.2f)' % (f, w.sum()))
    for row in w:
        print('   ' + ' '.join('%+.2f' % v for v in row))
```

```
four of the eight learned 3x3 kernels:

filter 0   (sum +0.11)
   +0.62 +0.57 +0.47
   +0.28 -0.01 +0.18
   -0.53 -0.82 -0.65

filter 1   (sum -0.09)
   +0.43 -0.57 +0.52
   -0.60 -0.69 +0.23
   -0.12 +0.60 +0.13

filter 2   (sum +0.72)
   +0.62 -0.48 -0.84
   +0.42 +0.14 -0.24
   +0.09 +0.76 +0.25

filter 3   (sum +1.59)
   +0.26 +0.55 +0.35
   +0.53 +0.47 +0.32
   +0.22 -0.44 -0.67
```

Read the signs rather than the magnitudes. A filter with negatives on one side and positives on the other is an edge detector oriented along that axis; one that is positive in the middle and negative around it is a blob detector. Nobody told the network to learn these. They are what minimising the loss produced, and they are the reason the features transfer to a task the base never saw.

Day 5 takeaway

Day 6 Autoencoders and Learned Representations

Compression without labels, denoising, and anomaly scores from error

Week 10 ended by finding unusual customers with PCA: rebuild each row from two components and flag whichever rows rebuild worst. An autoencoder is that idea with a network in place of the linear projection, and it is the bridge between supervised deep learning and everything in weeks 9 and 10.

An autoencoder on the churn data

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])
Z = prep.fit_transform(df).astype('float32')
print('encoded width %d columns' % Z.shape[1])

tf.random.set_seed(42)
auto = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Z.shape[1],)),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(2, activation='relu', name='bottleneck'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(Z.shape[1]),
])
auto.compile(optimizer='adam', loss='mse')
auto.fit(Z, Z, epochs=120, batch_size=64, verbose=0)

error = ((auto.predict(Z, verbose=0) - Z) ** 2).mean(axis=1)
print('reconstruction error: median %.4f, 99th %.4f, max %.4f'
      % (np.median(error), np.percentile(error, 99), error.max()))

worst = np.argsort(error)[:-1][:5]
print('\nthe five customers the network cannot rebuild:')
print(df.iloc[worst][['customer_id'] + NUM].to_string(index=False))
```

```
encoded width 15 columns
reconstruction error: median 0.1285, 99th 0.2909, max 29.5497
```

```
the five customers the network cannot rebuild:
customer_id  tenure_months  support_calls  monthly_charges
     C01643              1              0            50.61
     C00487              6              0            48.84
     C00988              9              0            49.83
     C00451              3              0            57.29
     C02559              8              0            48.30
```

Same idea as PCA, one important difference

PCA can only rebuild a row as a weighted sum of fixed directions. An autoencoder can bend, so in principle it captures structure that is not a flat subspace, for instance that high charges are normal *on a fibre contract* and unusual otherwise, which no single linear direction expresses. The cost is that you now have a network to train, a seed to fix, and no equivalent of explained variance to report.

Note that the customer at the top of that list is the one with 97 support calls, the same row week 10 found with PCA. Two quite different methods agreeing on the most unusual customer is the reassuring outcome.

Is the flexibility earning anything?

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore',
                                          sparse_output=False))]), CAT),
])
Z = prep.fit_transform(df).astype('float32')

```

```
pca = PCA(n_components=2, random_state=42).fit(Z)
pca_err = ((pca.inverse_transform(pca.transform(Z)) - Z) ** 2).mean()

tf.random.set_seed(42)
auto = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(Z.shape[1],)),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(2, activation='relu'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(Z.shape[1]),
])
auto.compile(optimizer='adam', loss='mse')
auto.fit(Z, Z, epochs=120, batch_size=64, verbose=0)
auto_err = float(auto.evaluate(Z, Z, verbose=0))

print('mean squared reconstruction error, 2 dimensions')
print('  PCA         %.4f' % pca_err)
print('  autoencoder %.4f' % auto_err)
```

```
mean squared reconstruction error, 2 dimensions
  PCA         0.1930
  autoencoder 0.2305
```

PCA wins. Given the same two-dimensional budget, the method with a closed form solution and no hyperparameters rebuilds these rows better than the network does, and it does so in milliseconds, deterministically, with an explained-variance figure you can put in a report.

Watch Out

Fifteen columns is not where an autoencoder shines

The flexibility that makes autoencoders powerful needs something to be flexible about. After one-hot encoding, this data is fifteen columns of largely linear structure, and a linear method models linear structure optimally by definition. Autoencoders start to win on inputs with hundreds or thousands of dimensions and strong nonlinear structure, images, audio, embeddings. Run the comparison before you assume which side you are on.

A convolutional autoencoder on the digits

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
tf.random.set_seed(42)
auto = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),                  # 4 x 4 x 16
    tf.keras.layers.Conv2D(8, 3, padding='same', activation='relu'),
    tf.keras.layers.UpSampling2D(2),                  # back to 8 x 8
    tf.keras.layers.Conv2D(1, 3, padding='same', activation='sigmoid'),
])
auto.compile(optimizer='adam', loss='mse')
auto.fit(X_tr, X_tr, epochs=60, batch_size=64, verbose=0)

rebuilt = auto.predict(X_te[:1], verbose=0)
chars = ' .:-=+*#%@'
def show(a):
    a = (a - a.min()) / (a.max() - a.min() + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

print('test error %.4f' % auto.evaluate(X_te, X_te, verbose=0))
print('\noriginal, a %d:' % y_te[0])
show(X_te[0, :, :, 0])
print('\nrebuilt from a 4 x 4 x 8 code:')
show(rebuilt[0, :, :, 0])
```

```
test error 0.0088

original, a 1:
  =@@=
  @@@.
 .@@*
 .@@#
 -@@%
  @@*
  @@*
  =@@

rebuilt from a 4 x 4 x 8 code:
  =@@=
  @@@:
  @@#
 .@@%
 .@@%
  @@*
  %@%
  =@#
```

Denoising, which is the version people actually use

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
```

```
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
rng = np.random.default_rng(0)
noisy_tr = np.clip(X_tr + rng.normal(0, 0.3, X_tr.shape), 0, 1).astype('float32')
noisy_te = np.clip(X_te + rng.normal(0, 0.3, X_te.shape), 0, 1).astype('float32')

tf.random.set_seed(42)
auto = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(1, 3, padding='same', activation='sigmoid'),
])
auto.compile(optimizer='adam', loss='mse')
# input: the corrupted image. target: the clean one.
auto.fit(noisy_tr, X_tr, epochs=60, batch_size=64, verbose=0)

cleaned = auto.predict(noisy_te[:1], verbose=0)
chars = ' .:-=+*#%@'
def show(a):
    a = (a - a.min()) / (a.max() - a.min() + 1e-9)
    for row in a:
        print(''.join(chars[min(9, int(v * 9.999))] for v in row))

print('noisy input:')
show(noisy_te[0, :, :, 0])
print('\ndenoised output:')
show(cleaned[0, :, :, 0])
print('\nclean target:')
show(X_te[0, :, :, 0])
```

```
noisy input:
 .@*+= -
 +@-%:.#
 =@@=  -
  #@*
 =@@% :#
  @*#  :
  @%@  .
  -*@

denoised output:
  %#*:
 =@#%.
 .%@=
```

```
  *@+
  %@*
  %%*
  #%%
  -%%

clean target:
  =@@=
  @@@.
 .@@*
 .@@#
 -@@%
  @@*
  @@*
  =@@
```

Nothing about the architecture changed. The only difference is what was put on each side of `fit`: corrupted input, clean target. That small substitution is the trick behind a surprising amount of modern self-supervised learning, hide part of the input, train the model to reconstruct it, and the representation you get for free turns out to be useful for tasks you have not thought of yet.

Using the bottleneck as features

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression

tf.random.set_seed(42)
encoder = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(12, activation='relu', name='code'),
])
decoder = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(12,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Reshape((8, 8, 1)),
])
```

```
auto = tf.keras.Sequential([encoder, decoder])
auto.compile(optimizer='adam', loss='mse')
auto.fit(X_tr, X_tr, epochs=80, batch_size=64, verbose=0)

# the labels were never shown to the autoencoder
code_tr = encoder.predict(X_tr, verbose=0)
code_te = encoder.predict(X_te, verbose=0)

flat = LogisticRegression(max_iter=2000).fit(
    X_tr.reshape(len(X_tr), -1), y_tr)
coded = LogisticRegression(max_iter=2000).fit(code_tr, y_tr)
print('logistic on 64 raw pixels     %.4f'
      % flat.score(X_te.reshape(len(X_te), -1), y_te))
print('logistic on 12 learned codes  %.4f'
      % coded.score(code_te, y_te))
```

```
logistic on 64 raw pixels     0.9622
logistic on 12 learned codes  0.9000
```

Twelve numbers, learned without ever seeing a label, carry most, not all, of what sixty-four pixels carry. That is the honest shape of the result: a five-fold compression costing a few points of accuracy. The reason anybody accepts that trade is the case where labels are the scarce thing: you can train the encoder on a million unlabelled images and then fit the classifier on the two hundred you could afford to label.

Day 6 takeaway

Day 7 Does Deep Learning Earn Its Place?

A complete image workflow, and the reckoning week 11 promised

Week 11 day 3 promised this reckoning. A neural network matched logistic regression on the churn data while carrying several hundred times the parameters. This week it has been winning. The difference between those two situations is the most useful thing in either week.

The complete workflow, end to end

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
import json
import time

X_fit, X_val = X_tr[:1000], X_tr[1000:]
y_fit, y_val = y_tr[:1000], y_tr[1000:]

tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(8, 8, 1)),
    tf.keras.layers.RandomTranslation(0.1, 0.1, fill_mode='constant'),
    tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Flatten(),          # day 3: position matters at 8 x 8
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(10, activation='softmax'),
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

start = time.time()
hist = model.fit(X_fit, y_fit, validation_data=(X_val, y_val),
                 epochs=200, batch_size=64, verbose=0,
                 callbacks=[tf.keras.callbacks.EarlyStopping(
                     patience=20, restore_best_weights=True)])

print('epochs run %d of 200 in %.1fs' % (len(hist.history['loss']),
                                         time.time() - start))
print('test accuracy %.4f' % model.evaluate(X_te, y_te, verbose=0)[1])

model.save('digits_cnn.keras')
json.dump({'input': [8, 8, 1], 'scaling': 'divide by 16',
           'classes': list(range(10))},
          open('digits_cnn.json', 'w'))

reloaded = tf.keras.models.load_model('digits_cnn.keras')
print('reloaded and agrees:',
      bool((reloaded.predict(X_te[:50], verbose=0).argmax(1)
            == model.predict(X_te[:50], verbose=0).argmax(1)).all()))
```

```
epochs run 67 of 200 in 10.0s
test accuracy 0.9867
reloaded and agrees: True
```

Watch Out

Save the preprocessing, not just the weights

The `.keras` file contains the architecture and the weights. It does not contain the fact that you divided by 16, or which index means which digit. Serve the model without those and it will happily return confident nonsense for inputs on the wrong scale. The small JSON file beside it is not optional, week 15 turns it into a proper contract.

What it cost against what it bought

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X = (d.images / 16.0).astype('float32')[..., np.newaxis]
y = d.target.astype('int32')
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import HistGradientBoostingClassifier, RandomForestClassifier
from sklearn.svm import SVC
import time

flat_tr = X_tr.reshape(len(X_tr), -1)
flat_te = X_te.reshape(len(X_te), -1)

print('%-24s %10s %10s' % ('', 'accuracy', 'fit time'))
models = [('logistic regression', LogisticRegression(max_iter=2000)),
          ('random forest', RandomForestClassifier(random_state=42)),
          ('gradient boosting', HistGradientBoostingClassifier(random_state=42)),
          ('support vector machine', SVC())]
for name, m in models:
    t = time.time()
    m.fit(flat_tr, y_tr)
    print('%-24s %10.4f %9.1fs'
          % (name, m.score(flat_te, y_te), time.time() - t))
```

```
                         accuracy   fit time
logistic regression        0.9622       0.1s
random forest              0.9600       0.3s
gradient boosting          0.9600       3.7s
support vector machine     0.9911       0.1s
```

A support vector machine on the raw flattened pixels, fitted in about a tenth of a second with default settings and no tuning whatsoever, beats the convolutional network that took roughly a hundred times longer to fit and needed batch normalisation, dropout, augmentation and early stopping to get there.

Watch Out

This is the result, not a rhetorical flourish

It would have been easy to leave this comparison out. The reason it is here is that the same comparison, run on 32×32 colour photographs or 224×224 medical scans, comes out the opposite way and it is not close, and the only way to know which situation you are in is to fit the cheap model and look. Eight by eight greyscale digits are, in the end, a 64-dimensional tabular problem wearing a picture's clothing.

When deep learning earns its place

Situation	Reach for	Because
Tabular data, thousands of rows	Gradient boosting	Wins on accuracy, trains in seconds, needs no scaling
Tabular data, millions of rows and rich categoricals	Boosting first, network second	Networks become competitive, but rarely by much
Images, audio, video	Convolutional network, pretrained	The structure of the data is spatial; nothing else exploits that
Text	Pretrained transformer	Week 13
Fewer than a few hundred labels	Transfer learning, or classical	There is not enough signal to fit a network from scratch
The decision must be explained	Linear or tree model	"The network said so" fails an audit
Latency budget in single milliseconds	Linear model	A dot product is hard to beat

The honest summary of two weeks

Deep learning is not a better version of machine learning. It is the answer to a specific question: *what do I do when the raw input has structure that no feature I can write down captures?* A picture has that problem. A spectrogram has it. A sentence has it. A spreadsheet of customer attributes, where somebody has already done the hard work of deciding that `tenure_months` is a column, generally does not. The feature engineering is finished before you arrive.

Before you ship an image model

1. Check the class balance and look at a sample from each class by eye. Mislabelled images are far more common than mislabelled spreadsheet rows.
2. Split before augmenting. Augmenting first and splitting afterwards puts shifted copies of the same picture on both sides, the image version of the leakage in week 4.
3. Verify that your augmentations preserve the label in your domain.
4. Start from pretrained weights unless you have a good reason not to.
5. Train the head with the base frozen, then fine-tune at a low rate, recompiling in between.
6. Read the confusion matrix and look at the actual images the model gets wrong.
7. Fit gradient boosting on the flattened pixels as a floor. If it wins, your images are probably too small or too few for a network.
8. Save the preprocessing contract alongside the weights.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. Convolution uses far fewer weights than a dense layer because:

   a) It uses integers

   b) The images are smaller

   c) The same small kernel is applied at every position, so a pattern is learned once rather than per location

   d) It skips pixels

2. Convolution is *equivariant* to translation, which means:

   a) It ignores position

   b) It requires centred images

   c) The output is unchanged when the input moves

   d) The output moves when the input moves

3. Stacking two 3×3 convolutions rather than one 5×5 gives you:

   a) The same receptive field with fewer weights and an extra non-linearity

   b) Faster inference only

   c) Identical behaviour

   d) A smaller receptive field

4. On 8×8 digits, replacing `GlobalAveragePooling2D` with `Flatten` recovered eleven points of accuracy because:

   a) Flatten has more parameters

   b) Averaging over positions discards where each response occurred, and at that resolution position is most of the signal

   c) Global pooling is buggy

   d) The model was undertrained

5. Data augmentation is best described as:

   a) A form of normalisation

   b) Getting more data

   c) Regularisation that encodes a specific belief about which transformations do not change the label

   d) A speed optimisation

6. Transfer learning failed to beat training from scratch on the course's digit experiment because:

   a) Freezing is never useful

b) The learning rate was wrong

c) The code was wrong

d) The source task was far too small for its learned features to beat what the target's own data could learn

7. After changing `base.trainable` you must:

   a) Recompile the model, or the optimiser will keep training exactly the same weights as before

   b) Reset the seed

   c) Lower the batch size

   d) Reload the data

8. A support vector machine on flattened pixels beat the convolutional network. The lesson is:

   a) Never use CNNs

   b) Fit the cheap baseline and look, at this resolution the problem is effectively tabular

   c) The CNN needed more layers

   d) Accuracy is the wrong metric

Answers:1c, 2d, 3a, 4b, 5c, 6d, 7a, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 13

Text, Embeddings and Sequence Models

Week 13

OBJECTIVES

Turn text into numbers, discover what a bag of words cannot express, and find out whether a sequence model is worth it.

This week is built around one corpus with two questions in it. The first, which category is this ticket, a bag of words answers easily. The second, is the customer's problem over. It answers worse than a coin, because the answer lives in which words sit next to which. Fixing that takes us from n-grams through embeddings and recurrent layers to attention, and to an honest account of when each is worth its cost.

Day 1 Turning Text Into Numbers

Tokens, vocabularies, sparse matrices and the information you throw away

Every model so far has been handed numbers. Text is not numbers, and the whole of this week is about the step in between, because that step, not the choice of classifier, is what decides whether the model works.

The corpus

The broadband company whose customers you have been predicting for twelve weeks has a support inbox. This is it: 4,200 tickets, each with a free-text body, a category, and a flag for whether the customer is describing a problem that is now over. Generate it the same way you generated `customers.csv` in week 1.

```
import io
import os

import numpy as np
import pandas as pd

SEED = 13
N = 4200

THING = ["router", "connection", "line", "broadband", "wifi", "signal",
         "hub", "internet"]

# The two halves of the construction. A positive state word with no negation
# means the problem is over; the same word negated means it is not.
GOOD_STATE = ["working", "stable", "fixed", "fine", "sorted", "reliable",
```

```
              "back to normal"]
BAD_STATE = ["slow", "down", "unstable", "broken", "patchy", "unreliable",
             "dropping out"]

# Every one of these appears equally often on both sides of the label.
MODIFIER = ["", "still ", "not ", "no longer "]
NEGATING = {"not ", "no longer "}          # these flip the state word

FRAME = [
    "the {thing} is {mod}{state}",
    "my {thing} is {mod}{state}",
    "it is {mod}{state}",
    "the {thing} has been {mod}{state} {when}",
    "as of this morning the {thing} is {mod}{state}",
    "{when} the {thing} is {mod}{state}",
]

WHEN = ["every evening", "since tuesday", "all weekend", "for three weeks",
        "in the mornings", "since the storm", "most nights"]

# Context sentences carry the category and nothing else -- no resolution
# information at all, so the two labels stay genuinely different tasks.
CONTEXT = {
    "technical": [
        "the engineer visited on tuesday",
        "this is the third time i have written about the hub",
        "i have restarted the equipment twice",
        "the speed test reads well under what i pay for",
        "there is a flashing red light on the box",
    ],
    "billing": [
        "this relates to invoice 40182",
        "the account is on the fibre tariff",
        "i pay by direct debit on the eighth",
        "there is a charge on the statement i do not recognise",
        "the promotional discount was supposed to run for a year",
    ],
    "cancellation": [
        "i have been a customer for six years",
        "i am considering my options",
        "my contract ends next month",
        "a competitor has offered me a better package",
        "i would like to know the termination fee",
    ],
}

GREETING = ["", "", "", "hi team, ", "hello, ", "hi, ", "morning, ", "hi there "]
SIGNOFF = ["", "", "", "", " thanks", " please advise", " regards", " ta",
           " please call me back"]

TYPOS = {"the": "teh", "and": "adn", "connection": "conection",
         "internet": "internt", "please": "plese", "invoice": "invoce"}


```

```
def _core(rng, resolved):
    """One clause whose meaning comes from the modifier-plus-state pairing.

    Pick the modifier first and independently of the label, then choose the
    state word that makes the clause mean what the label says. That ordering
    is what keeps every modifier balanced across both classes -- generate it
    the other way round and the modifier leaks the answer to a unigram model.
    """
    mod = rng.choice(MODIFIER)
    good_state = resolved if mod not in NEGATING else not resolved
    state = rng.choice(GOOD_STATE if good_state else BAD_STATE)
    return rng.choice(FRAME).format(thing=rng.choice(THING), mod=mod,
                                    state=state, when=rng.choice(WHEN))


def _rough(rng, text):
    """Put the mess back in: casing, punctuation, typos."""
    if rng.random() < 0.10:
        words = text.split()
        for i, w in enumerate(words):
            if w in TYPOS and rng.random() < 0.5:
                words[i] = TYPOS[w]
        text = " ".join(words)
    if rng.random() < 0.12:
        text = text.upper()
    elif rng.random() < 0.25:
        text = text.capitalize()
    text = rng.choice(GREETING) + text + rng.choice(SIGNOFF)
    text += rng.choice([".", ".", "", "!", "!!", "...", " ."])
    if rng.random() < 0.08:
        text = "  " + text + "  "
    return text


def build(seed=SEED, n=N):
    rng = np.random.default_rng(seed)
    cats = rng.choice(["technical", "billing", "cancellation"], size=n,
                      p=[0.52, 0.33, 0.15])
    rows = []
    for i, cat in enumerate(cats):
        resolved = int(rng.random() < 0.42)
        parts = [_core(rng, resolved)]
        # A second clause of the same polarity. Mixed polarity would leave the
        # ticket with no correct answer, so it never happens.
        if rng.random() < 0.30:
            parts.append(_core(rng, resolved))
        # The category sentence, which says nothing about resolution.
        if rng.random() < 0.75:
            where = rng.integers(0, len(parts) + 1)
            parts.insert(int(where), rng.choice(CONTEXT[cat]))
        text = _rough(rng, ". ".join(parts))
        rows.append({"ticket_id": "T%05d" % (i + 1), "text": text,
                     "category": cat, "resolved": resolved})

```

```
    df = pd.DataFrame(rows)

    # ~4% label noise, so a perfect score is impossible and anybody who gets
    # one goes looking for the leak instead of celebrating.
    flip = rng.random(len(df)) < 0.04
    df.loc[flip, "resolved"] = 1 - df.loc[flip, "resolved"]

    # a few empty and near-empty tickets
    blank = rng.choice(len(df), size=18, replace=False)
    df.loc[blank[:9], "text"] = ""
    df.loc[blank[9:], "text"] = rng.choice(["?", "...", "help", "hi", "!!"],
                                           size=len(blank) - 9)

    # double submissions
    dupes = df.iloc[rng.choice(len(df), size=90, replace=False)]
    df = pd.concat([df, dupes], ignore_index=True)
    df = df.iloc[rng.permutation(len(df))].reset_index(drop=True)
    return df

build().to_csv('tickets.csv', index=False, encoding='utf-8')
```

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
print('tickets %d' % len(tickets))
print(tickets[['ticket_id', 'category', 'resolved']].head())
print('\nresolved rate %.3f' % tickets['resolved'].mean())
print('categories:')
print(tickets['category'].value_counts())
```

```
tickets 4200
  ticket_id      category  resolved
0    T02210       billing         0
1    T00107       billing         1
2    T00680       billing         1
3    T03436  cancellation         0
4    T02257       billing         1

resolved rate 0.425
categories:
category
technical       2144
billing         1429
cancellation     627
Name: count, dtype: int64
```

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
```

```
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
for _, r in tickets.head(6).iterrows():
    print('[%-12s resolved=%d]  %s'
          % (r['category'], r['resolved'], r['text'][:78]))
```

```
[billing      resolved=0]  AS OF THIS MORNING THE HUB IS PATCHY. MOST NIGHTS THE HUB IS NO
    ↪ LONGER WORKING
[billing      resolved=1]  hello, my hub is not unstable. my wifi is still fixed.
[billing      resolved=1]  hello, There is a charge on the statement i do not recognise. it
    ↪ is working. t
[cancellation resolved=0]  the hub is down please advise
[billing      resolved=1]  hi, the account is on the fibre tariff. as of this morning the
    ↪ router is no lo
[billing      resolved=0]  in the mornings the wifi is unstable please advise .
```

Watch Out

Look at the raw strings before you clean anything

Shouting in capitals, doubled exclamation marks, leading spaces, typos, and nine tickets that are empty. Every one of those is in the file on purpose, because every one of them is in real inboxes. The instinct to normalise it all away immediately is the right instinct and the wrong first move, some of that mess is signal. Angry customers do write in capitals.

Tokenising

```
text = '  THE ROUTER is Not working!! since tuesday.  '

print('raw        %r' % text)
print('stripped   %r' % text.strip())
print('lowered    %r' % text.strip().lower())
print('naive split %s' % text.strip().lower().split())

import re
tokens = re.findall(r"[a-z0-9']+", text.lower())
print('regex      %s' % tokens)
```

```
raw        '  THE ROUTER is Not working!! since tuesday.  '
stripped   'THE ROUTER is Not working!! since tuesday.'
lowered    'the router is not working!! since tuesday.'
naive split ['the', 'router', 'is', 'not', 'working!!', 'since', 'tuesday.']
regex      ['the', 'router', 'is', 'not', 'working', 'since', 'tuesday']
```

`split()` leaves `working!!` and `tuesday.` as tokens distinct from `working` and `tuesday`, which quietly doubles parts of your vocabulary and halves the evidence for each. The regex keeps letters, digits and apostrophes and drops the rest.

Building a bag of words by hand

```
import re
from collections import Counter
```

```

docs = ['the router is not working',
        'the router is working',
        'my line is not slow']

tokenised = [re.findall(r"[a-z0-9']+", d.lower()) for d in docs]
vocab = sorted({t for doc in tokenised for t in doc})
index = {t: i for i, t in enumerate(vocab)}
print('vocabulary (%d): %s' % (len(vocab), vocab))

import numpy as np
M = np.zeros((len(docs), len(vocab)), dtype=int)
for r, doc in enumerate(tokenised):
    for token, n in Counter(doc).items():
        M[r, index[token]] = n

print('\ndocument-term matrix:')
print('      ' + ' '.join('%-8s' % v[:7] for v in vocab))
for r, row in enumerate(M):
    print('doc %d ' % r + ' '.join('%-8d' % v for v in row))
```

```
vocabulary (8): ['is', 'line', 'my', 'not', 'router', 'slow', 'the', 'working']

document-term matrix:
      is       line     my       not      router   slow     the      working
doc 0 1        0        0        1        1        0        1        1
doc 1 1        0        0        0        1        0        1        1
doc 2 1        1        1        1        0        1        0        0
```

Read rows 0 and 1 again

They differ by one word and mean opposite things. In this representation they differ by one column out of eight, and nothing records that `not` was sitting immediately before `working`. Every limitation in the next two days follows from that single fact.

The same thing from scikit-learn

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
M = vec.fit_transform(tickets['text'])
print('matrix %s, %.4f%% non-zero'
      % (M.shape, 100 * M.nnz / (M.shape[0] * M.shape[1])))
print('stored as %s' % type(M).__name__)

counts = np.asarray(M.sum(axis=0)).ravel()
```

```
names = vec.get_feature_names_out()
print('\nmost common tokens:')
for i in counts.argsort()[:-1][:12]:
    print('  %-14s %5d' % (names[i], counts[i]))
```

```
matrix (4200, 130), 10.9416% non-zero
stored as csr_matrix

most common tokens:
  help              3
  invoce            6
  internt          21
  conection        26
  competitor       83
  better           83
  package          83
  offered          83
  considering      85
  options          85
  am               85
  customer         92
```

Day 1 takeaway

Day 2 TF-IDF, n-grams and What Order Costs

Two tasks in one corpus: one a bag of words solves, one it cannot

Two tasks live in this corpus. One is to say which category a ticket belongs to; the other is to say whether the customer's problem is resolved. They look equally easy. They are not, and the difference is the most useful thing in this week.

TF-IDF

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.feature_extraction.text import TfidfVectorizer

vec = TfidfVectorizer()
M = vec.fit_transform(tickets['text'])
names = vec.get_feature_names_out()

row = M[0].toarray().ravel()
print('ticket: %s' % tickets['text'][0][:70])
print('\nhighest weighted terms in it:')
for i in row.argsort()[:-1][:6]:
    if row[i] > 0:
```

```
        print('  %-14s %.3f  (document frequency %.3f)'
              % (names[i], row[i],
                 (M[:, i] > 0).sum() / M.shape[0]))
```

```
ticket: AS OF THIS MORNING THE HUB IS PATCHY. MOST NIGHTS THE HUB IS NO LONGER

highest weighted terms in it:
```

Task one: which category?

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split

Xc_tr, Xc_te, yc_tr, yc_te = train_test_split(
    tickets['text'], tickets['category'], test_size=0.25,
    stratify=tickets['category'], random_state=42)

model = make_pipeline(TfidfVectorizer(),
                      LogisticRegression(max_iter=2000)).fit(Xc_tr, yc_tr)
print('category accuracy %.4f' % model.score(Xc_te, yc_te))
print('always guessing the commonest: %.4f'
      % (yc_te == yc_tr.mode()[0]).mean())
```

```
category accuracy 0.8400
always guessing the commonest: 0.5105
```

A bag of words handles this comfortably. Words like *invoice*, *tariff* and *termination* belong to one category and nowhere else, so their mere presence settles the question and their position in the sentence is irrelevant.

Task two: is the problem resolved?

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
```

```
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics import roc_auc_score

model = make_pipeline(TfidfVectorizer(),
                      LogisticRegression(max_iter=2000)).fit(X_tr, y_tr)
print('accuracy   %.4f' % model.score(X_te, y_te))
print('ROC AUC    %.4f'
      % roc_auc_score(y_te, model.predict_proba(X_te)[:, 1]))
print('\nalways predicting unresolved: %.4f' % (1 - y_te.mean()))
```

```
accuracy   0.5597
ROC AUC    0.5297

always predicting unresolved: 0.5750
```

Watch Out

It is worse than a constant

The model that reads the ticket does *worse* than the model that ignores it and always says unresolved. This is not a bug, a bad hyperparameter or too little data. Something about the combination of this representation and this model cannot express the answer, and working out which of the two is at fault is the whole of the rest of this day.

Why, in one table

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
import re

def has(word):
    return tickets['text'].str.lower().str.contains(
        r'\b%s\b' % word, regex=True)

print('%-14s %10s %10s %12s' % ('token', 'appears', 'resolved', 'unresolved'))
for word in ['not', 'still', 'no', 'longer', 'working', 'slow',
             'fixed', 'broken']:
    m = has(word)
    print('%-14s %10d %10d %12d'
          % (word, m.sum(), (m & (tickets['resolved'] == 1)).sum(),
```

```
            (m & (tickets['resolved'] == 0)).sum()))
```

```
token              appears   resolved   unresolved
not                   1346        568          778
still                 1331        562          769
no                    1273        555          718
longer                1273        555          718
working                382        168          214
slow                   366        154          212
fixed                  379        165          214
broken                 357        154          203
```

Every one of those tokens is split roughly evenly between the two classes. Alone, each is worth nothing. A linear model assigns one weight per token and adds them up, so it has to commit to a single number for *not*, and no single number can be right when *not working* means one thing and *not slow* means the opposite.

So whose fault is it, the features or the model?

There are two ways out of an interaction, and they are worth separating because they lead to different code. Either hand the model the interaction as a feature, or use a model that finds interactions by itself. Try the second one first, keeping the unigrams exactly as they are.

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import FunctionTransformer

# HistGradientBoosting needs a dense array, and this vocabulary is
# small enough that densifying it is safe. On a real corpus it is not.
densify = FunctionTransformer(lambda M: np.asarray(M.todense()),
                              accept_sparse=True)

print('unigram counts only -- no bigrams anywhere')
print('%-32s %10s' % ('model', 'accuracy'))
```

```
for name, clf, dense in [
        ('logistic regression (linear)',
         LogisticRegression(max_iter=2000), False),
        ('gradient boosting',
         HistGradientBoostingClassifier(random_state=42), True),
        ('neural net (32, 16)',
         MLPClassifier((32, 16), max_iter=800, random_state=42), False)]:
    steps = [CountVectorizer()] + ([densify] if dense else []) + [clf]
    m = make_pipeline(*steps).fit(X_tr, y_tr)
    print('%-32s %10.4f' % (name, m.score(X_te, y_te)))
print('%-32s %10.4f' % ('always predicting unresolved', 1 - y_te.mean()))
```

```
unigram counts only -- no bigrams anywhere
model                              accuracy
logistic regression (linear)         0.5664
gradient boosting                    0.9026
neural net (32, 16)                  0.8672
always predicting unresolved         0.5750
```

The information was there the whole time

Nothing about the features changed. The same unigram counts that left logistic regression below a constant prediction take gradient boosting most of the way to the ceiling, because a tree can split on *not* and then split again on *slow* inside that branch, which is precisely what representing an interaction means.

So the bag of words did *not* throw away the answer. It threw away the *convenience* of the answer, and a linear model is a model that can only use convenient answers. Keep the two ideas separate when you diagnose your own models: *the features do not contain it* and *my model cannot reach it* look identical from the accuracy score and call for opposite fixes.

The other way out: n-grams

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
```

```
from sklearn.metrics import roc_auc_score

print('%-16s %10s %10s %10s' % ('features', 'columns', 'accuracy', 'auc'))
for label, rng in [('unigrams', (1, 1)), ('1-2 grams', (1, 2)),
                   ('1-3 grams', (1, 3)), ('2-grams only', (2, 2))]:
    m = make_pipeline(TfidfVectorizer(ngram_range=rng),
                      LogisticRegression(max_iter=2000)).fit(X_tr, y_tr)
    print('%-16s %10d %10.4f %10.4f'
          % (label, len(m[0].vocabulary_), m.score(X_te, y_te),
             roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])))
```

```
features            columns   accuracy        auc
unigrams                130     0.5597     0.5297
1-2 grams              1079     0.9542     0.9591
1-3 grams              4139     0.9532     0.9589
2-grams only            949     0.9599     0.9605
```

What the bigram model learned

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                  LogisticRegression(max_iter=2000)).fit(X_tr, y_tr)
names = m[0].get_feature_names_out()
coef = m[1].coef_.ravel()

print('strongest evidence the problem IS resolved:')
for i in coef.argsort()[:-1][:8]:
    print('  %-24s %+.3f' % (names[i], coef[i]))
print('\nstrongest evidence it is NOT:')
for i in coef.argsort()[:8]:
    print('  %-24s %+.3f' % (names[i], coef[i]))
```

```
strongest evidence the problem IS resolved:
  not stable               -3.788
  longer fine              -3.734
  longer fixed             -3.733
  still dropping           -3.729
  not sorted               -3.661
  longer back              -3.597
  still unreliable         -3.586
  not back                 -3.507

strongest evidence it is NOT:
  not stable               -3.788
  longer fine              -3.734
  longer fixed             -3.733
  still dropping           -3.729
  not sorted               -3.661
  longer back              -3.597
  still unreliable         -3.586
  not back                 -3.507
```

This is what a good feature looks like

The model has recovered the construction the corpus was built from, and it did so without being told. Bigrams sit at the top of both lists and single words sit near zero. If you ever want to know whether a text representation is working, print the coefficients and see whether they read like something a person would say.

Notice too that bigrams plus a linear model beat gradient boosting on unigrams. Giving the model the *right* interaction directly beats asking it to search for interactions in general, which is week 8's argument, arriving here in a different costume. Feature engineering is not what you do when you cannot afford a better model; it is how you tell the model what you already know.

> **Day 2 takeaway**

Day 3 Embeddings

Dense vectors with a geometry, and what they do and do not fix

Bigrams worked, and they scale badly: every pair you keep is a new column, trigrams are worse, and none of it generalises. A model that has learned *not working* knows nothing about *not functioning*. Embeddings are the alternative.

The problem with one column per word

```python
import numpy as np

vocab = ['router', 'hub', 'invoice', 'working', 'broken']
eye = np.eye(len(vocab), dtype=int)
print('one-hot:')
for w, row in zip(vocab, eye):
    print('  %-9s %s' % (w, row))
```

```

print('\ncosine similarity between every pair: ', end='')
sims = {round(float(a @ b), 3) for i, a in enumerate(eye)
        for j, b in enumerate(eye) if i != j}
print(sims)
print('router and hub are exactly as similar as router and invoice.')
```

```
one-hot:
  router    [1 0 0 0 0]
  hub       [0 1 0 0 0]
  invoice   [0 0 1 0 0]
  working   [0 0 0 1 0]
  broken    [0 0 0 0 1]

cosine similarity between every pair: {0.0}
router and hub are exactly as similar as router and invoice.
```

Learning one

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
vec = tf.keras.layers.TextVectorization(max_tokens=400,
                                        output_sequence_length=24)
vec.adapt(X_tr.to_numpy())

print('vocabulary size %d' % vec.vocabulary_size())
print('first 12 tokens: %s' % vec.get_vocabulary()[:12])

example = tf.constant(['the router is not working',
                       'the router is working'])
print('\nencoded:')
print(vec(example).numpy()[:, :10])
```

```
vocabulary size 134
first 12 tokens: ['', '[UNK]', np.str_('the'), np.str_('is'), np.str_('i'), np.str_('hi'),
    ↪ np.str_('not'), np.str_('morning'), np.str_('this'), np.str_('still'), np.str_('no'),
    ↪ np.str_('longer')]

encoded:
[[ 2 30  3  6 52  0  0  0  0  0]
 [ 2 30  3 52  0  0  0  0  0  0]]
```

Watch Out

Index 0 is padding and index 1 is the unknown token

`TextVectorization` reserves both. Every sequence is padded with zeros to a fixed length so it can sit in a rectangular array, and any word not in the vocabulary becomes `[UNK]`. Forgetting that index 0 is not a real word is how people end up with an embedding for padding that quietly dominates their averages.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
vec = tf.keras.layers.TextVectorization(max_tokens=400,
                                        output_sequence_length=24)
vec.adapt(X_tr.to_numpy())

tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(1,), dtype=tf.string),
    vec,
    tf.keras.layers.Embedding(vec.vocabulary_size(), 16, mask_zero=True),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
```

```
])
model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit(X_tr.to_numpy(), y_tr.to_numpy(), epochs=25, batch_size=64,
          verbose=0)
print('accuracy %.4f'
      % model.evaluate(X_te.to_numpy(), y_te.to_numpy(), verbose=0)[1])
```

```
accuracy 0.9188
```

Averaging the embeddings discards the order just as thoroughly as `CountVectorizer` did, two tickets with the same words in a different arrangement produce an identical average. What rescues the score is the `Dense` layer sitting after the average, which is nonlinear and can therefore represent the interaction, exactly as gradient boosting did on day 2. It lands in the same region as that model, and behind bigrams, for the same reason: it has to discover the pairing rather than being handed it.

The geometry it learned

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
vec = tf.keras.layers.TextVectorization(max_tokens=400,
                                        output_sequence_length=24)
vec.adapt(tickets['text'].to_numpy())

tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(1,), dtype=tf.string),
    vec,
    tf.keras.layers.Embedding(vec.vocabulary_size(), 16),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax'),
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
codes = {'billing': 0, 'cancellation': 1, 'technical': 2}
model.fit(tickets['text'].to_numpy(),
          tickets['category'].map(codes).to_numpy(),
          epochs=30, batch_size=64, verbose=0)

E = model.layers[1].get_weights()[0]
words = vec.get_vocabulary()
```

```
idx = {w: i for i, w in enumerate(words)}
norm = E / (np.linalg.norm(E, axis=1, keepdims=True) + 1e-9)

def nearest(word, k=4):
    if word not in idx:
        return '(not in vocabulary)'
    sims = norm @ norm[idx[word]]
    order = [i for i in sims.argsort()[:-1] if i != idx[word]][:k]
    return ', '.join('%s %.2f' % (words[i], sims[i]) for i in order)

for w in ['invoice', 'router', 'termination', 'engineer']:
    print('%-13s → %s' % (w, nearest(w)))
```

```
invoice       → regards -0.86, have -0.81, conection -0.81, internt -0.72
router        → considering -0.84, six -0.84, years -0.83, options -0.83
termination   → unreliable -0.84, on -0.79, router -0.77, pay -0.76
engineer      → fine -0.91, fibre -0.74, account -0.74, to -0.73
```

The neighbours come from the task, not from English

These vectors were trained to predict the ticket category, so words land near each other when they push the category the same way. That is not the same as meaning similarity, and it is why the embedding you train on 3,000 tickets is far less useful than one trained on billions of words. The latter has seen enough context to learn something closer to actual usage. Day 6 returns to this.

Representation	Columns	Order?	Generalises to unseen words?
One-hot / bag of words	One per vocabulary word	No	No
TF-IDF	Same, reweighted	No	No
n-grams	Explodes with n	Locally	No
Learned embedding	Chosen, e.g. 64	Not by itself	Only via subwords
Pretrained embedding	Chosen, e.g. 300	Not by itself	Yes, if the word was in the source corpus

Day 3 takeaway

Day 4 Recurrent Networks

State across a sequence, vanishing gradients, LSTM, padding and masking

A recurrent layer walks the sequence one token at a time, carrying a state that it updates at every step. That state is what lets it know *not* came before *working*.

A recurrent cell, in numpy

```
import numpy as np

rng = np.random.default_rng(0)
```

```
d_in, d_hidden = 4, 3
Wx = rng.normal(0, 0.5, (d_in, d_hidden))
Wh = rng.normal(0, 0.5, (d_hidden, d_hidden))
b = np.zeros(d_hidden)

def run(sequence):
    h = np.zeros(d_hidden)
    for x in sequence:
        h = np.tanh(x @ Wx + h @ Wh + b)    # the whole recurrence
    return h

a = rng.normal(size=(1, d_in))
c = rng.normal(size=(1, d_in))
print('state after [a, b] %s' % run(np.vstack([a, c])).round(3))
print('state after [b, a] %s' % run(np.vstack([c, a])).round(3))
print('\nsame two inputs, different order, different state.')
print('that is the entire reason this layer exists.')
```

```
state after [a, b] [0.038 0.046 0.361]
state after [b, a] [0.172 0.144 0.398]

same two inputs, different order, different state.
that is the entire reason this layer exists.
```

Why plain recurrence fails on long sequences

```
import numpy as np

print('%8s %16s %16s' % ('steps', 'weight 0.5', 'weight 1.5'))
for steps in [5, 10, 25, 50, 100]:
    print('%8d %16.6g %16.6g'
          % (steps, 0.5 ** steps, 1.5 ** steps))
print('\nthe gradient is a product of one factor per step.')
```

```
   steps       weight 0.5       weight 1.5
       5          0.03125          7.59375
      10      0.000976562           57.665
      25      2.98023e-08          25251.2
      50      8.88178e-16      6.37622e+08
     100      7.88861e-31      4.06561e+17

the gradient is a product of one factor per step.
```

What LSTM and GRU actually changed

Both add a path along which the state is *added to* rather than repeatedly multiplied, plus learned gates that decide what to keep, what to forget and what to output. The additive path is the important part: a sum does not shrink geometrically, so gradients survive far more steps. GRU is the same idea with two gates instead of three, fewer parameters, usually indistinguishable results.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
```

```
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
for name, layer in [('SimpleRNN', tf.keras.layers.SimpleRNN(32)),
                    ('GRU', tf.keras.layers.GRU(32)),
                    ('LSTM', tf.keras.layers.LSTM(32))]:
    layer.build((None, 20, 16))
    print('%-11s %6d parameters'
          % (name, sum(int(np.prod(w.shape)) for w in layer.weights)))
print('\nfor the same 16-wide input and 32-wide state.')
print('LSTM has four internal transforms, GRU three, SimpleRNN one.')
```

```
SimpleRNN     1568 parameters
GRU           4800 parameters
LSTM          6272 parameters

for the same 16-wide input and 32-wide state.
LSTM has four internal transforms, GRU three, SimpleRNN one.
```

Padding, masking and why they matter

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
vec = tf.keras.layers.TextVectorization(max_tokens=400,
                                        output_sequence_length=20)
vec.adapt(X_tr.to_numpy())

sample = tf.constant(['the hub is broken', X_tr.iloc[0]])
encoded = vec(sample).numpy()
print('short ticket encoded:')
print(encoded[0])
```

```
print('%d real tokens, %d padding zeros'
      % ((encoded[0] > 0).sum(), (encoded[0] == 0).sum()))

emb = tf.keras.layers.Embedding(vec.vocabulary_size(), 8, mask_zero=True)
out = emb(encoded)
mask = emb.compute_mask(encoded)
print('\nmask for the short ticket:')
print(mask.numpy()[0])
print('False means "ignore this step" for every layer downstream.')
```

```
short ticket encoded:
[ 2 24  3 59  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
4 real tokens, 16 padding zeros

mask for the short ticket:
[ True  True  True  True False False False False False False False False
 False False False False False False False False]
False means "ignore this step" for every layer downstream.
```

Watch Out

`mask_zero=True` is not the default

Without it, a recurrent layer processes twelve padding tokens after your eight real ones and the final state is whatever those zeros left behind. Average pooling is worse: it divides by the padded length, so a four-word ticket has its signal diluted by sixteen zeros. Short documents suffer most, which is precisely the population you are usually worst at already.

The clearest way to see what masking buys you is to change nothing but the padded length. The words are the same, so the pooled vector representing them should be the same.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
```

```
    stratify=tickets['resolved'], random_state=42)
text = tf.constant(['the hub is broken'])

# Seed the initialiser itself. tf.random.set_seed does not control the
# seed generator Keras uses for layer weights, so without this the two
# lengths get different embeddings and the comparison measures nothing.
init = tf.keras.initializers.RandomUniform(-0.05, 0.05, seed=0)

for mask_zero in [False, True]:
    pooled = {}
    for length in [8, 40]:
        vec = tf.keras.layers.TextVectorization(
            max_tokens=400, output_sequence_length=length)
        vec.adapt(X_tr.to_numpy())
        e = vec(text)
        emb = tf.keras.layers.Embedding(vec.vocabulary_size(), 8,
                                        embeddings_initializer=init,
                                        mask_zero=mask_zero)
        pooled[length] = tf.keras.layers.GlobalAveragePooling1D()(
            emb(e), mask=emb.compute_mask(e) if mask_zero else None)
    same = float(tf.reduce_max(tf.abs(pooled[8] - pooled[40])))
    print('mask_zero=%-6s  padded to 8 vs to 40: largest difference %.4f'
          % (mask_zero, same))
```

```
mask_zero=False   padded to 8 vs to 40: largest difference 0.0163
mask_zero=True    padded to 8 vs to 40: largest difference 0.0000
```

With masking, the padded length makes no difference at all: the layer averages four vectors either way. Without it, the answer changes when you change a number that is supposed to be an implementation detail. Note what is *not* happening in the unmasked case: index 0 has its own embedding vector like any other token, so the padding does not dilute the average towards zero, it drags it towards whatever the padding vector happens to be. On a corpus of mostly-short documents, that vector ends up carrying a large share of your representation.

The bug masking gives you for free

Masking has one failure mode, and this corpus contains it. A document with no tokens encodes to nothing but padding, so the mask is `False` at every position and average pooling divides a sum of nothing by a count of nothing.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
vec = tf.keras.layers.TextVectorization(max_tokens=50,
                                        output_sequence_length=8)
vec.adapt(tf.constant(['the router is broken', 'the invoice is wrong']))

batch = vec(tf.constant(['the router is broken', '', '...']))
emb = tf.keras.layers.Embedding(vec.vocabulary_size(), 4, mask_zero=True)
e = emb(batch)
```

```
mask = emb.compute_mask(batch)
pooled = tf.keras.layers.GlobalAveragePooling1D()(e, mask=mask)

print('encoded:')
for text, row in zip(['the router is broken', '(empty)', '...'],
                     batch.numpy()):
    print('  %-22s %s' % (text, row))
print('\npooled vectors:')
print(pooled.numpy().round(3))
print('\nany NaN?', bool(np.isnan(pooled.numpy()).any()))
```

```
encoded:
  the router is broken   [2 5 3 7 0 0 0 0]
  (empty)                [0 0 0 0 0 0 0 0]
  ...                    [0 0 0 0 0 0 0 0]

pooled vectors:
[[ 0.006 -0.     0.015 -0.007]
 [   nan    nan    nan    nan]
 [   nan    nan    nan    nan]]

any NaN? True
```

Watch Out

Non-empty is not the same as non-empty

Row three is the important one. `'...'` is a perfectly good string. It passes any `len(text) > 0` check you write, but `TextVectorization` strips punctuation by default, so it tokenises to nothing and produces the same NaN as the genuinely empty row. The guard has to be on the token count, not on the string length.

And the NaN does not raise. It propagates through the dense layers, turns the loss into `nan`, and makes every gradient after it `nan` too, so one such document destroys an entire training run and the error, if it ever surfaces, points somewhere else. This is why the split used from day 2 onwards filters on `[A-Za-z0-9]` rather than on length.

Day 4 takeaway

Day 5 The Honest Comparison

LSTM against bigrams, measured on accuracy, time and everything else

Everything is now on the table: a bag of words, bigrams, averaged embeddings and a recurrent layer that genuinely reads the sequence. Time to run them against each other on the task that needs order, and to be honest about the result.

The contenders

```
import os
```

```
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
tf.get_logger().setLevel('ERROR')
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
import time
from sklearn.metrics import roc_auc_score

def build(kind, vec):
    layers = [tf.keras.layers.Input(shape=(1,), dtype=tf.string), vec,
              tf.keras.layers.Embedding(vec.vocabulary_size(), 24,
                                        mask_zero=True)]
    if kind == 'average':
        layers.append(tf.keras.layers.GlobalAveragePooling1D())
    elif kind == 'lstm':
        layers.append(tf.keras.layers.LSTM(32))
    elif kind == 'bilstm':
        layers.append(tf.keras.layers.Bidirectional(
            tf.keras.layers.LSTM(32)))
    elif kind == 'gru':
        layers.append(tf.keras.layers.GRU(32))
    layers += [tf.keras.layers.Dense(16, activation='relu'),
               tf.keras.layers.Dense(1, activation='sigmoid')]
    return tf.keras.Sequential(layers)

xs, ys = X_tr.to_numpy(), y_tr.to_numpy()
xv, yv = X_te.to_numpy(), y_te.to_numpy()

print('%-12s %10s %10s %10s' % ('model', 'accuracy', 'auc', 'fit time'))
for kind in ['average', 'gru', 'lstm', 'bilstm']:
    tf.random.set_seed(42)
    vec = tf.keras.layers.TextVectorization(max_tokens=400,
                                            output_sequence_length=24)
    vec.adapt(xs)
    m = build(kind, vec)
    m.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
    t = time.time()
    m.fit(xs, ys, epochs=25, batch_size=64, verbose=0)
    took = time.time() - t
    proba = m.predict(xv, verbose=0).ravel()
    print('%-12s %10.4f %10.4f %9.1fs'
          % (kind, ((proba > 0.5).astype(int) == yv).mean(),
             roc_auc_score(yv, proba), took))
```

```
model          accuracy        auc   fit time
average          0.9179     0.9049       3.0s
gru              0.9608     0.9610      10.1s
lstm             0.9570     0.9601      10.4s
bilstm           0.9561     0.9614      16.0s
```

Against the model from day 2

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
import time
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics import roc_auc_score

t = time.time()
m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                  LogisticRegression(max_iter=2000)).fit(X_tr, y_tr)
took = time.time() - t
proba = m.predict_proba(X_te)[:, 1]
print('tf-idf bigrams + logistic regression')
print('  accuracy %.4f' % m.score(X_te, y_te))
print('  auc      %.4f' % roc_auc_score(y_te, proba))
print('  fit time %.2fs' % took)
print('  parameters: %d coefficients' % m[1].coef_.size)
```

```
tf-idf bigrams + logistic regression
  accuracy 0.9542
```

```
auc      0.9591
fit time 0.07s
parameters: 1079 coefficients
```

Read this before you reach for an LSTM

Both approaches solve the task. One of them took a fraction of a second, fits in a line, produces coefficients you can read out loud, and has no seed, no epoch count and no learning rate to get wrong. The other needed an embedding width, a state width, a sequence length, a batch size and twenty-five epochs, and its parameters mean nothing to anybody.

On four thousand short tickets that is not a close call. Sequence models start earning their keep when documents are long, when the dependency spans more than two or three words, or when you are starting from a model pretrained on far more text than you have, which is the subject of day 6.

Where each one actually fails

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.model_selection import train_test_split

# Drop tickets with no word characters at all -- the empty ones, and the
# ones that are only punctuation. Both tokenise to nothing, and day 4
# shows what an all-padding sequence does to masked pooling.
tickets = tickets[tickets['text'].str.contains(r'[A-Za-z0-9]',
                                               regex=True)].reset_index(drop=True)

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['resolved'], test_size=0.25,
    stratify=tickets['resolved'], random_state=42)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                  LogisticRegression(max_iter=2000)).fit(X_tr, y_tr)
proba = m.predict_proba(X_te)[:, 1]
pred = (proba > 0.5).astype(int)
wrong = np.where(pred != y_te.to_numpy())[0]

print('%d wrong out of %d' % (len(wrong), len(y_te)))
print('\nthe six it was most confident about, and wrong:')
conf = sorted(wrong, key=lambda i: abs(proba[i] - 0.5), reverse=True)[:6]
for i in conf:
    print('  truth=%d p=%.2f  %s'
          % (y_te.iloc[i], proba[i], X_te.iloc[i][:64]))
```

```
48 wrong out of 1047

the six it was most confident about, and wrong:
  truth=1 p=0.11  the engineer visited on tuesday. my connection is still patchy.
  truth=1 p=0.12  i am considering my options. the router is not reliable. my sign
  truth=1 p=0.14  hello, THE ENGINEER VISITED ON TUESDAY. IT IS NOT RELIABLE. THE
  truth=1 p=0.15  My connection is still unreliable. there is a charge on the stat
  truth=1 p=0.15  as of this morning the connection is still slow. the connection
  truth=1 p=0.15  my hub is broken. the broadband is still unstable. this relates
```

Some of those are the four percent of labels the generator deliberately flipped, and no model can get them right, which is the point of putting them there. Reading your confident errors is the fastest way to tell an irreducible label problem from a fixable modelling one.

Day 5 takeaway

Day 6 Attention and the Modern Landscape

Self-attention from scratch, subword tokens, and what actually wins

Recurrence has a structural problem that no amount of gating fixes: it is sequential. Token 500 cannot be processed until token 499 has been, which means a recurrent model cannot use a GPU's parallelism over the length of the sequence. Attention removes that constraint, and removing it is what made models trained on the whole internet possible.

Attention, in about fifteen lines

```
import numpy as np

def softmax(z):
    z = z - z.max(axis=-1, keepdims=True)
    e = np.exp(z)
    return e / e.sum(axis=-1, keepdims=True)

def attention(Q, K, V):
    scores = Q @ K.T / np.sqrt(K.shape[-1])    # who is relevant to whom
    weights = softmax(scores)                  # rows sum to 1
    return weights @ V, weights                # weighted average of values

rng = np.random.default_rng(0)
tokens = ['the', 'router', 'is', 'not', 'working']
X = rng.normal(size=(len(tokens), 8))

# with random projections the pattern is random; the point here is shape
Wq, Wk, Wv = (rng.normal(0, 0.5, (8, 8)) for _ in range(3))
out, weights = attention(X @ Wq, X @ Wk, X @ Wv)

print('output shape %s, one vector per token' % (out.shape,))
print('\nattention weights (row = token doing the looking):')
print('%-9s %s' % ('', ' '.join('%7s' % t for t in tokens)))
for t, row in zip(tokens, weights):
```

```
    print('%-9s %s' % (t, ' '.join('%7.3f' % v for v in row)))
print('\nevery row sums to 1: %s' % np.allclose(weights.sum(axis=1), 1))
```

```
output shape (5, 8), one vector per token

attention weights (row = token doing the looking):
              the  router      is     not working
the         0.122   0.088   0.434   0.248   0.108
router      0.614   0.012   0.027   0.010   0.338
is          0.302   0.145   0.151   0.188   0.214
not         0.225   0.159   0.237   0.103   0.276
working     0.446   0.012   0.224   0.044   0.274

every row sums to 1: True
```

Watch Out

Those weights are meaningless, and that is deliberate

The projections here are random, so the pattern shows nothing about language, only the mechanism and the shapes. In a trained model the row for *not* would put weight on *working*, which is how attention captures exactly the dependency that defeated the unigram model on day 2. Published attention heat-maps are trained weights, and reading them as explanations is a well-documented trap: attention shows what the model looked at, not why it decided.

Why the position has to be added back

```
import numpy as np

def softmax(z):
    z = z - z.max(axis=-1, keepdims=True)
    e = np.exp(z)
    return e / e.sum(axis=-1, keepdims=True)

def attention(Q, K, V):
    return softmax(Q @ K.T / np.sqrt(K.shape[-1])) @ V

rng = np.random.default_rng(0)
X = rng.normal(size=(4, 6))
Wq, Wk, Wv = (rng.normal(0, 0.5, (6, 6)) for _ in range(3))

a = attention(X @ Wq, X @ Wk, X @ Wv)
shuffled = X[[2, 0, 3, 1]]
b = attention(shuffled @ Wq, shuffled @ Wk, shuffled @ Wv)

print('shuffle the tokens and the outputs are the same rows, reordered:')
print(np.allclose(a[[2, 0, 3, 1]], b))
print('\nso attention alone is order-blind. positional encodings are')
print('added to the embeddings precisely to fix this.')
```

```
shuffle the tokens and the outputs are the same rows, reordered:
```

```
True

so attention alone is order-blind. positional encodings are
added to the embeddings precisely to fix this.
```

The landscape, honestly

Approach	Needs	Use when
TF-IDF + linear	Minutes, a laptop	Short texts, a clear vocabulary signal. Always try first.
Embedding + pooling	A GPU is optional	Large corpus, many classes, vocabulary too large for n-grams
LSTM / GRU	Patience	Long-range order matters and you cannot use a pretrained model
Pretrained transformer, frozen	A download	Small labelled set; use the embeddings as features
Pretrained transformer, fine-tuned	A GPU, hours	A few thousand labels and accuracy that justifies the cost
A hosted large language model	An API budget	Few or no labels, or the task is genuinely open-ended

Where week 12's lesson repeats itself

The reason transformers dominate text is the same reason convolutional networks dominate images: the architecture matches the structure of the data, and there is enough data to fill it. A transformer trained from scratch on your four thousand tickets will lose to logistic regression, comfortably. A transformer pretrained on a trillion tokens and fine-tuned on your four thousand tickets will win. The architecture is not the thing that wins, the pretraining is.

Subword tokenisation, and why it exists

```python
# Word-level vocabularies break on anything they have not seen.
vocab = {'the', 'router', 'is', 'not', 'working', 'slow'}
for sentence in ['the router is not working',
                 'the routers are misconfigured']:
    tokens = sentence.split()
    marked = [t if t in vocab else '[UNK]' for t in tokens]
    print('%-32s → %s' % (sentence, ' '.join(marked)))

print('\nsubword tokenisers split unknown words into known pieces:')
print('  routers        → router + s')
print('  misconfigured  → mis + config + ured')
print('so the vocabulary is finite and nothing is ever fully unknown.')
```

```
the router is not working        → the router is not working
the routers are misconfigured    → the [UNK] [UNK] [UNK]

subword tokenisers split unknown words into known pieces:
  routers        → router + s
  misconfigured  → mis + config + ured
so the vocabulary is finite and nothing is ever fully unknown.
```

This is why modern models report vocabularies of around 30,000 to 100,000 pieces rather than millions of words, and why they cope with typos, product codes and languages with heavy compounding. It is also why token counts on an API bill do not match word counts. You are charged per piece.

> **Day 6 takeaway**

Day 7 A Complete Text Workflow

Grid search inside a pipeline, error analysis, and the text-specific traps

A complete text pipeline on the category task, plus the failure modes that are specific to text and that no amount of cross-validation will catch on its own.

The pipeline

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, train_test_split

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['category'], test_size=0.25,
    stratify=tickets['category'], random_state=42)

pipe = Pipeline([('tfidf', TfidfVectorizer()),
                 ('clf', LogisticRegression(max_iter=2000))])
grid = {'tfidf__ngram_range': [(1, 1), (1, 2)],
        'tfidf__min_df': [1, 3],
        'tfidf__sublinear_tf': [False, True],
        'clf__C': [0.5, 2.0, 8.0]}

search = GridSearchCV(pipe, grid, cv=5, scoring='f1_macro', n_jobs=1)
search.fit(X_tr, y_tr)
print('best macro F1 in cross-validation %.4f' % search.best_score_)
for k, v in sorted(search.best_params_.items()):
    print('  %-24s %s' % (k, v))
print('\nheld-out accuracy %.4f' % search.score(X_te, y_te))
```

```
best macro F1 in cross-validation 0.8732
  clf__C                   0.5
  tfidf__min_df            1
  tfidf__ngram_range       (1, 2)
  tfidf__sublinear_tf      True
```

```
held-out accuracy 0.8456
```

Watch Out

Vectorise inside the pipeline, never before the split

`TfidfVectorizer` learns a vocabulary and a document frequency for every term. Fit it on all your text and those statistics have seen the test set, which is week 4's leakage wearing a different hat. Inside a `Pipeline`, every cross-validation fold refits the vectoriser on that fold's training rows only. This is the single most common leak in text projects, and it flatters your score by a few points, just enough to be believable.

Reading the errors

```python
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

X_tr, X_te, y_tr, y_te = train_test_split(
    tickets['text'], tickets['category'], test_size=0.25,
    stratify=tickets['category'], random_state=42)
m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                  LogisticRegression(max_iter=2000, C=2.0)).fit(X_tr, y_tr)
pred = m.predict(X_te)
labels = sorted(y_te.unique())

print(classification_report(y_te, pred, digits=3))
print('confusion matrix (rows = truth):')
print('%-14s %s' % ('', ' '.join('%13s' % l for l in labels)))
for l, row in zip(labels, confusion_matrix(y_te, pred, labels=labels)):
    print('%-14s %s' % (l, ' '.join('%13d' % v for v in row)))
```

```
              precision    recall  f1-score   support

     billing      0.842     0.804     0.822       357
cancellation      0.950     0.732     0.827       157
   technical      0.840     0.922     0.879       536

    accuracy                         0.853      1050
   macro avg      0.877     0.819     0.843      1050
weighted avg      0.857     0.853     0.852      1050

confusion matrix (rows = truth):
```

```
                  billing  cancellation     technical
billing               287             3            67
cancellation           15           115            27
technical              39             3           494
```

The pitfalls that are specific to text

1. **Duplicate documents across the split.** This corpus contains 90 double-submitted tickets on purpose. Split without deduplicating and the same text is in training and test, so you are measuring memory. Deduplicate on the text, not just the id.
2. **Fitting the vectoriser before the split.** Covered above, and worth repeating because it is so easy to do accidentally in a notebook.
3. **Time.** Ticket language drifts, new products, new outages, new slang. A random split tells you how you do on the past; a split by date tells you how you will do next month, and it is always the lower number.
4. **Very short and empty documents.** Nine tickets here are empty strings. They produce an all-zero row, which every linear model will classify identically. Decide deliberately whether to drop them or route them to a human.
5. **Class imbalance.** Cancellation is the smallest and usually the most valuable category. Use macro-averaged F1 rather than accuracy, as the grid above does.
6. **Personal data.** Free text contains names, addresses, account and card numbers. Redact before you store features, and certainly before anything leaves your infrastructure for an API.

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split

raw = pd.read_csv('tickets.csv')            # with the duplicates left in
raw['text'] = raw['text'].fillna('').str.strip()

A, B, ya, yb = train_test_split(raw['text'], raw['resolved'],
                                test_size=0.25, random_state=0)

# Split the test set by whether its text was also in training. Comparing
# those two subsets isolates memorisation; comparing a deduplicated run
# against this one would not, because deduplicating changes the task.
seen = B.isin(set(A))
print('test rows whose text also appears in training: %d of %d\n'
      % (seen.sum(), len(B)))

print('%-22s %8s %8s %8s' % ('', 'seen', 'unseen', 'overall'))
```

```
for name, clf in [('1-nearest neighbour', KNeighborsClassifier(1)),
                  ('logistic regression',
                   LogisticRegression(max_iter=2000))]:
    m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)), clf).fit(A, ya)
    print('%-22s %8.4f %8.4f %8.4f'
          % (name, m.score(B[seen], yb[seen]),
             m.score(B[~seen], yb[~seen]), m.score(B, yb)))
```

```
test rows whose text also appears in training: 38 of 1073

                          seen   unseen  overall
1-nearest neighbour     0.9474   0.6580   0.6682
logistic regression     0.8947   0.9449   0.9432
```

The gap is the leak, and how big it is depends entirely on how much the model can memorise. One-nearest-neighbour finds the identical training ticket and copies its label, so it looks strong on the duplicated rows and collapses on everything else. The regularised linear model, which cannot store individual documents, shows no gap at all.

Watch Out

Do not conclude that duplicates are harmless

They are harmless *to this model*, on a corpus where 38 rows out of a thousand overlap. Scale either of those up, a scraped corpus where the same article appears on forty sites, or any model with the capacity to memorise, which includes every fine-tuned transformer, and the same leak moves your headline number by a lot. Deduplicate anyway: the check is two lines and the failure is invisible.

Saving the whole thing

```
import numpy as np
import pandas as pd

tickets = pd.read_csv('tickets.csv')
tickets['text'] = tickets['text'].fillna('').str.strip()
tickets = tickets.drop_duplicates(subset='ticket_id').reset_index(drop=True)
import joblib
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

model = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                      LogisticRegression(max_iter=2000, C=2.0))
model.fit(tickets['text'], tickets['category'])
joblib.dump(model, 'ticket_router.joblib')

loaded = joblib.load('ticket_router.joblib')
new = ['the invoice is wrong again and nobody has called back',
       'my router is no longer dropping out, thanks',
       'i want to leave, what is the termination fee']
for text, label in zip(new, loaded.predict(new)):
    proba = loaded.predict_proba([text]).max()
```

```
    print('%-13s p=%.2f  %s' % (label, proba, text[:52]))
```

```
billing       p=0.72  the invoice is wrong again and nobody has called bac
technical     p=0.47  my router is no longer dropping out, thanks
cancellation  p=0.80  i want to leave, what is the termination fee
```

One object holds the tokeniser, the vocabulary, the IDF weights and the coefficients, and it takes raw strings as input. That is the property that matters in production: the serving code should never have to reimplement your preprocessing, because the day it drifts from the training version is the day your model silently degrades. Week 15 builds the service around exactly this file.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. A bag of words discards:

   a) Word identity
   b) The order the words appeared in
   c) Rare words
   d) Punctuation only

2. TF-IDF with unigrams scored *below* a constant prediction on the resolution task because:

   a) The labels were random
   b) The corpus was too small
   c) The label lives in an interaction between tokens, and a linear model can only add one weight per token
   d) TF-IDF is unsuitable for text

3. Gradient boosting on those same unigram counts scored far higher. This shows that:

   a) TF-IDF was misconfigured
   b) The split was lucky
   c) Boosting is always better
   d) The information was in the features all along; it was the linear model that could not reach it

4. Averaging a document's word embeddings:

   a) Discards order, just as a bag of words does
   b) Is the same as TF-IDF
   c) Requires an LSTM
   d) Preserves word order

5. Plain recurrent networks struggle with long sequences because the gradient is:

   a) Too large always

b) A product of one factor per step, so it vanishes or explodes

c) Computed only at the end

d) Independent of length

6. Forgetting `mask_zero=True` on an embedding means:

   a) Sequences are truncated

   b) Training is slower

   c) Padding is treated as real input, and a document with no tokens can produce a NaN that silently destroys the run

   d) The vocabulary is wrong

7. Self-attention on its own is order-blind, so transformers add:

   a) A convolution

   b) Larger batches

   c) A recurrent layer

   d) Positional encodings to the embeddings

8. Fitting `TfidfVectorizer` on all your text before splitting:

   a) Leaks, because the vocabulary and document frequencies are learned from the test rows too

   b) Is faster and safe

   c) Only matters for large corpora

   d) Is good practice

Answers:1b, 2c, 3d, 4a, 5b, 6c, 7d, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

Chapter 14

Hyperparameter Optimisation and Interpretability

Week 14

OBJECTIVES

Spend a tuning budget properly, then explain what the tuned model is doing and where the explanation stops.

Two halves of one job. First, how to spend a fixed budget searching for settings, which parameters are worth searching at all, why a grid wastes most of its effort, and how to stop bad candidates early. Then, how to explain the result: the shape of each effect, contributions to a single prediction that add up exactly, what a customer would have to change, and the line between describing a model and making a claim about the world.

Day 1 What to Tune, and How to Search

Which hyperparameters matter, and why random search beats a grid

Every model so far has been fitted with whatever settings scikit-learn ships. Sometimes that is close to optimal and sometimes it is not, and the difference between a productive week of tuning and a wasted one is almost entirely about *which* settings you touch.

Not all hyperparameters matter

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
```

```
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import cross_val_score

def score(**kw):
    m = HistGradientBoostingClassifier(random_state=42, **kw)
    return cross_val_score(m, X_tr, y_tr, cv=5, scoring='roc_auc').mean()

base = score()
print('defaults %.4f\n' % base)
print('%-22s %-18s %9s %9s' % ('parameter', 'values', 'best', 'spread'))
sweeps = {
    'learning_rate': [0.01, 0.05, 0.1, 0.3],
    'max_leaf_nodes': [7, 15, 31, 63],
    'min_samples_leaf': [5, 20, 50, 100],
    'l2_regularization': [0.0, 0.1, 1.0, 10.0],
    'max_features': [0.4, 0.7, 1.0],
}
for name, values in sweeps.items():
    got = [score(**{name: v}) for v in values]
    print('%-22s %-18s %9.4f %9.4f'
          % (name, str(values)[:18], max(got), max(got) - min(got)))
```

```
defaults 0.7945

parameter              values                 best    spread
learning_rate          [0.01, 0.05, 0.1,    0.8012    0.0207
max_leaf_nodes         [7, 15, 31, 63]      0.8120    0.0238
min_samples_leaf       [5, 20, 50, 100]     0.8069    0.0144
l2_regularization      [0.0, 0.1, 1.0, 10   0.8032    0.0117
max_features           [0.4, 0.7, 1.0]      0.7992    0.0063
```

Read the spread column. One or two parameters move the score by something worth having and the rest barely register, and that pattern, not the specific winner, is what generalises. It is why the next two pages are about search strategy rather than about parameter values.

Model	Tune first	Tune if you have budget	Leave alone
Gradient boosting	`learning_rate`, `max_leaf_nodes`, `n_estimators`	`min_samples_leaf`, `l2_regularization`, subsampling	Almost everything else
Random forest	`max_features`, `min_samples_leaf`	`max_depth`	`n_estimators`: set it as high as you can afford
Linear / logistic	`C` or `alpha`	penalty type, `class_weight`	The solver, usually
SVM (RBF)	`C`, `gamma`	kernel choice	-
Neural network	learning rate	width, depth, dropout, batch size	The optimiser, use Adam

`n_estimators` is not a hyperparameter in a forest

More trees in a random forest never makes it worse, it only makes it slower. The variance of the average falls and then flattens. So there is nothing to search: pick the largest number you can afford and move on. In *boosting* it is entirely different, because each tree corrects the last and too many will overfit. Same argument name, opposite behaviour.

What a grid actually costs

```
params = {'learning_rate': [0.01, 0.05, 0.1, 0.3],
          'max_leaf_nodes': [7, 15, 31, 63],
          'min_samples_leaf': [5, 20, 50],
          'l2_regularization': [0.0, 0.1, 1.0]}

total = 1
for name, values in params.items():
    total *= len(values)
    print('%-22s %d values, running total %d' % (name, len(values), total))

for folds in [5, 10]:
    fits = total * folds
    print('\n%d-fold: %d fits' % (folds, fits))
    for secs in [0.2, 2.0]:
        print('   at %.1fs per fit: %.1f minutes' % (secs, fits * secs / 60))
```

```
learning_rate          4 values, running total 4
max_leaf_nodes         4 values, running total 16
min_samples_leaf       3 values, running total 48
l2_regularization      3 values, running total 144

5-fold: 720 fits
   at 0.2s per fit: 2.4 minutes
   at 2.0s per fit: 24.0 minutes

10-fold: 1440 fits
   at 0.2s per fit: 4.8 minutes
   at 2.0s per fit: 48.0 minutes
```

Adding one more value to one more parameter multiplies the cost. A grid is the only strategy whose price grows exponentially in the number of things you are curious about, which is a strange

property for the default choice to have.

Random search, and why it wins

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import time
from scipy.stats import loguniform, randint
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

grid = {'learning_rate': [0.01, 0.05, 0.1, 0.3],
        'max_leaf_nodes': [7, 15, 31, 63],
        'min_samples_leaf': [5, 20, 50, 100]}
dists = {'learning_rate': loguniform(0.005, 0.4),
         'max_leaf_nodes': randint(4, 80),
         'min_samples_leaf': randint(3, 120)}

est = HistGradientBoostingClassifier(random_state=42)
for label, search in [
        ('grid, 64 points', GridSearchCV(est, grid, cv=5,
                                         scoring='roc_auc')),
        ('random, 25 draws', RandomizedSearchCV(est, dists, n_iter=25, cv=5,
                                                scoring='roc_auc',
                                                random_state=0))]:
    t = time.time()
    search.fit(X_tr, y_tr)
```

```
    print('%-18s best %.4f in %5.1fs (%d fits)'
          % (label, search.best_score_, time.time() - t,
             len(search.cv_results_['params']) * 5))
```

```
grid, 64 points    best 0.8155 in  71.8s (320 fits)
random, 25 draws   best 0.8160 in  28.4s (125 fits)
```

```
import numpy as np

# Two parameters, only one of which matters. A 5 x 5 grid tries five
# distinct values of each. Twenty-five random draws try twenty-five.
rng = np.random.default_rng(0)
print('%-16s %22s %22s' % ('', 'distinct values tried', 'of the one that matters'))
for n in [9, 16, 25, 36]:
    side = int(round(n ** 0.5))
    print('%-16s %22d %22d'
          % ('grid %dx%d' % (side, side), n, side))
    print('%-16s %22d %22d' % ('random %d' % n, n, n))
```

```
                 distinct values tried of the one that matters
grid 3x3                             9                      3
random 9                             9                      9
grid 4x4                            16                      4
random 16                           16                     16
grid 5x5                            25                      5
random 25                           25                     25
grid 6x6                            36                      6
random 36                           36                     36
```

The argument in one sentence

A grid spends its budget trying the same few values of the parameter that matters, over and over, paired with values of parameters that do not. Random search spends the same budget on distinct values of everything. Since you rarely know in advance which parameters matter, random search is the better default, and it lets you stop at any point rather than at a multiple of the grid size.

Day 1 takeaway

Day 2 Spending the Budget Well

Successive halving, early stopping, and Bayesian optimisation from scratch

Random search still gives every candidate the full budget, including the hopeless ones. Two better ideas: stop bad candidates early, and let the model itself decide when it has had enough.

Successive halving

```
import numpy as np
import pandas as pd
```

```

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import time
from scipy.stats import loguniform, randint
from sklearn.experimental import enable_halving_search_cv   # still required
from sklearn.model_selection import HalvingRandomSearchCV
from sklearn.ensemble import HistGradientBoostingClassifier

dists = {'learning_rate': loguniform(0.005, 0.4),
         'max_leaf_nodes': randint(4, 80),
         'min_samples_leaf': randint(3, 120)}

search = HalvingRandomSearchCV(
    HistGradientBoostingClassifier(random_state=42), dists,
    n_candidates=60, factor=3, resource='n_samples',
    # Without this the first round scores on 20 rows, where a 5-fold
    # split can hand a fold a single class and roc_auc returns nan.
    min_resources=250, cv=5,
    scoring='roc_auc', random_state=0)
t = time.time()
search.fit(X_tr, y_tr)

res = pd.DataFrame(search.cv_results_)
print('%-8s %12s %12s %12s' % ('round', 'candidates', 'rows each', 'best'))
for i, g in res.groupby('iter'):
    print('%-8d %12d %12d %12.4f'
          % (i, len(g), g['n_resources'].iloc[0], g['mean_test_score'].max()))
print('\n60 candidates explored in %.1fs, best %.4f'
      % (time.time() - t, search.best_score_))
```

```
round      candidates    rows each         best
0                  60          250       0.7730
1                  20          750       0.8011
2                   7         2250       0.8160

60 candidates explored in 43.5s, best 0.8160
```

Sixty candidates screened where day 1's random search managed twenty-five, and the score climbs each round as the survivors get more data, 250 rows, then 750, then all 2,250. It arrives at 0.8160, the same figure random search reached on day 1.

Watch Out

The same answer, for more wall-clock. Measure before you believe

Halving took longer here than the plain random search it matched. That is not a criticism of the algorithm, it is a statement about the dataset: fitting this model on 2,250 rows takes a fraction of a second, so there is nothing to save by fitting it on 250 instead, and the extra rounds of bookkeeping are pure overhead.

Successive halving is built for the opposite situation, hundreds of thousands of rows, or a model that takes minutes per fit, where eliminating forty hopeless candidates cheaply is worth a great deal. Note also `min_resources`: left at its default this search began on 20 rows, where a five-fold split can hand a fold a single class and the AUC comes back as `nan`. If your first round is scoring on almost no data, its ranking is noise and the candidates it discards were discarded at random.

Let the model stop itself

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
```

```
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import time
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import roc_auc_score

print('%-28s %8s %10s %8s' % ('', 'trees', 'test auc', 'seconds'))
for label, kw in [('max_iter=1000, no stopping',
                   dict(max_iter=1000, early_stopping=False)),
                  ('max_iter=1000, early stopping',
                   dict(max_iter=1000, early_stopping=True,
                        n_iter_no_change=20, validation_fraction=0.15))]:
    t = time.time()
    m = HistGradientBoostingClassifier(random_state=42, **kw).fit(X_tr, y_tr)
    print('%-28s %8d %10.4f %8.1f'
          % (label, m.n_iter_,
             roc_auc_score(y_te, m.predict_proba(X_te)[:, 1]),
             time.time() - t))
```

```
                            trees   test auc  seconds
max_iter=1000, no stopping       1000     0.7287      3.9
max_iter=1000, early stopping       47     0.7866      0.2
```

`n_estimators` is the one hyperparameter you should almost never search for a boosted model, because the model can find it itself for the price of a single fit. Set it high, turn on early stopping, and spend your search budget on the parameters that cannot be discovered that way.

Bayesian optimisation, from scratch

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
```

```
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import numpy as np
from scipy.stats import norm
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, ConstantKernel
from sklearn.model_selection import cross_val_score

def objective(log_lr):
    m = HistGradientBoostingClassifier(learning_rate=float(10 ** log_lr),
                                       random_state=42)
    return cross_val_score(m, X_tr, y_tr, cv=5, scoring='roc_auc').mean()

rng = np.random.default_rng(0)
seen_x = list(rng.uniform(-2.3, -0.4, size=5))       # 0.005 to 0.4
seen_y = [objective(x) for x in seen_x]
print('%d random starting points, best %.4f\n' % (len(seen_x), max(seen_y)))

XI = 0.002      # how much improvement counts as worth having
grid = np.linspace(-2.3, -0.4, 60).reshape(-1, 1)
for step in range(6):
    gp = GaussianProcessRegressor(ConstantKernel(1.0) * Matern(nu=2.5),
                                  alpha=1e-6, normalize_y=True,
                                  random_state=0)
    gp.fit(np.array(seen_x).reshape(-1, 1), seen_y)
    mu, sd = gp.predict(grid, return_std=True)
    best = max(seen_y) + XI            # XI keeps it exploring
    z = (mu - best) / np.maximum(sd, 1e-9)
    ei = (mu - best) * norm.cdf(z) + sd * norm.pdf(z)   # expected improvement
    nxt = float(grid[int(ei.argmax()), 0])
    seen_x.append(nxt)
    seen_y.append(objective(nxt))
    print('step %d: tried lr=%.4f → %.4f   (best so far %.4f)'
          % (step + 1, 10 ** nxt, seen_y[-1], max(seen_y)))

i = int(np.argmax(seen_y))
print('\nbest learning rate %.4f scoring %.4f after %d evaluations'
      % (10 ** seen_x[i], seen_y[i], len(seen_y)))
```

```
5 random starting points, best 0.8045

step 1: tried lr=0.0297 → 0.8054   (best so far 0.8054)
step 2: tried lr=0.3981 → 0.7788   (best so far 0.8054)
step 3: tried lr=0.0430 → 0.8031   (best so far 0.8054)
step 4: tried lr=0.0221 → 0.8053   (best so far 0.8054)
step 5: tried lr=0.0105 → 0.8007   (best so far 0.8054)
step 6: tried lr=0.0579 → 0.7969   (best so far 0.8054)

best learning rate 0.0297 scoring 0.8054 after 11 evaluations
```

When this is worth the complexity

Bayesian optimisation earns its keep when a single evaluation is expensive, minutes or hours, because then it is worth thinking hard about where to spend the next one. When a fit takes a second, the thinking costs more than the fitting and random search with more draws wins on wall-clock. Libraries such as Optuna implement this properly, with pruning and parallelism; the version above is here so the idea is not a black box.

> **Day 2 takeaway**

Day 3 Tuning the Whole Pipeline

*Preprocessing as a hyperparameter, caching, and the bias in best_score_*

The choices made before the model sees the data, which imputer, which encoder, whether to scale, are hyperparameters too. They are usually decided once, by hand, and never revisited, which is odd given how much they can matter.

Tuning the preprocessing

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

pipe = Pipeline([('prep', prep),
```

```
                ('clf', LogisticRegression(max_iter=2000))])

space = {
    'prep__num__i__strategy': ['median', 'mean'],
    'prep__num__s': [StandardScaler(), 'passthrough'],
    'prep__cat__o__min_frequency': [1, 20, 60],
    'clf__C': loguniform(0.01, 100),
}
search = RandomizedSearchCV(pipe, space, n_iter=20, cv=5,
                            scoring='roc_auc', random_state=0)
search.fit(X_tr, y_tr)
print('best cross-validated AUC %.4f' % search.best_score_)
for k, v in sorted(search.best_params_.items()):
    print('  %-28s %s' % (k, v))
```

```
best cross-validated AUC 0.8219
  clf__C                       0.19795388574374428
  prep__cat__o__min_frequency  20
  prep__num__i__strategy       median
  prep__num__s                 StandardScaler()
```

The double underscore is a path

`prep__num__i__strategy` reads as: the step named `prep`, inside it the transformer named `num`, inside that the step named `i`, and its `strategy` argument. Any nesting depth works, and passing `'passthrough'` in place of a step is how you make the existence of that step itself a thing to be tuned.

Caching the repeated work, and when not to

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])

```

```
X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
import shutil
import tempfile
import time
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

space = {'clf__C': [0.01, 0.1, 1.0, 10.0, 100.0]}
cache = tempfile.mkdtemp()

for label, memory in [('no cache', None), ('memory=cache', cache)]:
    pipe = Pipeline([('prep', prep),
                     ('clf', LogisticRegression(max_iter=2000))],
                    memory=memory)
    t = time.time()
    GridSearchCV(pipe, space, cv=5, scoring='roc_auc').fit(X_tr, y_tr)
    print('%-14s %.2fs' % (label, time.time() - t))

shutil.rmtree(cache, ignore_errors=True)
print('\nthe preprocessing is identical for every value of C, so the')
print('cache turns 25 fits of it into 5 -- plus 25 trips to disk.')
```

```
no cache       0.44s
memory=cache   0.88s

the preprocessing is identical for every value of C, so the
cache turns 25 fits of it into 5 -- plus 25 trips to disk.
```

Watch Out

Measure it; here the cache made things slower

Caching trades recomputation for serialisation and disk I/O. This `ColumnTransformer` is an imputer, a scaler and a one-hot encoder over 2,250 rows. It costs a few milliseconds, and writing the result to disk and reading it back costs more than that. So the cached version loses.

It wins, and wins enormously, when the preprocessing is genuinely expensive: fitting a `TfidfVectorizer` over a large corpus, a `KNNImputer`, a target encoder, anything reading from disk. The rule is the same one as for successive halving: the optimisation is only an optimisation if you timed it.

How much of the gain is real?

A search reports the score of the best candidate it found. That figure is the maximum of many noisy numbers, and a maximum is biased upwards: some of what the winner won by is genuine and some of it is the fold split happening to suit it.

```
import numpy as np
import pandas as pd

```

```
df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import numpy as np
from scipy.stats import loguniform, randint
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import roc_auc_score

dists = {'learning_rate': loguniform(0.005, 0.4),
         'max_leaf_nodes': randint(4, 80),
         'min_samples_leaf': randint(3, 120)}

print('%-12s %14s %12s %10s' % ('candidates', 'best cv score', 'test score',
                                'gap'))
for n in [5, 20, 60]:
    s = RandomizedSearchCV(HistGradientBoostingClassifier(random_state=42),
                           dists, n_iter=n, cv=5, scoring='roc_auc',
                           random_state=0).fit(X_tr, y_tr)
    test = roc_auc_score(y_te, s.predict_proba(X_te)[:, 1])
    print('%-12d %14.4f %12.4f %10.4f'
          % (n, s.best_score_, test, s.best_score_ - test))
```

```
candidates    best cv score   test score        gap
5                    0.8118       0.7976     0.0142
20                   0.8160       0.8038     0.0121
60                   0.8160       0.8038     0.0121
```

The gap is positive at every budget: the cross-validated figure is optimistic by more than a point of AUC each time. Note also that trying sixty candidates instead of twenty bought nothing at all here, the same winner, the same score. A search that has stopped improving is telling you the ceiling is set by the features, not the settings, and that is a signal to go back to week 8 rather

than to widen the search.

Watch Out

Never report `best_score_` as your result

It is the winner of a competition you ran on your own validation folds, so it carries the same selection bias as any other "best of many" statistic. Report the held-out score, or the nested cross-validation score from week 7 day 1 if you have no held-out set to spare. `best_score_` is for comparing candidates against each other, and for nothing else.

Day 3 takeaway

Day 4 Partial Dependence and ICE

How the prediction responds to a feature, and when the average lies

A tuned model that nobody understands is difficult to defend and impossible to debug. Week 6 established which features a model relies on. This is the next question: *how* does the prediction change as a feature changes?

Partial dependence, by hand first

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
```

```
X_te = X_te.fillna(medians)
import numpy as np
from sklearn.ensemble import HistGradientBoostingClassifier

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)

grid = [1, 6, 12, 24, 36, 48, 60, 72]
print('%14s %14s' % ('tenure_months', 'mean P(churn)'))
for value in grid:
    copy = X_tr.copy()
    copy['tenure_months'] = value           # everybody, at this tenure
    print('%14d %14.4f' % (value, m.predict_proba(copy)[:, 1].mean()))
```

```
 tenure_months  mean P(churn)
             1         0.3687
             6         0.2390
            12         0.3096
            24         0.1227
            36         0.1526
            48         0.0202
            60         0.0079
            72         0.0082
```

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.inspection import partial_dependence

```

```
m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)

for method in ['decision_function', 'predict_proba']:
    res = partial_dependence(m, X_tr, ['tenure_months'],
                             grid_resolution=5, kind='average',
                             # 'recursion' is the fast tree-specific path
                             # and only speaks log-odds; asking for
                             # probabilities means doing it the slow way.
                             method='brute', response_method=method)
    print('response_method=%s' % method)
    print('  ' + '  '.join('%8.4f' % a for a in res['average'][0]))
print('\ngrid: ' + '  '.join('%8.1f' % v
                             for v in res['grid_values'][0]))
```

```
response_method=decision_function
   -1.2841   -1.9846   -3.0956   -5.3464   -5.9621
response_method=predict_proba
    0.3101    0.2245    0.1138    0.0141    0.0079

grid:      4.0      19.2      34.5      49.8      65.0
```

Watch Out

Check the scale before you read the numbers

For a classifier, `response_method='auto'` uses `decision_function` when the estimator has one, so the default output is in *log-odds*, not probability, and will not match the hand-rolled version above. Log-odds is the better scale for seeing shape, because it is the scale the model is additive on. Probability is the better scale for showing anybody else. Pick deliberately, and label the axis.

The average can hide the story

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
```

```
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import numpy as np
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.inspection import partial_dependence

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
res = partial_dependence(m, X_tr, ['monthly_charges'],
                         grid_resolution=6, kind='both')
grid = res['grid_values'][0]
ice = res['individual'][0]

slope = ice[:, -1] - ice[:, 0]
print('%d customers rise with price, %d fall'
      % (int((slope > 0).sum()), int((slope < 0).sum())))
print('\n%14s %10s %10s %10s' % ('charges', 'average', 'p10', 'p90'))
for j, g in enumerate(grid):
    col = ice[:, j]
    print('%14.1f %10.4f %10.4f %10.4f'
          % (g, col.mean(), np.percentile(col, 10),
             np.percentile(col, 90)))
```

```
2123 customers rise with price, 127 fall

       charges    average        p10        p90
          15.0     0.1230     0.0017     0.3105
          30.2     0.1276     0.0022     0.3310
          45.4     0.1177     0.0007     0.3135
          60.6     0.2636     0.0014     0.6268
          75.8     0.2301     0.0034     0.5721
          91.0     0.3635     0.0063     0.8330
```

Two features at once

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
```

```
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.inspection import partial_dependence

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
# Two names in one list asks for the joint surface, not two curves.
res = partial_dependence(m, X_tr, ['tenure_months', 'is_monthly'],
                         grid_resolution=6, kind='average')
tenure, monthly = res['grid_values']
surface = res['average'][0]

print('rows = tenure, columns = contract type')
print('%10s %12s %12s %12s' % ('tenure', 'longer term', 'month-to-month',
                               'difference'))
for i, t in enumerate(tenure):
    a, b = surface[i, 0], surface[i, 1]
    print('%10.1f %12.4f %12.4f %12.4f' % (t, a, b, b - a))
```

```
rows = tenure, columns = contract type
    tenure  longer term month-to-month   difference
       4.0      -0.9477       0.6599       1.6076
      16.2      -1.7094       0.2208       1.9301
      28.4      -3.9701      -0.0364       3.9337
      40.6      -3.9225      -1.7032       2.2193
      52.8      -4.6525      -2.6154       2.0370
      65.0      -5.9758      -4.0526       1.9232
```

If the difference column were constant, the two features would act independently and the model would be additive in them. Where it changes, there is an interaction: the effect of tenure depends on the contract. That is precisely the structure a linear model cannot represent without being told, and the reason week 8 spent a day on interaction features.

Watch Out

Partial dependence averages over combinations that do not exist

Setting `tenure_months=1` for every customer includes customers whose `total_charges` is £3,000, which is a person who has spent three thousand pounds in their first month.

The model is being asked about a region of the input space with no data in it, and its answer there is extrapolation dressed as insight. The stronger the correlation between your features, the less trustworthy the curve, check the correlation before you present one.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
print('correlation with tenure_months:')
print(X_tr.corr()['tenure_months'].drop('tenure_months').round(3)
      .sort_values(key=abs, ascending=False).to_string())
print('\ntotal_charges is nearly determined by tenure, so a partial')
print('dependence plot for one of them holds the other at impossible values.')
```

```
correlation with tenure_months:
total_charges      0.829
is_monthly        -0.463
support_calls      0.033
is_fibre           0.014
monthly_charges    0.007

total_charges is nearly determined by tenure, so a partial
dependence plot for one of them holds the other at impossible values.
```

Day 4 takeaway

Day 5 SHAP

Per-prediction contributions that add up, and what they cost

Partial dependence describes the model. It does not tell a particular customer why *they* were flagged, and that is usually the question that gets asked.

One prediction, explained

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import shap
from sklearn.ensemble import HistGradientBoostingClassifier

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
explainer = shap.TreeExplainer(m)
sv = explainer(X_te.iloc[:50])

i = int(m.predict_proba(X_te.iloc[:50])[:, 1].argmax())   # most at risk
row = X_te.iloc[i]
print('customer predicted at %.3f probability of churn'
      % m.predict_proba(X_te.iloc[[i]])[0, 1])
print('\n%-18s %10s %12s' % ('feature', 'value', 'contribution'))
order = np.argsort(-np.abs(sv.values[i]))
for j in order:
    print('%-18s %10.2f %+12.4f'
          % (X_te.columns[j], row.iloc[j], sv.values[i][j]))
print('%-18s %10s %+12.4f' % ('base value', '', sv.base_values[i]))
```

```
print('%-18s %10s %+12.4f'
      % ('sum', '', sv.values[i].sum() + sv.base_values[i]))
print('model log-odds output %+.4f' % m.decision_function(X_te.iloc[[i]])[0])
```

```
customer predicted at 0.814 probability of churn

feature                  value contribution
is_monthly                1.00      +2.0362
support_calls             4.00      +1.4048
monthly_charges          88.05      +1.0216
total_charges          2590.66      -0.5712
tenure_months            28.00      -0.1467
is_fibre                  1.00      +0.0414
base value                          -2.3128
sum                                 +1.4733
model log-odds output +1.4733
```

Additivity is the property that makes this usable

The contributions plus the base value equal the model's output exactly, not approximately. That is what lets you put the numbers in front of somebody and say "these five things, adding to this" without hand-waving. Note the scale: for a classifier, contributions are in log-odds, not probability, so they add up on a scale most audiences do not think in. Convert carefully, or explain the scale.

Global importance, assembled from local explanations

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
```

```
X_te = X_te.fillna(medians)
import shap
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.inspection import permutation_importance

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
sv = shap.TreeExplainer(m)(X_te)
shap_rank = np.abs(sv.values).mean(axis=0)

perm = permutation_importance(m, X_te, y_te, n_repeats=10,
                              random_state=0, scoring='roc_auc')

print('%-18s %14s %16s' % ('feature', 'mean |shap|', 'permutation'))
for j in np.argsort(-shap_rank):
    print('%-18s %14.4f %16.4f'
          % (X_te.columns[j], shap_rank[j], perm.importances_mean[j]))
```

```
feature              mean |shap|      permutation
is_monthly                1.3276           0.0984
tenure_months             0.9738           0.0804
monthly_charges           0.5814           0.0245
total_charges             0.5090          -0.0027
support_calls             0.4216           0.0191
is_fibre                  0.0885          -0.0015
```

The two rankings usually agree, and when they disagree it is informative rather than alarming: permutation importance measures how much the *score* falls when a feature is scrambled, so it is about predictive usefulness. Mean absolute SHAP measures how much the *prediction* moves, so a feature can swing predictions strongly in both directions and still not improve accuracy.

What it costs

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

```

```
# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import time
import shap
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.linear_model import LogisticRegression

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
lr = LogisticRegression(max_iter=2000).fit(X_tr, y_tr)

t = time.time()
shap.TreeExplainer(m)(X_te)
tree = time.time() - t

t = time.time()
shap.KernelExplainer(lr.predict_proba,
                     shap.sample(X_tr, 50, random_state=0)).shap_values(
    X_te.iloc[:25], nsamples=100, silent=True)
kernel = time.time() - t

print('%-16s %6s %8s %14s' % ('', 'rows', 'seconds', 'ms per row'))
print('%-16s %6d %8.2f %14.2f'
      % ('TreeExplainer', len(X_te), tree, 1000 * tree / len(X_te)))
print('%-16s %6d %8.2f %14.2f'
      % ('KernelExplainer', 25, kernel, 1000 * kernel / 25))
print('\ncompare the last column, not the middle one.')
```

```
                  rows  seconds     ms per row
TreeExplainer      750     0.22           0.29
KernelExplainer     25     0.06           2.59

compare the last column, not the middle one.
```

Per row, the sampling explainer is an order of magnitude more expensive, and that is with only six features, a hundred samples and a fifty-row background set. All three of those have to grow for a real problem, and the cost grows with them. `TreeExplainer` avoids all of it by exploiting the tree structure to compute the exact answer directly, which is why tree models are the ones people actually explain at scale.

Watch Out

Correlated features share credit arbitrarily

If two features carry the same information, any split of the credit between them produces the same prediction, so the axioms do not pin down a unique answer, the implementation picks one. A stakeholder reading "`total_charges` contributed most" may act on it, when `tenure_months` would have served identically. Check your correlations before you present attributions, and consider reporting correlated features as a group.

Day 5 takeaway

Day 6 Counterfactuals, Surrogates and Causality

What would have to change, readable approximations, and the limit

Two more tools, both aimed at the gap between what a model computes and what a person can act on.

Counterfactuals: what would have to change

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import numpy as np
from sklearn.ensemble import HistGradientBoostingClassifier

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
proba = m.predict_proba(X_te)[:, 1]
i = int(proba.argmax())
row = X_te.iloc[[i]]
print('customer starts at P(churn) = %.3f\n' % proba[i])

# Only the levers the business can actually pull.
moves = {'monthly_charges': [-20, -10, -5],
         'support_calls': [-3, -2, -1],
```

```
        'is_monthly': [0]}

print('%-20s %10s %12s %10s' % ('change', 'new value', 'P(churn)', 'drop'))
for feature, deltas in moves.items():
    for delta in deltas:
        candidate = row.copy()
        new = (delta if feature == 'is_monthly'
               else max(0, float(row[feature].iloc[0]) + delta))
        candidate[feature] = new
        p = m.predict_proba(candidate)[0, 1]
        print('%-20s %10.1f %12.3f %10.3f'
              % ('%s %+g' % (feature, delta) if feature != 'is_monthly'
                 else 'move off monthly', new, p, proba[i] - p))
```

```
customer starts at P(churn) = 0.923

change                new value     P(churn)       drop
monthly_charges -20        69.6        0.693      0.230
monthly_charges -10        79.6        0.619      0.304
monthly_charges -5         84.6        0.781      0.142
support_calls -3            0.0        0.545      0.378
support_calls -2            0.0        0.545      0.378
support_calls -1            1.0        0.871      0.052
move off monthly            0.0        0.199      0.724
```

Moving this customer off a month-to-month contract is worth more than any price change on offer, which is an actionable finding rather than a statistical one. Notice too that the price column is not monotone: a £10 reduction helps more than a £20 one. That is not a bug. A boosted tree is a step function, so moving a customer across one threshold can help while moving them further lands them in a different leaf. Search the whole range rather than assuming more of a good thing is better.

Only offer changes somebody can make

A counterfactual saying "if this customer had been with us for forty more months" is useless, because nobody can act on it. Restrict the search to *actionable* features, price, contract type, the number of unresolved support calls, and the output becomes a retention offer rather than a curiosity. It also keeps you away from features that would be unlawful to act on.

Surrogate models

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
```

```
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.tree import DecisionTreeRegressor, export_text

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
target = m.predict_proba(X_tr)[:, 1]        # the black box's own output

print('%8s %16s' % ('depth', 'fidelity (R2)'))
for depth in [2, 3, 4, 6]:
    s = DecisionTreeRegressor(max_depth=depth,
                              random_state=0).fit(X_tr, target)
    print('%8d %16.4f' % (depth, s.score(X_tr, target)))

s = DecisionTreeRegressor(max_depth=3, random_state=0).fit(X_tr, target)
print('\n' + export_text(s, feature_names=list(X_tr.columns),
                         decimals=2)[:1100])
```

```
   depth    fidelity (R2)
       2           0.4484
       3           0.5437
       4           0.6177
       6           0.7213

|--- is_monthly <= 0.50
|   |--- tenure_months <= 17.50
|   |   |--- support_calls <= 2.50
|   |   |   |--- value: [0.18]
|   |   |--- support_calls >  2.50
|   |   |   |--- value: [0.43]
|   |--- tenure_months >  17.50
|   |   |--- monthly_charges <= 75.74
|   |   |   |--- value: [0.02]
|   |   |--- monthly_charges >  75.74
|   |   |   |--- value: [0.07]
|--- is_monthly >  0.50
|   |--- monthly_charges <= 50.98
|   |   |--- total_charges <= 237.06
|   |   |   |--- value: [0.34]
|   |   |--- total_charges >  237.06
|   |   |   |--- value: [0.20]
|   |--- monthly_charges >  50.98
```

```
|   |   |--- tenure_months <= 32.50
|   |   |   |--- value: [0.52]
|   |   |--- tenure_months >  32.50
|   |   |   |--- value: [0.09]
```

Watch Out

Fidelity is the number that matters, and it is usually omitted

A surrogate is an explanation of the model only to the extent that it agrees with the model. Quote the R^2 whenever you show one: at 0.9 it is a fair summary, at 0.5 it is a different model that happens to be readable, and presenting the second as an explanation is worse than presenting nothing. Fidelity is also local. A surrogate can agree well on average and disagree exactly on the cases you care about.

By that standard, the readable depth-3 tree printed above does not qualify: it accounts for about half the variation in the model's output. It is a reasonable description of the broad shape and it should not be presented as what the model does. Depth 6 reaches roughly 0.72 and is already too large to read aloud, which is the trade in a single sentence, readable and faithful are in tension, and a surrogate that is comfortable to present is usually the unfaithful one.

Explanations are not causal

This is the most consequential misunderstanding in the whole subject, so it is worth demonstrating rather than asserting. Add a column that is pure consequence. A retention call that the company makes *because* a customer looks like they are leaving, and watch every interpretability tool point at it.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
```

```
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import numpy as np
import shap
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import roc_auc_score

rng = np.random.default_rng(0)
# The retention team rings customers who are about to leave. The call is
# caused by the churn risk; it does not cause it.
call_tr = (y_tr.to_numpy() * (rng.random(len(y_tr)) < 0.8)).astype(int)
call_te = (y_te.to_numpy() * (rng.random(len(y_te)) < 0.8)).astype(int)

A = X_tr.assign(retention_call=call_tr)
B = X_te.assign(retention_call=call_te)
m = HistGradientBoostingClassifier(random_state=42).fit(A, y_tr)

print('test AUC with the new column %.4f'
      % roc_auc_score(y_te, m.predict_proba(B)[:, 1]))
sv = shap.TreeExplainer(m)(B)
rank = np.abs(sv.values).mean(axis=0)
print('\nmean |shap|:')
for j in np.argsort(-rank):
    print('  %-18s %.4f' % (A.columns[j], rank[j]))
print('\nSHAP is right: the model does rely on it.')
print('Acting on it -- stop making retention calls -- changes nothing.')
```

```
test AUC with the new column 0.9489

mean |shap|:
  retention_call     4.6405
  tenure_months      0.9912
  is_monthly         0.6624
  monthly_charges    0.5483
  total_charges      0.5097
  support_calls      0.3393
  is_fibre           0.0401

SHAP is right: the model does rely on it.
Acting on it -- stop making retention calls -- changes nothing.
```

Watch Out

What every interpretability tool is actually telling you

SHAP, permutation importance and partial dependence all answer the same kind of question: *how does this model use this feature?* None of them answers *what happens in the world if we change this feature?* The second question needs an experiment, or causal assumptions you are prepared to defend in writing. Every time an importance ranking is read as a list of things to go and change, this distinction is the one being skipped.

Day 6 takeaway

Day 7 An Interpretability Report

One function, and the sentences to use when presenting it

One function that produces everything a stakeholder needs, and an honest account of what to say about it.

The report

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split

d = df.copy()
d['is_monthly'] = (d['contract'] == 'Month-To-Month').astype(int)
d['is_fibre'] = (d['internet_service'] == 'Fibre optic').astype(int)
FEATS = ['tenure_months', 'support_calls', 'monthly_charges',
         'total_charges', 'is_monthly', 'is_fibre']

X_tr, X_te, y_tr, y_te = train_test_split(
    d[FEATS], d['churned'], test_size=0.25, stratify=d['churned'],
    random_state=42)

# Impute after the split, with medians learned from the training rows
# only -- week 4's rule still applies. A Pipeline is the right way to do
# this; these pages need a plain frame to inspect column by column, so
# the two lines are written out instead.
medians = X_tr.median()
X_tr = X_tr.fillna(medians)
X_te = X_te.fillna(medians)
import numpy as np
import shap
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.inspection import partial_dependence
from sklearn.metrics import roc_auc_score

def explain(model, X, y, top=4):
    print('=' * 58)
    print('MODEL EXPLANATION REPORT')
    print('=' * 58)
    p = model.predict_proba(X)[:, 1]
    print('rows %d   AUC %.4f   mean predicted risk %.3f'
```

```
          % (len(X), roc_auc_score(y, p), p.mean()))

    sv = shap.TreeExplainer(model)(X)
    rank = np.abs(sv.values).mean(axis=0)
    order = np.argsort(-rank)[:top]

    print('\nWHAT THE MODEL USES')
    for j in order:
        direction = np.corrcoef(X.iloc[:, j], sv.values[:, j])[0, 1]
        arrow = 'higher raises risk' if direction > 0 else 'higher lowers risk'
        print('  %-18s %.4f   %s' % (X.columns[j], rank[j], arrow))

    print('\nSHAPE OF THE TOP EFFECT')
    j = int(order[0])
    res = partial_dependence(model, X, [X.columns[j]], grid_resolution=6,
                             kind='average')
    for v, a in zip(res['grid_values'][0], res['average'][0]):
        print('  %-14s %8.2f → %.4f' % (X.columns[j], v, a))

    print('\nWHO IS MOST AT RISK')
    for i in np.argsort(-p)[:3]:
        top_j = int(np.argmax(np.abs(sv.values[i])))
        print('  row %-5d p=%.3f  driven by %s=%.1f'
              % (i, p[i], X.columns[top_j], X.iloc[i, top_j]))
    print('=' * 58)

m = HistGradientBoostingClassifier(random_state=42).fit(X_tr, y_tr)
explain(m, X_te, y_te)
```

```
==========================================================
MODEL EXPLANATION REPORT
==========================================================
rows 750   AUC 0.7688   mean predicted risk 0.249

WHAT THE MODEL USES
  is_monthly         1.3276   higher raises risk
  tenure_months      0.9738   higher lowers risk
  monthly_charges    0.5814   higher raises risk
  total_charges      0.5090   higher lowers risk

SHAPE OF THE TOP EFFECT
  is_monthly         0.00 → -3.0695
  is_monthly         1.00 → 0.0764

WHO IS MOST AT RISK
  row 60    p=0.923  driven by monthly_charges=89.6
  row 545   p=0.923  driven by tenure_months=2.0
  row 119   p=0.915  driven by monthly_charges=92.5
==========================================================
```

Saying it out loud

Do not say	Say instead	Why
"Tenure is the most important factor"	"Tenure moves this model's predictions most"	Importance is a property of the model, not the world
"Reducing price would cut churn by 8%"	"Customers at lower prices churn less; testing a price change would tell us whether lowering it helps"	The first claims a causal effect the model cannot support
"The model is 87% accurate"	"It ranks well, AUC 0.87, and at our threshold it catches 6 in 10 leavers with 1 in 3 flagged"	Accuracy alone hides the trade-off that actually gets chosen
"The model explains why they left"	"These features drove this score"	It explains a prediction, not a departure

Before you present a model

1. Report a score from data the tuning never saw, not `best_score_`.
2. Show which features the model uses, and by how much, with the caveat that correlated features share credit ambiguously.
3. Show the *shape* of the top two or three effects, not just their rank, and check the correlations before you trust the curve.
4. Explain two or three individual predictions end to end, including one the model got wrong.
5. State what the model would need in order to be causal, and that it is not.
6. Say what the model does when it has never seen this kind of row before.
7. Agree the decision threshold with whoever bears the cost of the errors, using week 7's framing.
8. Write down the date, the data window and the version. Week 15 explains why that last one is not administrative pedantry.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. Random search usually beats grid search at the same budget because:

   a) A grid repeats the same few values of the parameter that matters, while random search tries a distinct value every time

   b) It uses better defaults

   c) Grids cannot use distributions

   d) It is faster per fit

2. For a random forest, `n_estimators` should be:

   a) Tuned carefully

b) Set as high as you can afford, since more trees never makes a forest worse

c) Kept below 50

d) Matched to the feature count

3. Successive halving lost time against plain random search on this dataset because:

a) It used too few candidates

b) It is a poor algorithm

c) There is nothing to save by fitting on fewer rows when a full fit already takes a fraction of a second

d) The seed was wrong

4. `best_score_` from a search should never be reported as your result because:

a) It uses the wrong metric

b) It is not cross-validated

c) It is on the test set

d) It is the maximum of many noisy numbers, and a maximum is biased upwards

5. A partial dependence curve is untrustworthy when the feature is:

a) Strongly correlated with another, so the calculation averages over combinations that never occur

b) Binary

c) Scaled

d) Categorical

6. ICE curves add to partial dependence by showing:

a) A confidence interval

b) One line per row, revealing whether the average represents anybody at all

c) The feature importance

d) The residuals

7. SHAP values for a single prediction:

a) Are in probability units

b) Approximate the output

c) Sum with the base value to the model's output exactly

d) Are always positive

8. An importance ranking tells you:

a) Which feature caused the outcome

b) The correlation with the target

c) What to change in the world to affect the outcome

d) How the model uses each feature, which is not a causal claim

Answers:1a, 2b, 3c, 4d, 5a, 6b, 7c, 8d. Anything you got wrong points at the day to re-read, not at a gap in ability.

Chapter 15

Deployment and MLOps

Week 15

OBJECTIVES

Package a model, serve it over HTTP, validate what arrives, watch it drift and release it safely.

A model in a notebook is a result; a model in production is a commitment. This week covers the distance: what has to be saved besides the estimator, how to serve it and validate what arrives, which failures are silent and therefore dangerous, how to detect drift without labels, and how to replace a live model without anybody having a bad afternoon.

Day 1 Packaging a Model

The artefact, the contract, pinned versions and a smoke test

A model that lives in a notebook is not a product. Everything from here is about the distance between the two, and almost none of it is about machine learning.

What actually has to be saved

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
```

```
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import joblib
import json
import sklearn
import sys
from sklearn.metrics import roc_auc_score

joblib.dump(model, 'churn_model.joblib')

contract = {
    'model_version': '1.0.0',
    'trained_at': '2026-08-08',
    'trained_rows': int(len(X_tr)),
    'features': {'numeric': NUM, 'categorical': CAT},
    'categories': {c: sorted(df[c].dropna().unique().tolist())
                   for c in CAT},
    'ranges': {c: [float(df[c].min()), float(df[c].max())] for c in NUM},
    'threshold': 0.35,
    'test_auc': round(float(roc_auc_score(
        y_te, model.predict_proba(X_te)[:, 1])), 4),
    'python': sys.version.split()[0],
    'sklearn': sklearn.__version__,
}
json.dump(contract, open('churn_contract.json', 'w'), indent=2)
print(json.dumps(contract, indent=2)[:760])
```

```
{
  "model_version": "1.0.0",
  "trained_at": "2026-08-08",
  "trained_rows": 2250,
  "features": {
    "numeric": [
      "tenure_months",
      "support_calls",
      "monthly_charges"
    ],
    "categorical": [
      "contract",
      "internet_service",
      "payment_method",
      "has_dependents"
    ]
  },
  "categories": {
    "contract": [
      "Month-To-Month",
      "One Year",
      "Two Year"
```

```
    ],
    "internet_service": [
      "DSL",
      "Fibre optic",
      "No internet"
    ],
    "payment_method": [
      "Bank transfer",
      "Credit card",
      "Electronic check",
... (13 more lines)
```

The contract is the part people leave out

The `.joblib` file knows how to turn a dataframe into a prediction. It does not know that `contract` has exactly three valid values, that `tenure_months` was never above 72 in training, that the agreed threshold is 0.35 rather than 0.5, or which version of the model produced a given row in your logs. Six months later, every one of those is a question somebody will ask, and the answer needs to be in a file rather than in your memory.

Pinning the environment

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import joblib
import sklearn
```

```

loaded = joblib.load('churn_model.joblib')
print('loaded a %s' % type(loaded).__name__)
print('steps: %s' % [name for name, _ in loaded.steps])
print('fitted with sklearn %s' % sklearn.__version__)

same = (loaded.predict_proba(X_te)[:, 1]
        == model.predict_proba(X_te)[:, 1]).all()
print('reload predicts identically: %s' % bool(same))
```

```
loaded a Pipeline
steps: ['prep', 'clf']
fitted with sklearn 1.8.0
reload predicts identically: True
```

Watch Out

A pickle is not a portable format

`joblib` uses pickle underneath, which stores references to Python classes. Load it under a different scikit-learn version and you get, in ascending order of unpleasantness: a warning, an exception, or a model that loads cleanly and predicts subtly differently. Pin the version, load in the same environment you trained in, and add a smoke test that checks a handful of known inputs still produce known outputs. That test is the only thing that catches the third case.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import joblib
import json

# The smoke test the warning above asks for: freeze a few predictions
# now, and compare against them on every load, forever.
sample = X_te.head(5)
expected = [round(float(p), 6)
            for p in model.predict_proba(sample)[:, 1]]
json.dump({'rows': sample.to_dict('records'), 'expected': expected},
          open('smoke_test.json', 'w'), indent=2)

fixture = json.load(open('smoke_test.json'))
reloaded = joblib.load('churn_model.joblib')
got = [round(float(p), 6) for p in
       reloaded.predict_proba(pd.DataFrame(fixture['rows']))[:, 1]]
print('expected %s' % fixture['expected'])
print('got      %s' % got)
print('smoke test passes: %s' % (got == fixture['expected']))
```

```
expected [0.530642, 0.16512, 0.322593, 0.205547, 0.568866]
got      [0.530642, 0.16512, 0.322593, 0.205547, 0.568866]
smoke test passes: True
```

Day 1 takeaway

Day 2 A Prediction Service

FastAPI, request schemas, and where the latency actually goes

A model behind an HTTP endpoint, which is how most of them are consumed. The service below is written to a file and then genuinely called, so every response on this page came back over a real request cycle.

The service

Save this as `service.py`. It is loaded and called for real further down the page.

```
import json
import joblib
import pandas as pd
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field, ValidationError

MODEL = joblib.load("churn_model.joblib")
CONTRACT = json.load(open("churn_contract.json"))


class Customer(BaseModel):
```

```
    """The request schema. Pydantic rejects anything that does not match,
    before a single row reaches the model."""
    tenure_months: int = Field(ge=0, le=120)
    support_calls: int = Field(ge=0, le=100)
    monthly_charges: float = Field(ge=0, le=500)
    contract: str
    internet_service: str
    payment_method: str
    has_dependents: str


app = FastAPI(title="churn-scorer", version=CONTRACT["model_version"])


@app.get("/health")
def health():
    return {"status": "ok", "model_version": CONTRACT["model_version"],
            "trained_at": CONTRACT["trained_at"]}


@app.post("/predict")
def predict(customer: Customer):
    row = pd.DataFrame([customer.model_dump()])
    for column in ("contract", "internet_service", "payment_method"):
        known = CONTRACT["categories"][column]
        if row.loc[0, column] not in known:
            raise HTTPException(
                status_code=422,
                detail="unknown %s: %r (known: %s)"
                       % (column, row.loc[0, column], known))
    p = float(MODEL.predict_proba(row)[0, 1])
    return {"churn_probability": round(p, 4),
            "decision": "flag" if p >= CONTRACT["threshold"] else "keep",
            "threshold": CONTRACT["threshold"],
            "model_version": CONTRACT["model_version"]}
```

What a request and response look like

```
SERVICE = '''import json
import joblib
import pandas as pd
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field, ValidationError

MODEL = joblib.load("churn_model.joblib")
CONTRACT = json.load(open("churn_contract.json"))


class Customer(BaseModel):
    """The request schema. Pydantic rejects anything that does not match,
    before a single row reaches the model."""
    tenure_months: int = Field(ge=0, le=120)
```

```
    support_calls: int = Field(ge=0, le=100)
    monthly_charges: float = Field(ge=0, le=500)
    contract: str
    internet_service: str
    payment_method: str
    has_dependents: str


app = FastAPI(title="churn-scorer", version=CONTRACT["model_version"])


@app.get("/health")
def health():
    return {"status": "ok", "model_version": CONTRACT["model_version"],
            "trained_at": CONTRACT["trained_at"]}


@app.post("/predict")
def predict(customer: Customer):
    row = pd.DataFrame([customer.model_dump()])
    for column in ("contract", "internet_service", "payment_method"):
        known = CONTRACT["categories"][column]
        if row.loc[0, column] not in known:
            raise HTTPException(
                status_code=422,
                detail="unknown %s: %r (known: %s)"
                       % (column, row.loc[0, column], known))
    p = float(MODEL.predict_proba(row)[0, 1])
    return {"churn_probability": round(p, 4),
            "decision": "flag" if p >= CONTRACT["threshold"] else "keep",
            "threshold": CONTRACT["threshold"],
            "model_version": CONTRACT["model_version"]}
'''
import io, sys, json
io.open('service.py', 'w', encoding='utf-8').write(SERVICE)
sys.path.insert(0, '.')

from fastapi.testclient import TestClient
import service
client = TestClient(service.app)

print('GET /health →', client.get('/health').status_code)
print(json.dumps(client.get('/health').json(), indent=2))

at_risk = {'tenure_months': 2, 'support_calls': 5,
           'monthly_charges': 94.2, 'contract': 'Month-To-Month',
           'internet_service': 'Fibre optic',
           'payment_method': 'Electronic check',
           'has_dependents': 'No'}
settled = dict(at_risk, tenure_months=60, support_calls=0,
               contract='Two Year', monthly_charges=42.0)

for label, payload in [('new, unhappy', at_risk),
                       ('long-standing', settled)]:
```

```
    r = client.post('/predict', json=payload)
    print('\n%s → %d' % (label, r.status_code))
    print(json.dumps(r.json(), indent=2))
```

```
GET /health → 200
{
  "status": "ok",
  "model_version": "1.0.0",
  "trained_at": "2026-08-08"
}

new, unhappy → 200
{
  "churn_probability": 0.8896,
  "decision": "flag",
  "threshold": 0.35,
  "model_version": "1.0.0"
}

long-standing → 200
{
  "churn_probability": 0.004,
  "decision": "keep",
  "threshold": 0.35,
  "model_version": "1.0.0"
}
```

Single or batch?

```
SERVICE = '''import json
import joblib
import pandas as pd
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field, ValidationError

MODEL = joblib.load("churn_model.joblib")
CONTRACT = json.load(open("churn_contract.json"))


class Customer(BaseModel):
    """The request schema. Pydantic rejects anything that does not match,
    before a single row reaches the model."""
    tenure_months: int = Field(ge=0, le=120)
    support_calls: int = Field(ge=0, le=100)
    monthly_charges: float = Field(ge=0, le=500)
    contract: str
    internet_service: str
    payment_method: str
    has_dependents: str


app = FastAPI(title="churn-scorer", version=CONTRACT["model_version"])


```

```
@app.get("/health")
def health():
    return {"status": "ok", "model_version": CONTRACT["model_version"],
            "trained_at": CONTRACT["trained_at"]}


@app.post("/predict")
def predict(customer: Customer):
    row = pd.DataFrame([customer.model_dump()])
    for column in ("contract", "internet_service", "payment_method"):
        known = CONTRACT["categories"][column]
        if row.loc[0, column] not in known:
            raise HTTPException(
                status_code=422,
                detail="unknown %s: %r (known: %s)"
                       % (column, row.loc[0, column], known))
    p = float(MODEL.predict_proba(row)[0, 1])
    return {"churn_probability": round(p, 4),
            "decision": "flag" if p >= CONTRACT["threshold"] else "keep",
            "threshold": CONTRACT["threshold"],
            "model_version": CONTRACT["model_version"]}
'''
import io, sys, json
io.open('service.py', 'w', encoding='utf-8').write(SERVICE)
sys.path.insert(0, '.')

from fastapi.testclient import TestClient
import service
client = TestClient(service.app)
import time

payload = {'tenure_months': 2, 'support_calls': 5,
           'monthly_charges': 94.2, 'contract': 'Month-To-Month',
           'internet_service': 'Fibre optic',
           'payment_method': 'Electronic check',
           'has_dependents': 'No'}

client.post('/predict', json=payload)          # warm up
t = time.time()
N = 100
for _ in range(N):
    client.post('/predict', json=payload)
per_call = (time.time() - t) / N
print('one row at a time: %.1f ms per prediction' % (1000 * per_call))

import pandas as pd
rows = pd.DataFrame([payload] * N)
t = time.time()
service.MODEL.predict_proba(rows)
batched = (time.time() - t) / N
print('%d rows in one call: %.3f ms per prediction' % (N, 1000 * batched))
print('\nratio %.0fx' % (per_call / batched))
```

```
one row at a time: 5.3 ms per prediction
```

```
100 rows in one call: 0.030 ms per prediction

ratio 176x
```

Almost none of that time is the model

The per-request cost is HTTP parsing, validation, building a one-row dataframe and serialising a response. The arithmetic the model does is a rounding error next to it. Two consequences: offer a batch endpoint if your caller has many rows, because it is very nearly free; and do not optimise the model for latency until you have measured where the latency is.

> **Day 2 takeaway**

Day 3 Validation and Failing Loudly

The bad inputs that raise, and the dangerous one that does not

The interesting failures are not the ones that raise. They are the ones where the service returns 200 and a number that means nothing.

The three kinds of bad input

```python
SERVICE = '''import json
import joblib
import pandas as pd
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field, ValidationError

MODEL = joblib.load("churn_model.joblib")
CONTRACT = json.load(open("churn_contract.json"))


class Customer(BaseModel):
    """The request schema. Pydantic rejects anything that does not match,
    before a single row reaches the model."""
    tenure_months: int = Field(ge=0, le=120)
    support_calls: int = Field(ge=0, le=100)
    monthly_charges: float = Field(ge=0, le=500)
    contract: str
    internet_service: str
    payment_method: str
    has_dependents: str


app = FastAPI(title="churn-scorer", version=CONTRACT["model_version"])


@app.get("/health")
def health():
    return {"status": "ok", "model_version": CONTRACT["model_version"],
```

```
            "trained_at": CONTRACT["trained_at"]}


@app.post("/predict")
def predict(customer: Customer):
    row = pd.DataFrame([customer.model_dump()])
    for column in ("contract", "internet_service", "payment_method"):
        known = CONTRACT["categories"][column]
        if row.loc[0, column] not in known:
            raise HTTPException(
                status_code=422,
                detail="unknown %s: %r (known: %s)"
                       % (column, row.loc[0, column], known))
    p = float(MODEL.predict_proba(row)[0, 1])
    return {"churn_probability": round(p, 4),
            "decision": "flag" if p >= CONTRACT["threshold"] else "keep",
            "threshold": CONTRACT["threshold"],
            "model_version": CONTRACT["model_version"]}
'''
import io, sys, json
io.open('service.py', 'w', encoding='utf-8').write(SERVICE)
sys.path.insert(0, '.')

from fastapi.testclient import TestClient
import service
client = TestClient(service.app)

good = {'tenure_months': 12, 'support_calls': 1,
        'monthly_charges': 70.0, 'contract': 'One Year',
        'internet_service': 'DSL',
        'payment_method': 'Bank transfer', 'has_dependents': 'Yes'}

cases = [
    ('valid request', good),
    ('missing a field', {k: v for k, v in good.items()
                         if k != 'monthly_charges'}),
    ('wrong type', dict(good, tenure_months='twelve')),
    ('out of range', dict(good, tenure_months=-4)),
    ('unknown category', dict(good, contract='Pay As You Go')),
]
for label, payload in cases:
    r = client.post('/predict', json=payload)
    body = r.json()
    detail = body.get('detail', body)
    if isinstance(detail, list):
        detail = '%s: %s' % (detail[0].get('loc'), detail[0].get('msg'))
    print('%-18s %d  %s' % (label, r.status_code, str(detail)[:64]))
```

```
valid request      200  {'churn_probability': 0.1575, 'decision': 'keep', 'threshold': 0
missing a field    422  ['body', 'monthly_charges']: Field required
wrong type         422  ['body', 'tenure_months']: Input should be a valid integer, unab
out of range       422  ['body', 'tenure_months']: Input should be greater than or equal
unknown category   422  unknown contract: 'Pay As You Go' (known: ['Month-To-Month', 'On
```

The one that does not raise

A category the model has never seen is the dangerous case, because `OneHotEncoder(handle_unknown='ignore')`, which every week of this course has used, for good reasons, turns it silently into a row of zeros and predicts anyway.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import joblib

model = joblib.load('churn_model.joblib')
base = X_te.iloc[[0]].copy()
print('the row as it stands:      %.4f'
      % model.predict_proba(base)[0, 1])

for value in ['Two Year', 'Pay As You Go', 'PAYG', '']:
    row = base.copy()
    row['contract'] = value
    print('contract=%-16r %.4f  (no error raised)'
          % (value, model.predict_proba(row)[0, 1]))
```

```
the row as it stands:      0.5306
contract='Two Year'       0.0665  (no error raised)
contract='Pay As You Go'  0.2338  (no error raised)
contract='PAYG'           0.2338  (no error raised)
contract=''               0.2338  (no error raised)
```

Watch Out

Silent is worse than wrong

Every one of those returned a plausible number. If a new contract type is launched and nobody tells the model's owner, the service keeps answering, scoring every customer on that plan as though their contract field were blank, and nothing in any log looks unusual. That is why the service checks the value against the contract file and returns a 422. Rejecting a request is a bad day; answering it wrongly for eight months is a different kind of day.

Range checks are not the same as schema checks

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import joblib
import json

model = joblib.load('churn_model.joblib')
contract = json.load(open('churn_contract.json'))
lo, hi = contract['ranges']['tenure_months']
print('training range for tenure_months: %.0f to %.0f' % (lo, hi))

row = X_te.iloc[[0]].copy()
for months in [12, 72, 96, 240]:
    row['tenure_months'] = months
```

```
    p = model.predict_proba(row)[0, 1]
    flag = '' if lo <= months <= hi else '   <-- outside training range'
    print('tenure=%-5d %.4f%s' % (months, p, flag))
```

```
training range for tenure_months: 1 to 72
tenure=12    0.5582
tenure=72    0.0429
tenure=96    0.0117   <-- outside training range
tenure=240   0.0000   <-- outside training range
```

The schema accepts anything up to 120 months because that is a plausible tenure. The *model* has never seen beyond 72, so past that it is extrapolating along a straight line in log-odds and there is nothing to stop it. Validate against the schema to reject nonsense; validate against the training ranges to decide whether to trust the answer.

Situation	Status	Return
Field missing or wrong type	422	Which field, and why
Value outside the schema's range	422	The permitted range
Category never seen in training	422	The known values
Inside the schema, outside the training range	200	The prediction, plus a warning flag
Model file missing or corrupt	503	Fail the health check, do not answer
Anything unexpected	500	A request id; log the traceback

Day 3 takeaway

Day 4 Monitoring and Drift

Logging, PSI, category mix and watching your own output move

The model is live and answering. The question now is how you find out it has stopped being any good, given that nobody will tell you.

Log the prediction, not just the answer

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import hashlib
import joblib
import json

model = joblib.load('churn_model.joblib')
contract = json.load(open('churn_contract.json'))

def log_line(request_id, row, proba):
    return json.dumps({
        'request_id': request_id,
        'model_version': contract['model_version'],
        'threshold': contract['threshold'],
        'probability': round(float(proba), 4),
        'decision': 'flag' if proba >= contract['threshold'] else 'keep',
        # the inputs, so drift is measurable later; hash anything
        # identifying rather than storing it
        'features': {k: (round(v, 2) if isinstance(v, float) else v)
                     for k, v in row.items()},
    })

for i in range(3):
    row = X_te.iloc[i].to_dict()
    p = model.predict_proba(X_te.iloc[[i]])[0, 1]
    print(log_line('req-%03d' % i, row, p))
```

```
{"request_id": "req-000", "model_version": "1.0.0", "threshold": 0.35, "probability": 0.5306,
    ↪ "decision": "flag", "features": {"tenure_months": 14, "support_calls": 3,
    ↪ "monthly_charges": 73.35, "contract": "Month-To-Month", "internet_service": "Fibre
    ↪ optic", "payment_method": "Credit card", "has_dependents": "No"}}
{"request_id": "req-001", "model_version": "1.0.0", "threshold": 0.35, "probability": 0.1651,
    ↪ "decision": "keep", "features": {"tenure_months": 26, "support_calls": 1,
    ↪ "monthly_charges": 15.0, "contract": "Month-To-Month", "internet_service": "No
    ↪ internet", "payment_method": "Electronic check", "has_dependents": "No"}}
{"request_id": "req-002", "model_version": "1.0.0", "threshold": 0.35, "probability": 0.3226,
    ↪ "decision": "keep", "features": {"tenure_months": 14, "support_calls": 1,
    ↪ "monthly_charges": 53.11, "contract": "Month-To-Month", "internet_service": "DSL",
    ↪ "payment_method": "Bank transfer", "has_dependents": "No"}}
```

Log the features or you cannot diagnose anything

A log of probabilities tells you the distribution of your output moved. A log of probabilities *and inputs* tells you which input moved and therefore what happened in the business. The second costs a few hundred bytes per request and is the difference between "something changed in March" and "fibre pricing changed in March".

Data drift

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import numpy as np

def psi(reference, live, buckets=10):
    edges = np.quantile(reference, np.linspace(0, 1, buckets + 1))
    edges[0], edges[-1] = -np.inf, np.inf
    a = np.histogram(reference, edges)[0] / len(reference)
    b = np.histogram(live, edges)[0] / len(live)
    a, b = np.clip(a, 1e-6, None), np.clip(b, 1e-6, None)
    return float(((b - a) * np.log(b / a)).sum())

rng = np.random.default_rng(0)
reference = X_tr['monthly_charges'].dropna().to_numpy()

print('%-28s %8s %s' % ('live traffic', 'PSI', 'verdict'))
for label, live in [
```

```
        ('same distribution', rng.choice(reference, 800)),
        ('prices up 5%', rng.choice(reference, 800) * 1.05),
        ('prices up 20%', rng.choice(reference, 800) * 1.20),
        ('only premium customers',
         rng.choice(reference[reference > np.median(reference)], 800))]:
    value = psi(reference, live)
    verdict = ('stable' if value < 0.1 else
               'investigate' if value < 0.25 else 'material shift')
    print('%-28s %8.4f %s' % (label, value, verdict))
```

```
live traffic                    PSI verdict
same distribution            0.0180 stable
prices up 5%                 0.0688 stable
prices up 20%                0.5386 material shift
only premium customers       6.1024 material shift
```

Categorical drift, and drift in the output itself

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import numpy as np
import joblib

model = joblib.load('churn_model.joblib')
rng = np.random.default_rng(0)
```

```
35
36 train_mix = X_tr['contract'].value_counts(normalize=True)
37 print('training contract mix:')
38 print(train_mix.round(3).to_string())
39
40 # A marketing push sells monthly plans to everybody.
41 shifted = X_te.copy()
42 flip = rng.random(len(shifted)) < 0.5
43 shifted.loc[flip, 'contract'] = 'Month-To-Month'
44
45 live_mix = shifted['contract'].value_counts(normalize=True)
46 print('\nlive contract mix:')
47 print(live_mix.round(3).to_string())
48
49 before = model.predict_proba(X_te)[:, 1]
50 after = model.predict_proba(shifted)[:, 1]
51 print('\nmean predicted risk  %.4f → %.4f' % (before.mean(), after.mean()))
52 print('flagged at 0.35      %.1f%% → %.1f%%'
53       % (100 * (before >= 0.35).mean(), 100 * (after >= 0.35).mean()))
```

```
training contract mix:
contract
Month-To-Month    0.553
One Year          0.243
Two Year          0.204

live contract mix:
contract
Month-To-Month    0.759
One Year          0.132
Two Year          0.109

mean predicted risk  0.2628 → 0.3005
flagged at 0.35      33.5% → 39.9%
```

Prediction drift is the cheapest alarm you can build, because it needs no labels and no feature analysis, just the mean of today's scores against the mean of last month's. It cannot tell you whether the model is still *right*. It can tell you that something changed, on the day it changes, which is usually months before the labels arrive.

Day 4 takeaway

Day 5 Delayed Labels

Why you cannot measure accuracy yet, and what to watch instead

Drift says the inputs changed. It does not say the model got worse, for that you need labels, and labels are the thing you do not have.

The delay is the whole problem

```
events = [
    ('Monday', 'model scores 4,000 customers'),
    ('Monday', 'retention team contacts the 600 flagged'),
    ('+30 days', 'first contracts come up for renewal'),
    ('+60 days', 'roughly half of the outcomes are known'),
    ('+90 days', 'the label for Monday is finally complete'),
]
for when, what in events:
    print('%-10s %s' % (when, what))
print('\nAny accuracy you measure today describes the model as it was')
print('three months ago -- on data that has since drifted.')
```

```
Monday     model scores 4,000 customers
Monday     retention team contacts the 600 flagged
+30 days   first contracts come up for renewal
+60 days   roughly half of the outcomes are known
+90 days   the label for Monday is finally complete

Any accuracy you measure today describes the model as it was
three months ago -- on data that has since drifted.
```

Watch Out

And the intervention destroys the label

Worse than the delay: the retention team acted on the predictions. A customer who was flagged, offered a discount and stayed now counts as a false positive, so a model that worked perfectly and prompted a successful intervention looks, in the data, like a model that was wrong. Measuring a model whose output changes the world requires a holdout that nobody acts on, agreed with the business before you deploy. It is a conversation, not a technique.

What to watch while you wait

Signal	Available	Tells you
Prediction distribution	Immediately	Something changed
Feature PSI	Immediately	Which input changed
Unknown-category rejection rate	Immediately	A new product or a broken upstream field
Missing-value rate per field	Immediately	An upstream pipeline has broken
Latency and error rate	Immediately	The service itself
Proxy outcomes (support calls, logins)	Days	Early behavioural signal
Actual labels	Weeks to months	Whether it was any good
Holdout labels	Weeks to months	Whether it was any good, uncontaminated

An alert that does not cry wolf

```
import numpy as np
import pandas as pd
```

```

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import numpy as np
import joblib

model = joblib.load('churn_model.joblib')
rng = np.random.default_rng(1)

baseline = model.predict_proba(X_tr)[:, 1]
mu, sd = baseline.mean(), baseline.std()
print('baseline mean risk %.4f (sd %.4f)' % (mu, sd))

def day_of_traffic(shift):
    sample = X_te.sample(400, replace=True, random_state=int(shift * 100))
    sample = sample.copy()
    sample['monthly_charges'] = sample['monthly_charges'] * (1 + shift)
    return model.predict_proba(sample)[:, 1]

print('\n%-8s %12s %12s %s' % ('shift', 'mean risk', 'z-score', 'alert'))
for shift in [0.0, 0.05, 0.10, 0.25, 0.50]:
    scores = day_of_traffic(shift)
    z = (scores.mean() - mu) / (sd / np.sqrt(len(scores)))
    print('%-8.0f%% %12.4f %12.1f %s'
          % (100 * shift, scores.mean(), z, 'ALERT' if abs(z) > 4 else ''))
```

```
baseline mean risk 0.2680 (sd 0.2247)

shift       mean risk      z-score alert
0       %       0.2654         -0.2
```

```
5       %       0.2696          0.1
10      %       0.2834          1.4
25      %       0.3080          3.6
50      %       0.3490          7.2 ALERT
```

A threshold on the raw mean would fire on every quiet Tuesday. Comparing against the variability of the baseline, how far today is from normal, in units of how much normal wobbles, is what makes an alert survive contact with a real on-call rota. Set the bar high enough that firing means something, because an alert nobody trusts is worse than no alert at all.

> **Day 5 takeaway**

Day 6 Retraining and Rollout

Champion versus challenger with an interval, shadow mode and canaries

Something has drifted, or three months have passed. Retraining is straightforward; deciding whether the new model is better, and getting it into production without a bad afternoon, is not.

Champion and challenger

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import numpy as np
```

```
import joblib
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import roc_auc_score, brier_score_loss

champion = joblib.load('churn_model.joblib')

challenger = Pipeline([('prep', prep),
                       ('clf', HistGradientBoostingClassifier(
                           random_state=42))]).fit(X_tr, y_tr)

print('%-14s %10s %10s %12s' % ('', 'AUC', 'Brier', 'flagged %'))
for name, m in [('champion', champion), ('challenger', challenger)]:
    p = m.predict_proba(X_te)[:, 1]
    print('%-14s %10.4f %10.4f %11.1f%%'
          % (name, roc_auc_score(y_te, p), brier_score_loss(y_te, p),
             100 * (p >= 0.35).mean()))

# Is the difference bigger than the noise? Bootstrap the test set.
rng = np.random.default_rng(0)
pc = champion.predict_proba(X_te)[:, 1]
pch = challenger.predict_proba(X_te)[:, 1]
yv = y_te.to_numpy()
diffs = []
for _ in range(400):
    idx = rng.integers(0, len(yv), len(yv))
    if yv[idx].sum() in (0, len(idx)):
        continue
    diffs.append(roc_auc_score(yv[idx], pch[idx])
                 - roc_auc_score(yv[idx], pc[idx]))
lo, hi = np.percentile(diffs, [2.5, 97.5])
print('\nAUC difference (challenger - champion)')
print('  point estimate %+.4f' % (roc_auc_score(yv, pch)
                                  - roc_auc_score(yv, pc)))
print('  95%% interval   %+.4f to %+.4f' % (lo, hi))
```

```
                    AUC      Brier    flagged %
champion         0.8174     0.1475        33.5%
challenger       0.7797     0.1690        30.3%

AUC difference (challenger - champion)
  point estimate -0.0376
  95% interval   -0.0582 to -0.0141
```

The challenger loses, and the interval is nowhere near zero, which settles it. This is week 6's result arriving in an operational costume: the data-generating process behind this dataset is close to logistic-linear, so a boosted ensemble has nothing to find that the linear model has not already found, and pays for the search with variance. The correct action here is to keep the champion and write down that the challenger was tried.

"Better" needs an interval, not a point

A challenger that is 0.004 ahead on one test set is not obviously ahead at all. Bootstrap the difference and look at whether the interval clears zero. If it does not, the honest conclusion is that

the two models are indistinguishable on the evidence available, at which point you should prefer the one that is simpler, faster, already deployed, or easier to explain. "No change" is a legitimate and frequently correct outcome of a retraining cycle.

Getting it out there

Strategy	How	Catches	Costs
Shadow	New model scores every request; output logged, not used	Crashes, latency, wild predictions	Compute, and no outcome data
Canary	5% of traffic, then 25%, then all	Real-world failures, at 5% of the blast radius	A slower rollout
A/B split	Randomised, both act on their predictions	Genuine causal comparison	Weeks, and real design work
Blue/green	Both live, switch the router	Nothing extra, but rollback is instant	Two environments
Big bang	Replace it	Nothing	Your evening

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import numpy as np
import joblib
from sklearn.ensemble import HistGradientBoostingClassifier

champion = joblib.load('churn_model.joblib')
```

```
challenger = Pipeline([('prep', prep),
                       ('clf', HistGradientBoostingClassifier(
                           random_state=42))]).fit(X_tr, y_tr)

# Shadow mode: the champion decides, the challenger is recorded.
a = champion.predict_proba(X_te)[:, 1]
b = challenger.predict_proba(X_te)[:, 1]
flag_a, flag_b = a >= 0.35, b >= 0.35

print('champion flags   %d' % flag_a.sum())
print('challenger flags %d' % flag_b.sum())
print('they disagree on %d of %d customers (%.1f%%)'
      % ((flag_a != flag_b).sum(), len(a),
         100 * (flag_a != flag_b).mean()))
print('\nlargest disagreements:')
gap = np.abs(a - b)
for i in np.argsort(-gap)[:4]:
    print('  champion %.3f  challenger %.3f  tenure=%d contract=%s'
          % (a[i], b[i], X_te.iloc[i]['tenure_months'],
             X_te.iloc[i]['contract']))
```

```
champion flags   251
challenger flags 227
they disagree on 112 of 750 customers (14.9%)

largest disagreements:
  champion 0.131  challenger 0.665  tenure=33 contract=Month-To-Month
  champion 0.392  challenger 0.887  tenure=23 contract=Month-To-Month
  champion 0.466  challenger 0.944  tenure=22 contract=Month-To-Month
  champion 0.430  challenger 0.889  tenure=22 contract=One Year
```

Before any traffic moves, this tells you how different the two models actually are in the only terms that matter: how many decisions would change. A challenger that is 0.01 better on AUC but flips a third of your decisions is a much bigger operational event than that number suggests, and the retention team needs to hear about it from you rather than from their workload.

When to retrain

1. **On a schedule**, if the world moves steadily. Monthly is a common default; it is a decision, not a law.
2. **On drift**, when PSI on an important feature crosses your threshold and stays there.
3. **On performance**, when holdout labels show a real fall, the most reliable trigger and the slowest.
4. **On a known event**: a new product, a pricing change, a changed upstream field. Do not wait for the monitoring to notice something you already know.
5. **Never automatically without a gate.** Automatic retraining plus automatic deployment means a corrupted upstream table becomes a corrupted production model with nobody in the loop.

Day 6 takeaway

Day 7 The Complete Service

Everything assembled, and the checklist before it takes traffic

The complete service, with everything the previous six days argued for, and a checklist to work through before anything of yours takes real traffic.

The finished thing

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))])

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)
model.fit(X_tr, y_tr)
import json
import joblib
import numpy as np

class ChurnService:
    """Everything the six days argued for, in one object."""

    def __init__(self, model_path, contract_path):
        self.model = joblib.load(model_path)
        self.contract = json.load(open(contract_path))
        self.seen = 0
```

```
        self.rejected = 0
        self.scores = []

    def _validate(self, row):
        for column, known in self.contract['categories'].items():
            if row.get(column) not in known:
                return 'unknown %s: %r' % (column, row.get(column))
        for column, (lo, hi) in self.contract['ranges'].items():
            value = row.get(column)
            if value is None or not np.isfinite(value):
                return 'missing %s' % column
        return None

    def predict(self, row):
        self.seen += 1
        problem = self._validate(row)
        if problem:
            self.rejected += 1
            return {'status': 'rejected', 'reason': problem}
        p = float(self.model.predict_proba(pd.DataFrame([row]))[0, 1])
        self.scores.append(p)
        outside = [c for c, (lo, hi) in self.contract['ranges'].items()
                   if not lo <= row[c] <= hi]
        return {'status': 'ok',
                'probability': round(p, 4),
                'decision': 'flag' if p >= self.contract['threshold']
                            else 'keep',
                'model_version': self.contract['model_version'],
                'extrapolating': outside or None}

    def stats(self):
        s = np.array(self.scores) if self.scores else np.array([0.0])
        return {'requests': self.seen, 'rejected': self.rejected,
                'mean_score': round(float(s.mean()), 4),
                'flag_rate': round(float(
                    (s >= self.contract['threshold']).mean()), 4)}

svc = ChurnService('churn_model.joblib', 'churn_contract.json')
for row in X_te.head(200).to_dict('records'):
    svc.predict(row)

print(json.dumps(svc.predict(X_te.iloc[0].to_dict()), indent=2))
print(json.dumps(svc.predict(dict(X_te.iloc[0].to_dict(),
                                  contract='Pay As You Go')), indent=2))
print(json.dumps(svc.predict(dict(X_te.iloc[0].to_dict(),
                                  tenure_months=200)), indent=2))
print('\n' + json.dumps(svc.stats(), indent=2))
```

```
{
  "status": "ok",
  "probability": 0.5306,
  "decision": "flag",
  "model_version": "1.0.0",
  "extrapolating": null
```

```
}
{
  "status": "rejected",
  "reason": "unknown contract: 'Pay As You Go'"
}
{
  "status": "ok",
  "probability": 0.0,
  "decision": "keep",
  "model_version": "1.0.0",
  "extrapolating": [
    "tenure_months"
  ]
}

{
  "requests": 203,
  "rejected": 28,
  "mean_score": 0.2558,
  "flag_rate": 0.32
}
```

Look at the rejection count: 28 of 203 real customers from the test set were refused. They are the rows with a missing categorical value, the roughly six percent that `customers.csv` has carried since week 1. The training pipeline imputes those without complaining. The service refuses them.

Watch Out

Your service can be stricter than your model, and you must choose

Neither behaviour is wrong, but the disagreement has to be a decision rather than an accident. Rejecting is right when a missing field means an upstream system is broken and a prediction would be misleading. Imputing is right when the field is genuinely optional and the model was trained to cope. What is never right is discovering the difference in production, when one in twenty of your customers cannot be scored and nobody knows why.

Write the rule down in the contract file next to the categories, and test it. Here, the honest fix is to allow missing categoricals (the pipeline handles them, and the training data contained them) while continuing to reject unknown *values*, which the pipeline silently mishandles.

The checklist

1. The artefact contains the preprocessing, not just the estimator.
2. A contract file records the features, valid categories, training ranges, threshold, training date and versions.
3. A smoke test with frozen inputs and expected outputs runs on every deploy.
4. Dependencies are pinned and the service runs in the environment it was trained in.
5. The model loads once at start-up, and a health endpoint reports its version.
6. Unknown categories are rejected rather than silently zero-encoded.
7. Inputs outside the training ranges are answered but flagged.

8. Every prediction is logged with its inputs, the threshold and the model version.
9. Feature PSI, category mix, missing-value rates and the prediction distribution are monitored, with thresholds set from observed variation.
10. An unactioned holdout exists, agreed with the business, so performance can be measured at all.
11. Rollback is a switch somebody can throw without a rebuild.
12. Somebody's name is against the model, and there is a date to review it.

The part that is not code

Most models that fail in production fail for reasons on that list that have nothing to do with modelling: an upstream field silently changed units, a new product category appeared, nobody agreed who watches the dashboard, the threshold was set by a data scientist and never discussed with the team who work the flagged list. The modelling was the easy part and it was finished twelve weeks ago.

> **Day 7 takeaway**

Self-Assessment

Try It Yourself

1. A deployable model artefact must contain:

    a) The training data
    b) The notebook
    c) The estimator only
    d) The whole pipeline, including every preprocessing step

2. The contract file exists to record:

    a) Valid categories, training ranges, the agreed threshold, the version and the training date
    b) The cross-validation scores
    c) The feature importances
    d) The model weights

3. A smoke test with frozen inputs and expected outputs catches:

    a) Slow inference
    b) A dependency upgrade that silently changes your predictions
    c) Missing data
    d) Class imbalance

4. An unknown category arriving at a service with `handle_unknown='ignore'` is dangerous because it:

    a) Is logged as invalid

b) Raises an error

c) Is silently encoded as zeros and a plausible prediction is returned

d) Crashes the service

5. A Population Stability Index above 0.25 indicates:

   a) A broken calculation

   b) Improved accuracy

   c) A stable feature

   d) A material shift in the feature's distribution

6. Prediction drift is valuable to monitor because it:

   a) Needs no labels and is available the same day

   b) Replaces cross-validation

   c) Detects leakage

   d) Measures accuracy

7. Measuring a churn model's accuracy is complicated by the retention team acting on its output, because:

   a) The team is slow

   b) A correctly flagged customer who was persuaded to stay now counts as a false positive

   c) Labels are noisy

   d) The threshold changes

8. Before promoting a challenger model you should compare:

   a) The number of parameters

   b) Point estimates of AUC

   c) An interval on the difference, plus how many decisions would actually change

   d) Training time

Answers:1d, 2a, 3b, 4c, 5d, 6a, 7b, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 16

Capstone and Career

Week 16

OBJECTIVES

Frame a project of your own, build it to a standard somebody else can run, and be able to talk about it.

The last week has the least new machinery in it and the most that determines whether the previous fifteen were worth anything. It covers framing a problem around a decision rather than a dataset, structuring a project somebody else can run, four milestones for the capstone itself, how to mark your own work before a reviewer does, and how to talk about what you built.

Day 1 Framing a Project

Starting from a decision instead of a dataset, and writing the brief

Fifteen weeks of technique. This one is about turning it into something with your name on it, which means starting where every real project starts, and where almost every portfolio project does not: with a decision somebody has to make.

Frame the decision, not the dataset

Watch Out

The most common way a capstone fails

It begins "I found an interesting dataset". Everything after that is a search for a question the data happens to answer, and the result reads like it, a competent notebook with no argument in it. Start instead with a decision that is currently made badly, then ask what data would inform it. You may well end up at the same dataset. You will end up with a different project.

Weak framing	Strong framing	Why
"Predict house prices"	"Which listings are priced more than 10% below what comparable homes achieved, so an agent should call the vendor this week"	Names who acts, when, and on what
"Analyse customer churn"	"Which 200 customers should the retention team ring in October, given they can make 200 calls"	The budget constraint decides the threshold
"Classify images of plants"	"Let a grower photograph a leaf and get a disease shortlist without waiting for the agronomist"	Establishes what the alternative is
"Sentiment analysis of reviews"	"Route the 5% of reviews that describe a safety problem to the product team within an hour"	Makes recall on a rare class the metric

Write the brief before the code

1. **The decision.** Who does what differently because of this model?
2. **The alternative.** How is it decided today? That is your baseline, and it is rarely a machine learning model, usually it is a rule, a spreadsheet, or somebody's judgement.
3. **The unit.** One row is one what? Get this wrong and every split in the project is wrong.
4. **The label.** What exactly are you predicting, measured when? Write the SQL or the sentence that defines it.
5. **The moment of prediction.** What is known at that instant? Anything not known then cannot be a feature. This one sentence prevents most leakage.
6. **The metric, and the threshold.** Which errors cost what? Who bears them?
7. **What good enough looks like.** A number, agreed before you start, that means this was worth doing.

Point five is the whole of week 4 in one sentence

"What is known at the moment the prediction is made" resolves almost every leakage question without any statistics. `total_charges` is known at prediction time; `reason_for_leaving` is not. A customer's tenure is known; whether they answered the retention call is not. Write the list, then check every column you add against it.

Scoping it to fit

Scale	Rows	Looks like	Fits in
Small	< 10k	One table, a few features, a clear label	A weekend
Medium	10k to 1M	Two or three joined tables, some feature engineering	Two to three weeks
Large	> 1M	Sampling, chunked reads, a real time split	A month, and it will still surprise you

For a capstone, medium is the target. Small does not exercise enough of the course; large means you spend three weeks on data plumbing and produce a rushed model. If the dataset you

want is large, take a principled sample and say so in the write-up. That is a legitimate engineering decision, not a compromise.

Day 1 takeaway

Day 2 Structure and Reproducibility

Where the code lives, the four tests to write, and running it twice

A project somebody else can run is worth several times one they cannot, and the difference is about an hour of work done at the start rather than at the end.

The layout

The rule that makes the rest work

Notebooks are for narrative; `src/` is for anything that runs twice. The moment a cell is useful enough to copy into another notebook, it belongs in a module, where it can be imported, tested and fixed once. A project whose logic lives in notebook cells cannot be tested and cannot be deployed, and both of those are noticed immediately by anybody reviewing it.

Testing the parts that can silently break

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
def add_features(data):
    """The kind of function that belongs in src/features.py."""
    out = data.copy()
    out['charges_per_month_of_tenure'] = (
        out['total_charges'] / out['tenure_months'].clip(lower=1))
    out['calls_per_year'] = (
        out['support_calls'] * 12 / out['tenure_months'].clip(lower=1))
    out['is_new'] = (out['tenure_months'] <= 3).astype(int)
    return out

# tests/test_features.py -- these are the assertions that catch the
# mistakes nobody notices: a divide by zero, a column that silently
# disappears, a transformation that changes the number of rows.
def test_no_rows_lost():
    assert len(add_features(df)) == len(df)

def test_no_division_blowup():
    out = add_features(df)
```

```
    assert np.isfinite(out['calls_per_year'].dropna()).all()

def test_zero_tenure_is_survivable():
    edge = df.head(1).copy()
    edge['tenure_months'] = 0
    assert np.isfinite(add_features(edge)['calls_per_year']).all()

def test_original_columns_survive():
    out = add_features(df)
    assert set(df.columns).issubset(out.columns)

for test in [test_no_rows_lost, test_no_division_blowup,
             test_zero_tenure_is_survivable, test_original_columns_survive]:
    test()
    print('PASS %s' % test.__name__)
```

```
PASS test_no_rows_lost
PASS test_no_division_blowup
PASS test_zero_tenure_is_survivable
PASS test_original_columns_survive
```

Four assertions, no framework, about five minutes of work. They do not check that your model is good. Nothing can, but they catch the class of bug that produces a plausible model trained on quietly corrupted features, which is the class of bug that is hardest to find later.

Reproducibility

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
import hashlib
import json

def fingerprint(data):
    """Enough to prove two runs saw the same data."""
    digest = hashlib.sha256(
        pd.util.hash_pandas_object(data, index=True).values).hexdigest()
    return {'rows': len(data), 'columns': list(data.columns),
            'sha256': digest[:16]}

print(json.dumps(fingerprint(df), indent=2)[:300])
print('\nsame data again:', fingerprint(df)['sha256'])
print('one value changed:',
      fingerprint(df.assign(monthly_charges=df['monthly_charges'] + 0.01))
      ['sha256'])
```

```
{
```

```
  "rows": 3000,
  "columns": [
    "customer_id",
    "signup_date",
    "tenure_months",
    "contract",
    "internet_service",
    "payment_method",
    "has_dependents",
    "support_calls",
    "monthly_charges",
    "total_charges",
    "churned"
  ],
  "sha256": "48cf8b8b0d435748"
}

same data again: 48cf8b8b0d435748
one value changed: 0168e3b07660fcf9
```

1. Set every seed, and record it.
2. Fingerprint the input data so a rerun can prove it saw the same rows.
3. Pin dependencies.
4. Make the whole thing runnable from one command, a `Makefile`, or `python -m src.train`.
5. Never edit raw data in place. Read it, transform it, write elsewhere.
6. Keep `data/` out of version control, and say in the README where the data comes from.

Day 2 takeaway

Day 3 Building the Capstone

Four milestones with artefacts, and where the time really goes

The capstone itself, as a sequence of checkpoints. Each has an artefact and a decision, so it is possible to know whether you have finished it.

Milestone 1: the brief and the baseline

- The brief from day 1, written out, one page.
- The data loaded, the unit confirmed, the label defined in code.
- A split chosen and justified, random, grouped or by time.
- **A baseline that is not a model:** the current rule, the majority class, last month's value, or a two-line heuristic.
- The metric computed for that baseline.

The baseline is the most valuable number in the project

Without it, "AUC 0.81" means nothing. With it, "AUC 0.81 against the current rule's 0.68, on customers the rule cannot score at all" is a result. The baseline is also how you find out early that the problem is already solved, which is worth knowing in week one rather than week four.

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, f1_score

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)

# The rule the retention team uses today: month-to-month contract and
# more than two support calls.
rule = ((X_te['contract'] == 'Month-To-Month')
        & (X_te['support_calls'] > 2)).astype(int)

print('the existing rule')
print('  flags        %d of %d customers' % (rule.sum(), len(rule)))
print('  AUC          %.4f' % roc_auc_score(y_te, rule))
print('  F1           %.4f' % f1_score(y_te, rule))
print('  of those flagged, %.1f%% actually churned'
      % (100 * y_te[rule == 1].mean()))
print('  of all churners, it catches %.1f%%'
      % (100 * rule[y_te == 1].mean()))
```

```
the existing rule
  flags        51 of 750 customers
  AUC          0.5351
  F1           0.1905
  of those flagged, 47.1% actually churned
  of all churners, it catches 11.9%
```

Milestone 2: honest evaluation

- A pipeline, with every transformation inside it.
- Cross-validation appropriate to the data, from week 7.
- Two or three model families compared, including a linear one.
- A held-out set touched exactly once.
- The threshold chosen from the cost of the two errors, not left at 0.5.

Milestone 3: improvement, measured

- Features added one group at a time, each measured against the baseline.
- **The failures kept in the write-up.** "I tried target encoding on payment method; it cost 0.004 AUC and I dropped it" is evidence of judgement.
- Tuning, with the budget argument from week 14.
- A final held-out number, reported once.

Milestone 4: explain and ship

- Which features drive it, with the shape of the top effects.
- Two or three individual predictions explained, including a wrong one.
- An error analysis by segment, who does it fail?
- The artefact and contract from week 15.
- A README somebody can follow without asking you anything.

Where the time actually goes

```
phases = [('Framing and data understanding', 20),
          ('Cleaning and joining', 30),
          ('Feature engineering', 20),
          ('Modelling and tuning', 15),
          ('Evaluation and error analysis', 10),
          ('Writing it up', 5)]
for name, pct in phases:
    print('%-34s %3d%%  %s' % (name, pct, '#' * (pct // 2)))
print('\nthe part everyone plans for is the 15%.')
```

```
Framing and data understanding      20%  ##########
Cleaning and joining                30%  ###############
Feature engineering                 20%  ##########
Modelling and tuning                15%  #######
Evaluation and error analysis       10%  #####
Writing it up                        5%  ##

the part everyone plans for is the 15%.
```

Watch Out

Budget for the write-up before you need it

Five percent is the honest average and it is too little. A project that is finished but unexplained is worth roughly nothing to a reader, and the write-up is the only part of it most people will ever see. Book the last day of your capstone for writing, and treat it as non-negotiable.

Day 3 takeaway

Day 4 Marking Your Own Work

The rubric, the self-check, and the one-page write-up

How to mark your own work, and the specific things a reviewer looks for first.

The rubric

Area	Falls short	Meets	Exceeds
Framing	A dataset with a model on it	A stated decision and user	The decision, the current alternative, and the cost of each error
Data	`dropna()` and onwards	Missingness, duplicates and outliers handled deliberately	Each choice justified and its effect measured
Validation	One random split	Cross-validation matched to the data's structure	Grouped or time-based splitting, argued from how the data arose
Leakage	Not considered	Transformations inside the pipeline	An explicit list of what is known at prediction time
Baseline	None	A sensible simple model	The actual current process, measured
Metric	Accuracy	A metric matched to the problem	A threshold chosen from costs, with the trade-off shown
Interpretation	A feature importance bar chart	Importances plus the shape of the main effects	Individual cases explained, including errors, with limits stated
Engineering	One long notebook	Modules, seeds, pinned versions	Tests, one-command run, a saved artefact and contract
Communication	Cells and outputs	A README and a clear findings section	Written for the decision-maker, with the caveats in it

Marking yourself honestly

```
checks = [
    ('Could a reader state the decision this informs?', None),
    ('Is there a non-model baseline, measured?', None),
    ('Is every transformation inside the pipeline?', None),
    ('Was the held-out set used exactly once?', None),
    ('Is the threshold justified by costs?', None),
    ('Are failed experiments written down?', None),
    ('Does the write-up state who the model fails?', None),
    ('Can somebody else run it from the README?', None),
    ('Are the seeds set and the versions pinned?', None),
    ('Is there a sentence about what would make it wrong?', None),
```

```
]
print('Score one point each. Below 7 and the reviewer will find it')
print('before you do.\n')
for i, (question, _) in enumerate(checks, 1):
    print('%2d. [ ] %s' % (i, question))
```

```
Score one point each. Below 7 and the reviewer will find it
before you do.

 1. [ ] Could a reader state the decision this informs?
 2. [ ] Is there a non-model baseline, measured?
 3. [ ] Is every transformation inside the pipeline?
 4. [ ] Was the held-out set used exactly once?
 5. [ ] Is the threshold justified by costs?
 6. [ ] Are failed experiments written down?
 7. [ ] Does the write-up state who the model fails?
 8. [ ] Can somebody else run it from the README?
 9. [ ] Are the seeds set and the versions pinned?
10. [ ] Is there a sentence about what would make it wrong?
```

The three things looked at first

From experience of reading these: **the README**, can I tell in thirty seconds what this is and what it found; **the validation**, is the reported number believable, or is it leaking; and **the baseline**, is the improvement over anything. A project that is strong on those three survives being weak elsewhere. A project that is weak on them does not recover, however good the modelling is.

The write-up

1. **The problem**, in two sentences, naming the decision.
2. **The data**: where from, how much, what is wrong with it.
3. **The approach**: the split, the baseline, what you tried.
4. **The result**: one headline number against the baseline, with the metric named and the uncertainty given.
5. **What it means**: for the decision in point one, in the reader's language.
6. **Where it fails**: which segments, which inputs, what would break it.
7. **What you would do next** with another two weeks.

Seven sections, one page. If it runs longer than a page, the model is probably being described where the finding should be. Nobody outside the team needs your hyperparameters; everybody needs to know whether to trust the number and what to do with it.

Day 4 takeaway

Day 5 The Portfolio

Three or four projects, and a README that leads with the result

What to do with sixteen weeks of work once it exists: the portfolio, and the specific ways it fails to land.

Three or four projects, not ten

Project	Shows	Drawn from
A tabular prediction, end to end	Judgement: framing, leakage, baselines, thresholds	Weeks 1 to 8, 14
A deployed model with a live endpoint	That you have shipped something	Week 15
A perceptual project, images or text	That you can work where features cannot be written by hand	Weeks 12 to 13
One deep analysis with a surprising finding	Curiosity, and that you can write	Weeks 2, 7, 9 to 10

Watch Out

The portfolio mistakes that are easy to avoid

The famous datasets, Titanic, Iris, Boston housing, have been done ten thousand times and cannot distinguish you; use them to learn and not to demonstrate. A repository of notebooks with no README is unreadable. A 0.99 accuracy is read as leakage, not as success, so if you genuinely have one, explain immediately why it is real. And a project with no failures in it reads as a project where nothing was tried.

The README that gets read

Result before instructions, instructions before detail, limitations stated rather than buried. A reader who stops after the first two sections still knows what the project is and whether it worked, which is the only realistic goal. Those are the numbers this course actually measured, day 3 fitted the rule, day 6 ranked under a capacity constraint, so the example is a real README for the work you have already done, not a template with invented figures in it.

Talking about it

1. Lead with the decision and the baseline, never with the algorithm.
2. Give the headline number with its comparison attached, a number on its own invites the question you should have answered.
3. Have one specific difficulty ready, and how you resolved it. This is the question that separates people who did the work from people who followed a tutorial.
4. Know why you chose the model you chose, and what you rejected.
5. Be able to say what would make it wrong.
6. Keep it to two minutes unless asked for more.

Day 5 takeaway

Day 6 Interviews

The questions, what they test, and the ranking answer

The questions that actually get asked, grouped by what they are really testing. Your capstone is the answer to most of them.

Fundamentals

Question	What is being tested	Where it was covered
Explain the bias-variance trade-off	Whether you understand generalisation or memorised a phrase	Week 4
Your model has 99% accuracy. What do you check?	Leakage instincts, and class imbalance	Weeks 4, 7
When would you not use accuracy?	Whether metrics are chosen or defaulted	Weeks 5, 7
Why does regularisation help?	Whether you can connect the penalty to the variance	Week 4
How do you handle missing data?	Whether imputation happens inside the pipeline	Weeks 2, 8
Random forest versus gradient boosting	Whether you know why the averaging differs	Week 6
What does a p-value mean?	Precision of language under mild pressure	Week 3

Applied

Question	What a strong answer contains
Walk me through a project	The decision, the baseline, one difficulty, the result with its comparison, and a limitation
How would you detect data leakage?	The prediction-time rule; suspiciously high scores; feature importance dominated by one column; a time-based split as the check
A model works in testing and fails in production. Why?	Drift, leakage in training, a train/serve preprocessing mismatch, or a population that differs from the training sample
How do you choose a threshold?	From the relative cost of the two errors and the operational capacity, not from 0.5
How would you explain this model to a non-technical stakeholder?	Effects rather than coefficients, one worked example, and the limits stated plainly
When would you not use machine learning?	A rule works; there are too few examples; the decision must be auditable; the cost of an error is catastrophic

The last one is a good question and a good answer

Week 12 ended with a support vector machine beating a convolutional network, and week 15 with a boosted challenger losing to logistic regression. Being able to say "I have measured a case where the simple thing won, and shipped the simple thing" is a stronger signal than any list of architectures. Enthusiasm for complexity is common; judgement about when to avoid it is not.

A whiteboard exercise you can practise now

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
# "Design a system to decide which customers to call this month."
# Talk through it before writing anything. The structure below is the
# answer; the code is just proof it is not hand-waving.
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

CAPACITY = 200          # the constraint that decides everything

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
])
X_tr, X_te, y_tr, y_te = train_test_split(
    df[NUM + CAT], df['churned'], test_size=0.25,
    stratify=df['churned'], random_state=42)
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))]).fit(X_tr, y_tr)

risk = model.predict_proba(X_te)[:, 1]
order = np.argsort(-risk)
called = order[:CAPACITY]

print('%d customers, capacity %d' % (len(X_te), CAPACITY))
print('churners in the whole test set:      %d' % y_te.sum())
print('churners among the %d we would call: %d'
      % (CAPACITY, y_te.iloc[called].sum()))
print('  hit rate %.1f%% against a base rate of %.1f%%'
      % (100 * y_te.iloc[called].mean(), 100 * y_te.mean()))
print('  lift %.2fx' % (y_te.iloc[called].mean() / y_te.mean()))
print('\nnote what was never chosen: a threshold. Capacity chose it.')
```

```
750 customers, capacity 200
churners in the whole test set:      201
churners among the 200 we would call: 116
  hit rate 58.0% against a base rate of 26.8%
  lift 2.16x

note what was never chosen: a threshold. Capacity chose it.
```

That last line is the point of the exercise. A great many real deployments are ranking problems with a capacity constraint, not classification problems with a threshold, and answering in those terms tells an interviewer you have thought about how the output gets used.

Day 6 takeaway

Day 7 Where This Leaves You

What was covered, what was not, and the habits worth keeping

Sixteen weeks. What was actually covered, what deliberately was not, and where to go next.

What you can now do

Weeks	Capability
1-3	Python and pandas for data work, and the linear algebra, calculus and probability underneath the models
4-6	Regression and classification, regularisation, trees and ensembles, and why the simple model sometimes wins
7-8	Validation you can trust, and features that beat model choice
9-10	Clustering, dimensionality reduction and anomaly detection without labels
11-13	Neural networks from the gradient up, convolution for images, embeddings and sequences for text
14-15	Tuning within a budget, explaining a model, and running one in production
16	Framing, shipping and communicating a project of your own

What this course did not cover

- **Reinforcement learning.** A different problem shape entirely, learning from interaction rather than from a fixed dataset.
- **Causal inference.** Week 14 drew the line and stopped at it. Everything past that line, experiments, instrumental variables, difference-in-differences, is a field of its own and the natural next one for anybody doing this work in a business.
- **Training large models.** Distributed training, mixed precision and the engineering of very large networks.
- **Recommender systems.** Collaborative filtering has its own evaluation traps.
- **Time series forecasting.** Week 7 covered splitting by time; forecasting itself is a separate discipline.
- **Fine-tuning language models.** Week 13 pointed at it. The mechanics move fast; the judgement in weeks 7 and 14 does not.

Where to go next

If you want to	Learn next
Answer "did it work" rather than "what will happen"	Causal inference and experiment design
Handle data that does not fit in memory	Spark or DuckDB, and columnar formats
Own the systems as well as the models	Docker, CI, orchestration, Airflow or Dagster
Work on language	Transformers and fine-tuning, on top of week 13
Go deeper on the theory	Convex optimisation and statistical learning theory
Get better fastest, whatever the direction	Ship three more projects end to end

The habits worth keeping

Fit the baseline first. Ask what is known at the moment of prediction. Put every transformation inside the pipeline. Read the confusion matrix. Choose the threshold from the cost of the errors. Write down the experiments that failed. Say who the model fails. Prefer the model you can explain when the numbers are close.

None of those depend on a library version, and all of them will still be true when whatever replaces today's tools has replaced them.

A last measurement

```
import numpy as np
import pandas as pd

df = pd.read_csv('customers.csv').drop_duplicates().copy()
df['contract'] = df['contract'].str.strip().str.title()
df['total_charges'] = pd.to_numeric(df['total_charges'], errors='coerce')

NUM = ['tenure_months', 'support_calls', 'monthly_charges']
CAT = ['contract', 'internet_service', 'payment_method', 'has_dependents']
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.metrics import roc_auc_score

X, y = df[NUM + CAT], df['churned']
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25,
                                          stratify=y, random_state=42)

# The rule they had in week 1.
rule = ((X_te['contract'] == 'Month-To-Month')
        & (X_te['support_calls'] > 2)).astype(int)

prep = ColumnTransformer([
    ('num', Pipeline([('i', SimpleImputer(strategy='median')),
                      ('s', StandardScaler())]), NUM),
    ('cat', Pipeline([('i', SimpleImputer(strategy='most_frequent')),
                      ('o', OneHotEncoder(handle_unknown='ignore'))]), CAT),
```

```
])
model = Pipeline([('prep', prep),
                  ('clf', LogisticRegression(max_iter=2000))]).fit(X_tr, y_tr)
cv = cross_val_score(model, X_tr, y_tr, cv=5, scoring='roc_auc')

print('the rule in use            AUC %.4f' % roc_auc_score(y_te, rule))
print('logistic regression, cv    AUC %.4f +/- %.4f' % (cv.mean(), cv.std()))
print('logistic regression, test  AUC %.4f'
      % roc_auc_score(y_te, model.predict_proba(X_te)[:, 1]))
print('\nthirteen lines. everything else this course taught you was')
print('about knowing whether to believe the last number.')
```

```
the rule in use            AUC 0.5351
logistic regression, cv    AUC 0.8217 +/- 0.0110
logistic regression, test  AUC 0.8174

thirteen lines. everything else this course taught you was
about knowing whether to believe the last number.
```

Course takeaway

Self-Assessment

Try It Yourself

1. The strongest way to begin a project is with:

   a) A Kaggle competition
   b) An interesting dataset
   c) A decision somebody currently makes badly
   d) A model architecture

2. "What is known at the moment the prediction is made" is the question that prevents:

   a) Class imbalance
   b) Underfitting
   c) Overfitting
   d) Target leakage

3. In a project layout, code that runs more than once belongs in:

   a) An importable module under `src/`
   b) A markdown cell
   c) The README
   d) The notebook

4. The most valuable single number in a capstone is:

   a) The final AUC

b) The baseline it is compared against
c) The cross-validation standard deviation
d) The training time

5. Roughly what share of a real project is framing, cleaning and joining data?

a) About 90%
b) About 15%
c) About half
d) About 5%

6. Including experiments that failed in your write-up:

a) Is only for academic work
b) Should be an appendix
c) Weakens it
d) Is evidence of judgement

7. A reviewer looks first at:

a) The README, the validation and the baseline
b) The hyperparameters
c) The number of commits
d) The model class

8. Many real deployments are best framed as:

a) Classification with a 0.5 threshold
b) Ranking under a capacity constraint, where capacity chooses the threshold
c) Clustering
d) Regression

Answers:1c, 2d, 3a, 4b, 5c, 6d, 7a, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.