EDUSHARK TRAINING

Self-paced programmes in data and AI

Artificial Intelligence

A 14-Week Introduction

What AI is, what it is not, and how to tell the difference.

edusharktraining.com

Artificial Intelligence: A 14-Week Introduction

Publisher: EDUSHARK TRAINING
Web: https://edusharktraining.com/artificial-intelligence
Contact: info@edusharktraining.com

This workbook is the offline form of a course published in full and free to read at edusharktraining.com. It may be passed on to anyone, including a whole team. All trademarks are the property of their respective owners.

Typeset with LaTeX.

How to Use This Workbook

This book covers 14 weeks, 98 lessons and 112 self-assessment questions. No maths background required. No prior AI.

What is different about it

Rules, search and learning are taken apart on the same problem, and the comparisons were run rather than asserted. Several of them favour the unglamorous option, and this book keeps those results.

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/artificial-intelligence. The website version carries the same material with the code laid out for copying. Use whichever suits where you are reading.

Contents

How to Use This Workbook i

1 What Artificial Intelligence Actually Is 1

2 Measuring a Model Honestly 29

3 Search, Knowledge and Reasoning 43

4 Text as Data 59

5 The Standard Language Tasks 81

6 Meaning and Embeddings 99

7 Transformers and Large Language Models 119

8 Images as Data 133

9 Classifying Images 147

10 Beyond Classification 163

11 Bias and Fairness 179

12 Explaining What a Model Did 195

13 Privacy, Safety and Governance 213

14 The Capstone 225

Chapter 1

What Artificial Intelligence Actually Is

Week 1

OBJECTIVES

Rules, search and learning taken apart and compared on the same problem, with the habits that make any result trustworthy.

The word AI covers at least three completely different mechanisms, and knowing which one a problem needs is worth more than knowing any particular algorithm. This week builds all three, solves the same support inbox with hand-written rules and with a learned model, and shows the learning loop running in the open so there is nothing mysterious left in it. It ends with the two habits the rest of the course depends on: measure on data the system has not seen, and put a do-nothing baseline beside every number.

Day 1 Three Things All Called AI

Rules, search and learning, and why the distinction is practical

Artificial intelligence is a label attached to a very wide range of things, and the first useful thing this course can do is take it apart. A thermostat is not intelligent. A chess program that beats you is doing something you would call thinking if a person did it. A system that writes an essay is doing something different again. All three have been called AI at some point.

Three systems, three completely different mechanisms

Rather than argue about definitions, it is more useful to see the three families of approach side by side, because they fail in different ways and suit different problems.

One: rules a person wrote down

Somebody works out the logic and encodes it. The system is exactly as good as the person who wrote it, it can explain every decision, and it knows nothing that was not written into it.

```
def triage(temperature, breathing_hard, age):
    """The kind of system that ran hospitals and helpdesks for
    decades, and still runs a great deal of both."""
    if temperature > 39.0 and breathing_hard:
        return 'urgent'
    if temperature > 38.0 and age > 65:
        return 'urgent'
    if temperature > 38.0:
        return 'same day'
    return 'routine'
```

```

cases = [(39.4, True, 30), (38.2, False, 70), (38.2, False, 30),
         (37.1, False, 45)]
for temp, breathing, age in cases:
    print('%.1f  breathing hard %-5s  age %2d  →  %s'
          % (temp, breathing, age, triage(temp, breathing, age)))
```

```
39.4  breathing hard True   age 30  →  urgent
38.2  breathing hard False  age 70  →  urgent
38.2  breathing hard False  age 30  →  same day
37.1  breathing hard False  age 45  →  routine
```

Every decision there is traceable to a line you can point at. That is a real advantage and it is why rules still run anything with a regulator attached. The limit is equally real: nobody can write down the rules for recognising a cat in a photograph.

Two: searching through possibilities

The computer is not told the answer, it is told the rules of the game and how to recognise a solution, and then it looks. This is how route planners and puzzle solvers work, and it is how chess programs beat people long before machine learning was involved.

```
from collections import deque

ROADS = {'depot': ['north', 'east'], 'north': ['ring', 'depot'],
         'east': ['ring', 'south'], 'ring': ['airport'],
         'south': ['airport'], 'airport': []}

def shortest(start, goal):
    """Breadth first search. No learning, no data, no training."""
    queue = deque([[start]])
    seen = {start}
    while queue:
        path = queue.popleft()
        if path[-1] == goal:
            return path
        for nxt in ROADS[path[-1]]:
            if nxt not in seen:
                seen.add(nxt)
                queue.append(path + [nxt])
    return None

print(' → '.join(shortest('depot', 'airport')))
print('the program was never shown a single example route')
```

```
depot → north → ring → airport
the program was never shown a single example route
```

Nobody gave that program any examples. It was given the map and a way to tell when it had arrived, and it worked the rest out. When a problem genuinely has that shape, search is often better than learning: it is exact, it needs no training data, and it can prove there is no shorter route.

Three: learning the rules from examples

Nobody writes the logic. You supply examples of the answer and the system works out for itself what distinguishes them. This is machine learning, it is what almost everyone means by AI today, and days 2 to 7 are about it.

> **Why the distinction matters on day 1**

Day 2 Rules Against Examples

The same inbox solved both ways, and measured

The clearest way to feel the difference between writing rules and learning them is to do both on the same problem and compare. The problem is a support inbox of short messages, half of them unwanted.

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
```

```
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
for message, label in list(zip(messages, labels))[:6]:
    print('%-5s %s' % ('spam' if label else 'ok', message))
print()
print('%d messages, %d spam, %d for training and %d held back'
      % (len(messages), sum(labels), len(train_x), len(test_x)))
```

```
spam  note limited deal, 75% discount if you order now fyi
ok    note thanks for the figures, they match the password i have quick
spam  quick claim your new phone before it expires today hi
ok    thanks the report is finished, it is in the shared folder note
spam  cheers congratulations you have won a gift card, claim it now morning
ok    fyi their offer came in within 24 hours, i think we should accept please
```

```
360 messages, 180 spam, 240 for training and 120 held back
```

How this set is held back matters as much as that it is

The obvious split is to shuffle the messages and keep a quarter back. Do that here and every model scores a perfect 1.000, because the held out messages are near duplicates of training ones and remembering is enough.

So the split is by *phrasing* instead. The training set uses ten templates a side and the held out set uses four completely different ones. That is the situation a real inbox presents: tomorrow's mail is not a reshuffle of today's, and the only interesting question is whether the system handles wording it has never seen.

The rules a sensible person would write

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
```

```
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
BANNED = ['won', 'free', 'claim', 'offer', 'discount', 'urgent',
          'verify', 'exclusive', 'selected', 'prize']

def rule_predict(message):
    """Somebody sat down and wrote what spam looks like."""
    words = message.lower().replace(',', '').split()
    return 1 if any(word in BANNED for word in words) else 0

def score(predict, xs, ys):
    right = sum(1 for x, y in zip(xs, ys) if predict(x) == y)
    caught = sum(1 for x, y in zip(xs, ys) if y == 1 and predict(x) == 1)
    flagged = sum(1 for x, y in zip(xs, ys) if y == 0 and predict(x) == 1)
    spam = sum(ys)
```

```
    return right / len(xs), caught / spam, flagged / (len(ys) - spam)
acc, caught, wrongly = score(rule_predict, test_x, test_y)
print('%-28s %8s %8s %8s' % ('', 'accuracy', 'caught', 'false'))
print('%-28s %8.3f %8.3f %8.3f' % ('keyword rules', acc, caught, wrongly))
print()
print('spam the rules let through:')
shown = 0
for message, label in zip(test_x, test_y):
    if label == 1 and rule_predict(message) == 0 and shown < 4:
        print('   ' + message)
        shown += 1
```

```
                            accuracy   caught    false
keyword rules                  0.492    0.200    0.217

spam the rules let through:
   fyi a 75% refund is pending, tell us where to send it morning
   best a 90% refund is pending, tell us where to send it team
   thanks a 75% refund is pending, tell us where to send it note
   regards act before friday to release the 1000 pounds waiting on your account thanks
```

Ten sensible keywords, and on mail phrased in ways they were not written for they catch a fifth of the spam and score 0.492 overall, which is worse than tossing a coin. Read the messages they let through: every one is obviously unwanted to a human and contains none of the ten words.

This is not a contrived defeat. It is the actual dynamic of the problem, because whoever is sending the spam can read the filter too, and the phrasings that survive are exactly the ones that avoid the list. Each miss is an argument for adding another keyword, and that argument never ends. The rules also block ordinary mail: one of the held out messages is somebody saying they *won* an argument about a deadline.

The same problem, learned

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
```

```
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
```

```
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
BANNED = ['won', 'free', 'claim', 'offer', 'discount', 'urgent',
          'verify', 'exclusive', 'selected', 'prize']

def rule_predict(message):
    """Somebody sat down and wrote what spam looks like."""
    words = message.lower().replace(',', '').split()
    return 1 if any(word in BANNED for word in words) else 0

def score(predict, xs, ys):
    right = sum(1 for x, y in zip(xs, ys) if predict(x) == y)
    caught = sum(1 for x, y in zip(xs, ys) if y == 1 and predict(x) == 1)
    flagged = sum(1 for x, y in zip(xs, ys) if y == 0 and predict(x) == 1)
    spam = sum(ys)
    return right / len(xs), caught / spam, flagged / (len(ys) - spam)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

model = make_pipeline(TfidfVectorizer(), LogisticRegression())
model.fit(train_x, train_y)

def learned_predict(message):
    return int(model.predict([message])[0])

print('%-28s %8s %8s %8s' % ('', 'accuracy', 'caught', 'false'))
for name, fn in [('keyword rules', rule_predict),
                 ('learned from examples', learned_predict)]:
    acc, caught, wrongly = score(fn, test_x, test_y)
    print('%-28s %8.3f %8.3f %8.3f' % (name, acc, caught, wrongly))
```

```
                            accuracy   caught    false
keyword rules                  0.492    0.200    0.217
learned from examples          0.883    0.983    0.217
```

0.883 against 0.492, on wording neither system has seen. Nobody told the model that *claim* or *congratulations* matter. It was shown the training messages and worked out for itself which words carry signal and how much, which is a far more forgiving thing to carry into unfamiliar phrasing than a list of ten words that either appear or do not.

Note also what the model did not do. It scored 0.883, not 1.000. Unfamiliar wording costs it too, and a system that is honest about this is worth more than one that is only ever tested on mail like the mail it learned from.

And look at the last column, which is the one people skip. Both systems wrongly flag ordinary mail at exactly the same rate, 0.217. The model's entire advantage is in the middle column: it catches almost all of the spam where the rules catch a fifth, at no extra cost in blocked invoices. That is a much more precise claim than *the model is better*, and it is the kind of claim worth learning to make.

What the model decided was important

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
```

```
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
BANNED = ['won', 'free', 'claim', 'offer', 'discount', 'urgent',
          'verify', 'exclusive', 'selected', 'prize']

def rule_predict(message):
    """Somebody sat down and wrote what spam looks like."""
    words = message.lower().replace(',', '').split()
    return 1 if any(word in BANNED for word in words) else 0

def score(predict, xs, ys):
    right = sum(1 for x, y in zip(xs, ys) if predict(x) == y)
    caught = sum(1 for x, y in zip(xs, ys) if y == 1 and predict(x) == 1)
    flagged = sum(1 for x, y in zip(xs, ys) if y == 0 and predict(x) == 1)
    spam = sum(ys)
    return right / len(xs), caught / spam, flagged / (len(ys) - spam)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

vec = TfidfVectorizer()
X = vec.fit_transform(train_x)
clf = LogisticRegression().fit(X, train_y)
words = vec.get_feature_names_out()
weights = clf.coef_[0]
order = weights.argsort()
print('%-16s %10s' % ('most spam-like', 'weight'))
for i in order[::-1][:6]:
    print('%-16s %10.3f' % (words[i], weights[i]))
print()
print('%-16s %10s' % ('most ordinary', 'weight'))
for i in order[:6]:
    print('%-16s %10.3f' % (words[i], weights[i]))
```

```
most spam-like       weight
your                  2.201
it                    1.184
now                   0.945
claim                 0.926
today                 0.895
order                 0.775

most ordinary        weight
the                  -2.881
we                   -1.165
should               -1.163
in                   -1.003
is                   -0.854
meeting              -0.766
```

That table is worth pausing on, because it shows the mechanism is not magic. The model learned a weight per word and adds them up. Some of those words you would have guessed. Others are there because of how this particular inbox is written, which is exactly the knowledge a person writing rules would have had to acquire by reading all two hundred messages.

Watch Out

And that is also the weakness

The model learned this inbox. Words that signal spam here may signal nothing elsewhere, and a spammer who reads the table can avoid every word in it. Rules have the same weakness and at least they are visible. A learned model's assumptions are implicit, which is why weeks 11 to 13 are about getting them back out.

Day 3 What a Learning System Is Made Of

Examples, labels, features, a model and a loss, with the loop running in the open

Yesterday's model was four lines. Underneath those four lines is a structure that every supervised learning system shares, and naming its parts makes the rest of the course much easier to follow.

The five parts

1. **Examples.** Things you have seen before. Here, messages.
2. **Labels.** The answer for each example. Here, spam or not. Somebody had to provide these, and that cost is usually the real constraint on a project.
3. **Features.** The numbers the model actually sees. A message is text; the model needs numbers, so something has to turn one into the other.
4. **A model.** A function with adjustable settings, mapping features to a prediction.
5. **A loss and a way to reduce it.** A number saying how wrong the predictions are, and a procedure that adjusts the settings to make it smaller.

Features, made visible

The step that surprises people most is the third one, so here it is explicitly. This is what the model receives instead of a sentence.

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
```

```
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
from sklearn.feature_extraction.text import CountVectorizer

sample = ['claim your free prize today', 'the invoice is attached']
vec = CountVectorizer()
counts = vec.fit_transform(sample)
print('vocabulary: %s' % list(vec.get_feature_names_out()))
print()
for text, row in zip(sample, counts.toarray()):
    print('%-32s %s' % (text, row))
```

```
vocabulary: ['attached', 'claim', 'free', 'invoice', 'is', 'prize', 'the', 'today', 'your']

claim your free prize today      [0 1 1 0 0 1 0 1 1]
the invoice is attached          [1 0 0 1 1 0 1 0 0]
```

The sentence has become a row of counts, one column per word in the vocabulary. Word order is gone entirely, which is why this representation is called a bag of words. It is crude, and it is enough to have beaten the rules yesterday.

Watching the loss come down

The last part is the training loop. Most libraries hide it, so here it is running in the open on the same data, with the error printed every few passes.

```
import random
```

```

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']
```

```

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

vec = TfidfVectorizer()
X = vec.fit_transform(train_x).toarray()
y = np.array(train_y)
w = np.zeros(X.shape[1])
b = 0.0

print('%6s %12s %10s' % ('pass', 'loss', 'accuracy'))
for step in range(1, 301):
    z = X @ w + b
    guess = 1 / (1 + np.exp(-z))
    # how wrong we are, averaged over every training message
    loss = -np.mean(y * np.log(guess + 1e-9)
                    + (1 - y) * np.log(1 - guess + 1e-9))
    error = guess - y
    w -= 0.5 * (X.T @ error) / len(y)
    b -= 0.5 * error.mean()
    if step % 60 == 0:
        acc = ((guess > 0.5).astype(int) == y).mean()
        print('%6d %12.4f %10.4f' % (step, loss, acc))
```

```
  pass         loss   accuracy
    60       0.4286     1.0000
   120       0.2976     1.0000
   180       0.2243     1.0000
   240       0.1785     1.0000
   300       0.1475     1.0000
```

That is the whole of learning, in eight lines. Make a prediction, measure how wrong it is, nudge

every setting in the direction that makes it less wrong, repeat. Every model in this course and every model in the news does a version of this, differing in the shape of the function and the size of the numbers rather than in the idea.

Day 4 Generalisation

Memorising against learning, and the baseline nobody runs

A model that scores well on the data it learned from has demonstrated nothing. It could have memorised. The entire question is whether it works on examples it has never seen, and that question has a name.

Memorising, demonstrated

Here is a model deliberately given enough freedom to memorise, scored both ways.

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
```

```
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression

vec = TfidfVectorizer()
Xtr = vec.fit_transform(train_x)
Xte = vec.transform(test_x)

print('%-34s %12s %12s' % ('', 'on training', 'on held out'))
for name, model in [('a deep decision tree',
                     DecisionTreeClassifier(random_state=0)),
                    ('logistic regression',
                     LogisticRegression())]:
    model.fit(Xtr, train_y)
    print('%-34s %12.4f %12.4f'
```

```
          % (name, model.score(Xtr, train_y), model.score(Xte, test_y)))
```

```
                                  on training  on held out
a deep decision tree                   1.0000       0.7333
logistic regression                    1.0000       0.8833
```

Both models score a perfect 1.000 on the messages they were built from, and that column tells you nothing whatsoever. The right hand column is where they separate: 0.733 for the tree against 0.883 for logistic regression.

Same data, same features, same perfect training score, and one of them carries fifteen points more of it across to unfamiliar wording. The tree memorised harder, in the sense that more of what it learned was specific to the exact messages in front of it. The gap between those two columns is the thing to watch for the rest of your career, and it is invisible if you only ever look at the left one.

The baseline nobody runs

Before celebrating any accuracy, you need to know what a system that does nothing clever would score. Otherwise the number has no meaning at all.

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
```

```
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
from sklearn.dummy import DummyClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

vec = TfidfVectorizer()
Xtr, Xte = vec.fit_transform(train_x), vec.transform(test_x)
print('%-34s %12s' % ('', 'held out'))
for name, model in [('always guess the commoner class',
                     DummyClassifier(strategy='most_frequent')),
```

```
                      ('guess at random, in proportion',
                       DummyClassifier(strategy='stratified',
                                       random_state=0)),
                      ('the actual model', LogisticRegression())]:
    model.fit(Xtr, train_y)
    print('%-34s %12.4f' % (name, model.score(Xte, test_y)))
```

```
                                    held out
always guess the commoner class       0.5000
guess at random, in proportion        0.4750
the actual model                      0.8833
```

Watch Out

An accuracy without a baseline is not a result

This inbox is deliberately balanced, so guessing gets about half and the model's score is clearly meaningful. Real data is rarely so obliging. If one in a hundred transactions is fraud, a model that says *not fraud* every single time is 99 percent accurate and completely worthless, and it is reported as a success more often than you would believe.
Always run the do-nothing baseline first. It takes one line and it tells you what your real number has to beat.

Day 5 The Shapes a Problem Comes In

Classification, regression, ranking, clustering and generation

Almost every applied AI problem is one of a small number of shapes. Recognising which shape you have tells you what kind of data you need, what model to reach for, and how to measure success, so it is worth learning the list properly.

The shapes

Shape	You are predicting	An example	Measured with
Binary classification	One of two categories	Is this message spam	Accuracy, precision, recall
Multi-class classification	One of several categories	Which of ten digits is this	Accuracy, confusion matrix
Regression	A number	What will this house sell for	Average error in the same units as the answer
Ranking	An order	Which results to show first	How high the good ones land
Clustering	Groups, with no labels given	Which customers behave alike	Judgement, mostly
Generation	New content	Write a summary of this	Difficult, and week 7 returns to it

The same library, three different shapes

```
from sklearn.datasets import load_diabetes, load_digits
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
import numpy as np

# regression: predict a number
X, y = load_diabetes(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.25,
                                      random_state=0)
pred = LinearRegression().fit(Xtr, ytr).predict(Xte)
print('regression, average error   %.1f units' % np.abs(pred - yte).mean())
print('   guessing the average is  %.1f units'
      % np.abs(ytr.mean() - yte).mean())

# multi-class classification: predict which of ten
X, y = load_digits(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.25,
                                      random_state=0)
clf = RandomForestClassifier(n_estimators=60, random_state=0).fit(Xtr, ytr)
print('classification, accuracy    %.4f' % clf.score(Xte, yte))
print('   guessing would give      %.4f' % (1 / 10))

# clustering: no labels used at all
groups = KMeans(n_clusters=10, n_init=10, random_state=0).fit_predict(X)
agree = max((groups == g).sum() for g in range(10))
print('clustering found 10 groups, largest holds %d of %d images'
      % (agree, len(X)))
```

```
regression, average error   45.1 units
   guessing the average is  58.3 units
classification, accuracy    0.9778
   guessing would give      0.1000
clustering found 10 groups, largest holds 247 of 1797 images
```

Three problems, three shapes, one library, and in each case a do-nothing comparison next to the result. Notice that the clustering line has no accuracy at all, because no labels were used, and that is the honest situation rather than an omission.

Getting the shape wrong is expensive

Predicting a five-star rating is a regression if you care how far off you are, and a classification if you only care about the exact star. Those two framings need different models, different metrics and sometimes different data. Teams routinely build one and evaluate it as though it were the other.

Day 6 When Not to Use It

The questions to ask before any code exists

Day 1 said that reaching for machine learning where rules would do is a common and expensive mistake. This day is the practical version of that claim.

When rules are the better answer

- The logic is known and stable. Tax rates, opening hours, eligibility thresholds. Learning them from examples is slower, less accurate and impossible to audit.
- You have no labelled examples, and getting them would cost more than writing the logic.
- Every decision must be explainable to a regulator in terms of the rule that produced it.
- The cost of a wrong answer is severe and the volume is low enough for a person to review.

When learning is the better answer

- The pattern is real but nobody can articulate it. Recognising speech, reading handwriting, judging whether a photograph contains a pedestrian.
- The rules would be enormous. Ten thousand keyword rules for spam is not a system anybody can maintain.
- The pattern changes and you can keep collecting examples, so the system can be retrained rather than rewritten.
- You have, or can get, labelled examples in reasonable numbers.

The questions worth asking before anything is built

1. **What decision changes as a result?** If nothing is done differently, the model has no value however accurate it is.
2. **What does a wrong answer cost, and which kind of wrong?** Missing spam and blocking an invoice are not equally bad, and no single accuracy number distinguishes them.
3. **What would a simple rule achieve?** Build it. It is a day's work and it is the number your model has to beat.
4. **Do the labels exist?** If not, who makes them, how long does it take, and do they agree with each other?
5. **Will the data still look like this in a year?** A model is a snapshot of the past, and week 13 covers what happens when the world moves.

⚠ Watch Out

Not one of those questions is technical

That is the point. The most consequential mistakes in applied AI are made before any code exists, by choosing a problem that does not need this, or whose labels do not exist, or whose output nobody will act on. Every one of those is discoverable in a conversation, and the conversation is cheaper than the project.

The one line worth remembering from the week

Day 7 One Problem, End to End

The whole week applied in order, and the four counts that matter

Everything from the week, applied once, in the order it should be done in.

```
import random

random.seed(11)

# Ten phrasings to learn from and four the model never sees, per class.
SPAM_TRAIN = [
    'congratulations you have won a {prize}, claim it {when}',
    'your {thing} will be closed, verify it {when}',
    'earn {amount} a week from home, no experience needed',
    'exclusive offer, {pct} off everything {when}',
    'we noticed unusual activity on your {thing}, confirm it {when}',
    'final notice about your {thing}, respond {when}',
    'you have been selected for a {prize}, reply to arrange it',
    'limited deal, {pct} discount if you order {when}',
    'your {thing} is on hold, update your details {when}',
    'claim your {prize} before it expires {when}',
]
SPAM_TEST = [
    'act {when} to release the {amount} waiting on your {thing}',
    'we tried to deliver your {prize}, reply with your details',
    'a {pct} refund is pending, tell us where to send it',
    'security alert on your {thing}, reply {when} or lose access',
]
# Ordinary mail uses the same words. That is the point: 'account',
# 'offer' and 'urgent' are perfectly normal words at work.
HAM_TRAIN = [
    'the invoice for {month} is attached, let me know if it looks wrong',
    'can we move the {day} meeting, i have a clash',
    'the report is finished, it is in the shared folder',
    'thanks for the figures, they match the {thing} i have',
    'i am out on {day}, send anything urgent before then',
    'the training session is confirmed for {month}',
    'could you check the {thing} settings when you get a moment',
    'the client asked about dates, i said {month}',
    'we should discuss the {pct} increase before {day}',
    'their offer came in {when}, i think we should accept',
]
HAM_TEST = [
    'payroll needs your {thing} details before {month}',
    'the supplier quoted {amount}, which is over budget',
    'reminder that the {day} review starts at nine',
    'i have won the argument about the {month} deadline',
]
FILL = {
    'prize': ['free holiday', 'new phone', 'cash prize', 'gift card'],
    'when': ['today', 'within 24 hours', 'before friday', 'now'],
    'thing': ['account', 'password', 'payment', 'subscription'],
    'amount': ['500 pounds', '2000 pounds', '1000 pounds'],
    'pct': ['90%', '75%', '80%'],
    'month': ['january', 'march', 'october', 'june'],
    'day': ['monday', 'tuesday', 'thursday', 'friday'],
```

```
}
# Both classes draw from this, so plenty of words carry no signal.
NOISE = ['please', 'thanks', 'regards', 'best', 'hi', 'hello', 'team',
         'quick', 'note', 'fyi', 'cheers', 'morning']

def fill(template):
    out = template
    for key, options in FILL.items():
        while '{' + key + '}' in out:
            out = out.replace('{' + key + '}', random.choice(options), 1)
    # a couple of ordinary words, so the two classes overlap
    return ' '.join([random.choice(NOISE)] + out.split()
                    + [random.choice(NOISE)])

def build(spam_templates, ham_templates, n):
    xs, ys = [], []
    for _ in range(n):
        xs.append(fill(random.choice(spam_templates)))
        ys.append(1)
        xs.append(fill(random.choice(ham_templates)))
        ys.append(0)
    return xs, ys

train_x, train_y = build(SPAM_TRAIN, HAM_TRAIN, 120)
# held out mail is phrased in ways the training set never contained
test_x, test_y = build(SPAM_TEST, HAM_TEST, 60)
messages, labels = train_x + test_x, train_y + test_y
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.pipeline import make_pipeline
from sklearn.metrics import confusion_matrix

# 1. the do-nothing baseline, first, always
dummy = make_pipeline(TfidfVectorizer(),
                      DummyClassifier(strategy='most_frequent'))
dummy.fit(train_x, train_y)

# 2. the model
model = make_pipeline(TfidfVectorizer(), LogisticRegression())
model.fit(train_x, train_y)

print('%-24s %10s' % ('', 'held out'))
print('%-24s %10.4f' % ('do nothing', dummy.score(test_x, test_y)))
print('%-24s %10.4f' % ('the model', model.score(test_x, test_y)))
print()

# 3. not one number but the four kinds of outcome
tn, fp, fn, tp = confusion_matrix(test_y, model.predict(test_x)).ravel()
print('spam correctly caught      %3d' % tp)
print('spam that got through      %3d' % fn)
print('ordinary mail blocked      %3d' % fp)
print('ordinary mail delivered    %3d' % tn)
```

```
                          held out
do nothing                  0.5000
the model                   0.8833

spam correctly caught       59
spam that got through        1
ordinary mail blocked       13
ordinary mail delivered     47
```

The four counts at the bottom matter more than the accuracy above them. Blocking somebody's invoice and letting a junk message through are both errors and they are not equally bad, and no single percentage can tell you which the system is doing. Week 2 develops this properly.

What the week established

- AI covers rules, search and learning, and they are suited to different problems rather than being stages of sophistication.
- A learned model is examples, labels, features, a function and a loss being reduced. Nothing more mysterious than that.
- Every number must be measured on data the system has not seen.
- Every number needs a do-nothing baseline beside it before it means anything.
- The shape of the problem, and whether it needs learning at all, are decisions made before any code is written.

Where the course goes next

Week 2 goes deeper into how machines learn and how to measure them honestly. Week 3 covers the search and knowledge methods that predate learning and still run a great deal of production software. Weeks 4 to 7 are language, weeks 8 to 10 are images, and weeks 11 to 13 are about the consequences of deploying any of it, which is the part that turns a working model into something you can defend.

Self-Assessment

Try It Yourself

1. What is the AI effect?

   a) Models improving over time

   b) Work stops being called AI once it works reliably, so the word describes a frontier rather than a fixed set of methods

   c) Accuracy falling in production

   d) The tendency of AI to replace jobs

2. When is search a better choice than machine learning?

   a) Never, it is an older technique

   b) When you have very little data

   c) When the rules are known and the difficulty is the number of possibilities, and

you need a provably optimal answer

d) When the problem involves images

3. On held out phrasings the keyword rules scored 0.492. Why is that result not contrived?

   a) Because the corpus was too small

   b) Because the rules were badly chosen

   c) It is contrived, real rules do better

   d) Because whoever sends the spam can read the filter, so the phrasings that survive are the ones avoiding the keyword list

4. What are the five parts of a supervised learning system?

   a) Examples, labels, features, a model, and a loss being reduced

   b) Input, hidden layer, output, weights, bias

   c) Train, validate, test, deploy, monitor

   d) Data, code, server, model, output

5. A decision tree scored 1.000 on training data and 0.733 on held out data, while logistic regression scored 1.000 and 0.883. What does that show?

   a) The tree is better

   b) Both are equally good

   c) A perfect training score says nothing; the gap to held out data is what distinguishes learning from memorising

   d) The held out set was too small

6. Why must an accuracy figure have a baseline beside it?

   a) It does not need one if the accuracy is high

   b) To satisfy reviewers

   c) Because without one the number cannot be interpreted at all, and on imbalanced data a do-nothing model can score very highly

   d) Because baselines are faster

7. What is a bag of words representation?

   a) A dictionary of word meanings

   b) The vocabulary of the training set

   c) Words sorted alphabetically

   d) Which words a document contains and how often, discarding the order

8. Which three conditions together make machine learning the right tool?

   a) The pattern is real, nobody can write it down, and examples are obtainable

   b) The problem is important, urgent and well funded

   c) Clean data, known rules and a clear metric

   d) Large budget, big data, fast hardware

Answers:1b, 2c, 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

Measuring a Model Honestly

Week 2

OBJECTIVES

Why accuracy misleads, what to use instead, and the four ways a good result can turn out not to be one.

A model that predicts *not fraud* every single time is 96 percent accurate on this week's data, and that figure is the most common dishonest number in the field. This week replaces it with measurements that mean something, shows how information leaks across a split without anybody noticing, and demonstrates that a result chosen for being the best of six is optimistic by construction.

Day 1 The Model That Catches Nothing

Why accuracy is the wrong metric for anything rare

Week 1 ended with a warning: an accuracy figure with no baseline beside it means nothing. This week takes that apart properly, because on real data accuracy is not merely unhelpful, it is actively misleading.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
```

The number that looks like success

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
```

```

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression

lazy = DummyClassifier(strategy='most_frequent').fit(Xtr, ytr)
model = LogisticRegression(max_iter=2000).fit(Xtr, ytr)
print('%-34s %10s' % ('', 'accuracy'))
print('%-34s %10.4f' % ('predict "not fraud" every time',
                        lazy.score(Xte, yte)))
print('%-34s %10.4f' % ('an actual model', model.score(Xte, yte)))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
                                    accuracy
predict "not fraud" every time        0.9525
an actual model                       0.9650
```

A model that has never once said the word fraud is 96 percent accurate. Put that figure in a slide and it reads as a triumph. It catches nothing, it would save nobody any money, and it is a single line of code that ignores its input entirely.

Watch Out

This is the most common dishonest number in the field

It is rarely dishonest on purpose. Accuracy is the default metric in every library, it is the one non-specialists ask for, and on imbalanced data it is dominated entirely by the majority class. Any problem where the interesting event is rare, and most valuable problems are, has this property.

What to look at instead

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
```

```
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix

model = LogisticRegression(max_iter=2000).fit(Xtr, ytr)
tn, fp, fn, tp = confusion_matrix(yte, model.predict(Xte)).ravel()
print('%-28s %6d' % ('fraud caught', tp))
print('%-28s %6d' % ('fraud missed', fn))
print('%-28s %6d' % ('honest flagged as fraud', fp))
print('%-28s %6d' % ('honest left alone', tn))
print()
print('of the %d real fraud cases, it found %d' % (tp + fn, tp))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
fraud caught                    18
fraud missed                    39
honest flagged as fraud          3
honest left alone             1140

of the 57 real fraud cases, it found 18
```

Day 2 Precision, Recall and the Threshold

One model, five different systems, and who decides which

Two ratios of those four counts come up constantly, they are easy to confuse, and the difference between them is usually the whole conversation with whoever is paying for the system.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score, recall_score, f1_score

model = LogisticRegression(max_iter=2000).fit(Xtr, ytr)
pred = model.predict(Xte)
print('%-14s %8.4f' % ('accuracy', model.score(Xte, yte)))
print('%-14s %8.4f' % ('precision', precision_score(yte, pred)))
print('%-14s %8.4f' % ('recall', recall_score(yte, pred)))
print('%-14s %8.4f' % ('f1', f1_score(yte, pred)))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
accuracy         0.9650
```

```
precision        0.8571
recall           0.3158
f1               0.4615
```

You get to choose the trade

A classifier does not really output a class. It outputs a score, and somebody chose a threshold of 0.5 to turn that into a decision. That choice is yours, it is not a technical detail, and moving it moves precision and recall in opposite directions.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score, recall_score

model = LogisticRegression(max_iter=2000).fit(Xtr, ytr)
scores = model.predict_proba(Xte)[:, 1]
print('%10s %10s %10s %12s' % ('threshold', 'precision', 'recall',
                               'flagged'))
for t in [0.05, 0.10, 0.25, 0.50, 0.75]:
    pred = (scores >= t).astype(int)
    if pred.sum() == 0:
        continue
    print('%10.2f %10.4f %10.4f %12d'
          % (t, precision_score(yte, pred), recall_score(yte, pred),
             pred.sum()))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
 threshold  precision     recall      flagged
      0.05     0.1674     0.7018          239
      0.10     0.3274     0.6491          113
      0.25     0.5750     0.4035           40
      0.50     0.8571     0.3158           21
      0.75     1.0000     0.1228            7
```

One model, five different systems. At a low threshold it catches far more fraud and sends many more honest customers to be investigated. At a high threshold it is almost always right when it speaks and it stays quiet through most of the fraud.

Neither column is the right answer on its own

The right threshold depends on what each error costs. If investigating a flagged case costs five pounds and a missed fraud costs five hundred, the arithmetic is not close and you want the low threshold. If flagging a customer means freezing their account, the calculation changes completely.

That is a business decision informed by a table like this one. It is not a hyperparameter, and it should not be left at whatever the library defaulted to.

The metric that does not need a threshold

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.metrics import roc_auc_score, average_precision_score

model = LogisticRegression(max_iter=2000).fit(Xtr, ytr)
scores = model.predict_proba(Xte)[:, 1]
coin = DummyClassifier(strategy='stratified',
                       random_state=0).fit(Xtr, ytr)
print('%-34s %10s %12s' % ('', 'roc auc', 'avg precision'))
print('%-34s %10.4f %12.4f'
      % ('the model', roc_auc_score(yte, scores),
         average_precision_score(yte, scores)))
print('%-34s %10.4f %12.4f'
      % ('random guessing',
         roc_auc_score(yte, coin.predict_proba(Xte)[:, 1]),
         yte.mean()))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data

                                    roc auc avg precision
the model                            0.8281       0.5114
random guessing                      0.4891       0.0475
```

ROC AUC asks whether the model ranks a real fraud above an honest case more often than not, across every possible threshold. Random guessing scores 0.5 by construction, which makes it easy to read. Average precision does something similar but weights the top of the ranking, and its random baseline is the fraud rate itself, which is why the second column's two numbers are so far apart.

Split luck, cross-validation, and reporting the spread

A single held out set gives you one number, measured once, on one particular slice of the data. Whether you can trust it depends on how big that slice is and how lucky the split was.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

print('%8s %12s' % ('split seed', 'roc auc'))
aucs = []
for seed in range(6):
    a, b, ya, yb = train_test_split(X, y, test_size=0.3, stratify=y,
                                    random_state=seed)
    m = LogisticRegression(max_iter=2000).fit(a, ya)
    auc = roc_auc_score(yb, m.predict_proba(b)[:, 1])
    aucs.append(auc)
    print('%8d %12.4f' % (seed, auc))
print()
print('spread across splits %.4f' % (max(aucs) - min(aucs)))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
split seed      roc auc
       0       0.8281
       1       0.8270
       2       0.8429
       3       0.8613
       4       0.8357
       5       0.8045

spread across splits 0.0569
```

Same data, same model, same code. Only which rows landed in which half differs, and the answer moves. That spread is the resolution of your measurement: any improvement smaller than it has not been demonstrated.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
```

```
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
print('%-24s %10s %10s' % ('', 'mean auc', 'spread'))
for name, m in [('logistic regression',
                 LogisticRegression(max_iter=2000)),
                ('random forest',
                 RandomForestClassifier(n_estimators=120,
                                        random_state=0))]:
    s = cross_val_score(m, X, y, cv=cv, scoring='roc_auc')
    print('%-24s %10.4f %10.4f' % (name, s.mean(), s.max() - s.min()))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
                         mean auc     spread
logistic regression        0.8322     0.0951
random forest              0.8585     0.0711
```

Watch Out

Stratify, or the folds will not be comparable

With fraud at four percent, an ordinary five-way split can easily leave one fold with noticeably fewer fraud cases than another, and then the folds are measuring different problems. `StratifiedKFold` keeps the class balance the same in every fold. On imbalanced data it is not an optimisation, it is the difference between a meaningful average and a noisy one.

Day 4 Leakage

How to get an excellent score on a model that cannot work

There is a way to get an excellent held out score on a model that will fail completely in production, and it is common enough to have a name.

The most common version, demonstrated

Scaling the data before splitting it. It looks harmless: the scaler only computes a mean and a standard deviation. But it computes them over the test rows too, so information about the held out set is baked into the training data.

```
import numpy as np
from sklearn.datasets import make_classification
```

```
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)

# wrong: the scaler sees every row, including the fold being tested on
leaked = StandardScaler().fit_transform(X)
bad = cross_val_score(LogisticRegression(max_iter=2000), leaked, y,
                      cv=cv, scoring='roc_auc').mean()

# right: the scaler is part of the pipeline, refitted inside each fold
good = cross_val_score(make_pipeline(StandardScaler(),
                                     LogisticRegression(max_iter=2000)),
                       X, y, cv=cv, scoring='roc_auc').mean()
print('%-38s %10.4f' % ('scaled before splitting', bad))
print('%-38s %10.4f' % ('scaled inside the pipeline', good))
print('%-38s %10.4f' % ('difference', bad - good))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
scaled before splitting                  0.8326
scaled inside the pipeline               0.8325
difference                               0.0001
```

On this data the gap is small, and that is exactly why leakage is dangerous rather than reassuring. A small unearned advantage does not look like a bug. It looks like your model being slightly better than the alternative, which is the conclusion you were hoping for.

The versions that are not small

- **A feature computed from the future.** Predicting whether a customer will churn, using their total spend, which includes spending after they churned.
- **An identifier.** A customer number that happens to correlate with the outcome because of how the data was collected.
- **Duplicate rows across the split.** The same case in both halves, so the model is tested on something it memorised.
- **Choosing features by looking at all the data.** Picking the twenty columns most correlated with the target, over the whole set, before splitting.

The rule that prevents all of them

Every step that learns anything from the data, and that includes scalers, encoders, imputers and feature selectors, belongs inside the pipeline so it is refitted on each training fold. If you find yourself calling `fit` on anything before `train_test_split`, stop.

Day 5 Working With Imbalance

Class weights measured, and what they actually change

Imbalanced data does not only break the metric. It also affects training, because a loss averaged over every row is dominated by the majority class. There are standard remedies and they are worth measuring rather than assuming.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (roc_auc_score, average_precision_score,
                             recall_score, precision_score)

def report(name, model):
    model.fit(Xtr, ytr)
    s = model.predict_proba(Xte)[:, 1]
    pred = model.predict(Xte)
    print('%-32s %8.4f %10.4f %10.4f %8.4f'
          % (name, roc_auc_score(yte, s),
             average_precision_score(yte, s),
             precision_score(yte, pred, zero_division=0),
             recall_score(yte, pred)))

print('%-32s %8s %10s %10s %8s'
      % ('', 'auc', 'avg prec', 'precision', 'recall'))
report('logistic regression',
       LogisticRegression(max_iter=2000))
report('  with balanced class weights',
       LogisticRegression(max_iter=2000, class_weight='balanced'))
report('random forest',
       RandomForestClassifier(n_estimators=200, random_state=0))
report('  with balanced class weights',
       RandomForestClassifier(n_estimators=200, random_state=0,
                              class_weight='balanced'))
```

```
2800 training rows, 1200 held out
```

```
fraud is 4.7% of the data
                                    auc   avg prec  precision   recall
logistic regression              0.8281     0.5114     0.8571   0.3158
  with balanced class weights    0.8223     0.5057     0.1639   0.6842
random forest                    0.8736     0.5400     0.8889   0.2807
  with balanced class weights    0.8724     0.5606     0.8462   0.1930
```

Read the ranking columns and the decision columns separately, because they answer different questions. Class weighting mostly does not change how well the model *ranks* cases, which is what AUC and average precision measure. What it changes is where the default threshold falls, which moves precision and recall a long way.

Watch Out

Class weighting is often just threshold moving in disguise

That is worth knowing because it is usually presented as a way to make the model better at finding the rare class. If the ranking metric barely moves, the model has not got better at the problem, it has become more willing to say yes, and you could have achieved the same thing by choosing a threshold from yesterday's table.
The advantage of doing it explicitly with a threshold is that you can see the trade you are making and pick the point you actually want.

Day 6 Two More Ways to Fool Yourself

Tuning on the test set, and not knowing the human ceiling

Two more things that quietly invalidate a measurement, both of which look like good practice while you are doing them.

Tuning on the test set

You try eight models, pick the one with the best held out score, and report that score. The number is now optimistic, because you used the held out set to make a choice, which makes it a training set for that choice.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

```

```
# a proper three way split: train, choose on validation, report on test
Xa, Xrest, ya, yrest = train_test_split(X, y, test_size=0.4,
                                        stratify=y, random_state=0)
Xv, Xt, yv, yt = train_test_split(Xrest, yrest, test_size=0.5,
                                  stratify=yrest, random_state=0)

best, best_c = -1, None
for c in [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]:
    m = LogisticRegression(C=c, max_iter=2000).fit(Xa, ya)
    v = roc_auc_score(yv, m.predict_proba(Xv)[:, 1])
    if v > best:
        best, best_c = v, c

final = LogisticRegression(C=best_c, max_iter=2000).fit(Xa, ya)
test = roc_auc_score(yt, final.predict_proba(Xt)[:, 1])
print('chose C=%g on validation, scoring %.4f' % (best_c, best))
print('the same model on the untouched test set  %.4f' % test)
print('the validation figure was optimistic by   %.4f' % (best - test))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
chose C=10 on validation, scoring 0.8285
the same model on the untouched test set  0.8152
the validation figure was optimistic by   0.0133
```

The validation score is the one you used to choose, so it is the highest of six numbers and is biased upward by that selection alone. The test score is the honest one. With six candidates the gap is modest; with two hundred it is not.

Not knowing what a person would score

A model at 0.85 sounds mediocre until you learn that two trained annotators only agree with each other 0.87 of the time. Above that ceiling the remaining errors are largely disagreement about what the right answer even is, and no model will fix them.

- Have two people label the same two hundred cases and measure how often they agree. That is your ceiling.
- If they agree rarely, the task definition is the problem, not the model. Fix the definition first.
- Report the ceiling next to your score. A model at 0.85 against a human ceiling of 0.87 is a very different result from 0.85 against 0.99.

Day 7 The Honest Procedure

Everything assembled, and the checklist

The whole week as one procedure, in the order that keeps it honest.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# A fraud-shaped problem: one case in twenty five is real, which is
# roughly what payment fraud looks like and nothing like a textbook set.
```

```
X, y = make_classification(n_samples=4000, n_features=12, n_informative=5,
                           n_redundant=2, weights=[0.96, 0.04],
                           flip_y=0.02, class_sep=0.9, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3,
                                      stratify=y, random_state=0)
print('%d training rows, %d held out' % (len(Xtr), len(Xte)))
print('fraud is %.1f%% of the data' % (100 * y.mean()))
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.metrics import (confusion_matrix, roc_auc_score,
                             precision_score, recall_score)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))

# 1. baseline, 2. model, both cross validated, both on a ranking metric
base = cross_val_score(DummyClassifier(strategy='stratified',
                                       random_state=0),
                       X, y, cv=cv, scoring='roc_auc')
got = cross_val_score(pipe, X, y, cv=cv, scoring='roc_auc')
print('%-22s %10s %10s' % ('', 'mean auc', 'spread'))
print('%-22s %10.4f %10.4f' % ('baseline', base.mean(),
                               base.max() - base.min()))
print('%-22s %10.4f %10.4f' % ('model', got.mean(),
                               got.max() - got.min()))
print()

# 3. choose a threshold deliberately, on validation not on test
pipe.fit(Xtr, ytr)
s = pipe.predict_proba(Xte)[:, 1]
for t in [0.10, 0.50]:
    pred = (s >= t).astype(int)
    tn, fp, fn, tp = confusion_matrix(yte, pred).ravel()
    print('threshold %.2f  caught %3d  missed %3d  false alarms %3d'
          % (t, tp, fn, fp))
```

```
2800 training rows, 1200 held out
fraud is 4.7% of the data
                         mean auc     spread
baseline                   0.4939     0.0553
model                      0.8325     0.0950

threshold 0.10  caught  37  missed  20  false alarms  77
threshold 0.50  caught  17  missed  40  false alarms   3
```

The checklist

1. State the class balance before anything else. It decides which metrics are meaningful.

2. Run a do-nothing baseline and keep it beside every later number.

3. Put every step that learns from data inside a pipeline.
4. Cross-validate, and report the spread as well as the average.
5. Choose the metric from the decision being made, not from the library default. On imbalanced problems that is rarely accuracy.
6. Choose the threshold deliberately, from the cost of each kind of error.
7. Keep a test set that has influenced no decision at all, and look at it once.
8. Find out what a person scores on the same task.

What this week is really teaching

Self-Assessment

Try It Yourself

1. A model predicts *not fraud* every time on data where fraud is 4 percent. What is its accuracy?

    a) About 50 percent
    b) About 96 percent
    c) Undefined
    d) About 4 percent

2. What does recall measure?

    a) Of the cases flagged, how many were real
    b) Of the real cases, how many were flagged
    c) How often the model is right overall
    d) How confident the model is

3. Moving the decision threshold does what?

    a) Changes the ROC AUC
    b) Retrains the model
    c) Trades precision against recall, turning one model into several different systems
    d) Improves both precision and recall

4. Why is ROC AUC useful for comparing models?

    a) It works only on balanced data
    b) It measures calibration
    c) It is always higher than accuracy
    d) It needs no threshold and random guessing scores 0.5 by construction

5. You scale the whole dataset before splitting it. What have you done?

    a) Leaked information about the held out rows into training, producing an unearned and usually small advantage

b) Made training slower

c) Broken the scaler

d) Nothing wrong

6. Class weighting barely changed AUC but moved precision and recall a long way. What does that suggest?

   a) The model got much better at the rare class

   b) It mostly moved where the default threshold falls rather than improving the ranking

   c) The metric is broken

   d) The data needs resampling

7. You pick the best of six models by held out score and report it. What is wrong?

   a) You should have used accuracy

   b) Nothing

   c) That score is the highest of six and is biased upward by the selection, so a set the choice never touched is needed

   d) Six is too few models

8. Why measure how often two human annotators agree?

   a) It is required by law

   b) To estimate labelling cost only

   c) To decide who to employ

   d) It establishes a ceiling: above it the remaining errors are largely disagreement about the right answer

Answers:1b, 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 3

Search, Knowledge and Reasoning

Week 3

OBJECTIVES

The half of AI that derives answers instead of learning them, and why it is still the right tool for a great deal of real work.

Before machine learning, artificial intelligence meant searching through possibilities and reasoning over facts, and a great deal of software still works that way because for some problems it is simply better. This week builds a maze solver, a timetabler that can prove no timetable exists, and a knowledge base that contradicts itself over a penguin. It ends with the arithmetic showing that a test which is 99 percent accurate in both directions can still mean you are probably fine.

Day 1 Answers You Derive Rather Than Learn

States, goals, and the one line that separates two search strategies

Week 2 was about learning from examples. This week is about the other half of artificial intelligence, the half that came first and is still running a great deal of software: getting an answer by searching through possibilities rather than by learning from data.

It matters for two reasons. Plenty of real problems have this shape and are solved badly with machine learning. And the vocabulary of this tradition, states, goals, heuristics and constraints, is how a lot of AI work is still described.

The problem shape

```
GRID = [
    '...#........',
    '...#........',
    '.#......#..#',
    '....#..#..#.',
    '#...#...#...',
    '..........##',
    '...#........',
    '.#..........',
    '.#....#.##..',
    '.##.#....#..',
    '....##.###..',
    '#.###..#.##.',
]
ROWS, COLS = len(GRID), len(GRID[0])
START, GOAL = (0, 0), (ROWS - 1, COLS - 1)
```

```

def neighbours(cell):
    r, c = cell
    for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        nr, nc = r + dr, c + dc
        if 0 <= nr < ROWS and 0 <= nc < COLS and GRID[nr][nc] != '#':
            yield (nr, nc)

def show(path):
    marked = set(path or [])
    for r in range(ROWS):
        print(''.join('o' if (r, c) in marked else GRID[r][c]
                      for c in range(COLS)))
show([])
print()
print('start %s, goal %s' % (START, GOAL))
```

```
...#........
...#........
.#......#..#
....#..#..#.
#...#...#...
..........##
...#........
.#..........
.#....#.##..
.##.#....#..
....##.###..
#.###..#.##.

start (0, 0), goal (11, 11)
```

Breadth first: correct, and wasteful

Explore everything one step away, then everything two steps away, and so on. The first time you reach the goal you are guaranteed to have done it in the fewest possible moves, because you have already ruled out every shorter route.

```
GRID = [
    '...#........',
    '...#........',
    '.#......#..#',
    '....#..#..#.',
    '#...#...#...',
    '..........##',
    '...#........',
    '.#..........',
    '.#....#.##..',
    '.##.#....#..',
    '....##.###..',
    '#.###..#.##.',
]
ROWS, COLS = len(GRID), len(GRID[0])
START, GOAL = (0, 0), (ROWS - 1, COLS - 1)
```

```

def neighbours(cell):
    r, c = cell
    for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        nr, nc = r + dr, c + dc
        if 0 <= nr < ROWS and 0 <= nc < COLS and GRID[nr][nc] != '#':
            yield (nr, nc)

def show(path):
    marked = set(path or [])
    for r in range(ROWS):
        print(''.join('o' if (r, c) in marked else GRID[r][c]
                      for c in range(COLS)))
from collections import deque

def breadth_first(start, goal):
    queue = deque([[start]])
    seen = {start}
    expanded = 0
    while queue:
        path = queue.popleft()
        expanded += 1
        if path[-1] == goal:
            return path, expanded
        for nxt in neighbours(path[-1]):
            if nxt not in seen:
                seen.add(nxt)
                queue.append(path + [nxt])
    return None, expanded

path, expanded = breadth_first(START, GOAL)
show(path)
print()
print('%d steps, %d states examined' % (len(path) - 1, expanded))
```

```
o..#........
o..#........
o#......#..#
oo..#..#..#.
#o..#...#...
.o........##
.oo#........
.#oooooooo.
.#....#.##o.
.##.#....#o.
....##.###oo
#.###..#.##o

22 steps, 108 states examined
```

Depth first: cheap, and not shortest

```
GRID = [
```

```
    '...#........',
    '...#........',
    '.#......#..#',
    '....#..#..#.',
    '#...#...#...',
    '..........##',
    '...#........',
    '.#..........',
    '.#....#.##..',
    '.##.#....#..',
    '....##.###..',
    '#.###..#.##.',
]
ROWS, COLS = len(GRID), len(GRID[0])
START, GOAL = (0, 0), (ROWS - 1, COLS - 1)

def neighbours(cell):
    r, c = cell
    for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        nr, nc = r + dr, c + dc
        if 0 <= nr < ROWS and 0 <= nc < COLS and GRID[nr][nc] != '#':
            yield (nr, nc)

def show(path):
    marked = set(path or [])
    for r in range(ROWS):
        print(''.join('o' if (r, c) in marked else GRID[r][c]
                      for c in range(COLS)))
def depth_first(start, goal):
    stack = [[start]]
    seen = {start}
    expanded = 0
    while stack:
        path = stack.pop()
        expanded += 1
        if path[-1] == goal:
            return path, expanded
        for nxt in neighbours(path[-1]):
            if nxt not in seen:
                seen.add(nxt)
                stack.append(path + [nxt])
    return None, expanded

path, expanded = depth_first(START, GOAL)
print('%d steps, %d states examined' % (len(path) - 1, expanded))
print('breadth first found a shorter route, having looked at more')
```

```
48 steps, 64 states examined
breadth first found a shorter route, having looked at more
```

One line differs between those two functions: whether you take work from the front of the queue or the back. That single choice decides whether the answer is guaranteed shortest and how much memory the search needs, which is a good illustration of how much structure sits in very small

decisions here.

Day 2 Giving the Search a Hint

Heuristics, A, and reducing work without changing the answer*

Breadth first searches in every direction equally, including directly away from the goal. Since we know where the goal is, that is obviously wasteful, and the fix is to give the search a hint.

```
GRID = [
    '...#........',
    '...#........',
    '.#......#..#',
    '....#..#..#.',
    '#...#...#...',
    '..........##',
    '...#........',
    '.#..........',
    '.#....#.##..',
    '.##.#....#..',
    '....##.###..',
    '#.###..#.##.',
]
ROWS, COLS = len(GRID), len(GRID[0])
START, GOAL = (0, 0), (ROWS - 1, COLS - 1)

def neighbours(cell):
    r, c = cell
    for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        nr, nc = r + dr, c + dc
        if 0 <= nr < ROWS and 0 <= nc < COLS and GRID[nr][nc] != '#':
            yield (nr, nc)

def show(path):
    marked = set(path or [])
    for r in range(ROWS):
        print(''.join('o' if (r, c) in marked else GRID[r][c]
                      for c in range(COLS)))
import heapq

def manhattan(cell, goal):
    """Straight line distance ignoring walls. It can never be more
    than the true distance, which is what makes A* safe to trust."""
    return abs(cell[0] - goal[0]) + abs(cell[1] - goal[1])

def a_star(start, goal):
    queue = [(manhattan(start, goal), 0, [start])]
    best = {start: 0}
    expanded = 0
    while queue:
        _, cost, path = heapq.heappop(queue)
        expanded += 1
        if path[-1] == goal:
            return path, expanded
        for nxt in neighbours(path[-1]):
```

```
            if cost + 1 < best.get(nxt, 10 ** 9):
                best[nxt] = cost + 1
                heapq.heappush(queue, (cost + 1 + manhattan(nxt, goal),
                                       cost + 1, path + [nxt]))
    return None, expanded

path, expanded = a_star(START, GOAL)
show(path)
print()
print('%d steps, %d states examined' % (len(path) - 1, expanded))
```

```
ooo#........
..o#........
.#ooooo.#..#
....#.o#..#.
#...#.oo#...
.......ooo##
...#.....ooo
.#.........o
.#....#.##.o
.##.#....#.o
....##.###.o
#.###..#.##o

22 steps, 74 states examined
```

```
GRID = [
    '...#........',
    '...#........',
    '.#......#..#',
    '....#..#..#.',
    '#...#...#...',
    '..........##',
    '...#........',
    '.#..........',
    '.#....#.##..',
    '.##.#....#..',
    '....##.###..',
    '#.###..#.##.',
]
ROWS, COLS = len(GRID), len(GRID[0])
START, GOAL = (0, 0), (ROWS - 1, COLS - 1)

def neighbours(cell):
    r, c = cell
    for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        nr, nc = r + dr, c + dc
        if 0 <= nr < ROWS and 0 <= nc < COLS and GRID[nr][nc] != '#':
            yield (nr, nc)

def show(path):
    marked = set(path or [])
    for r in range(ROWS):
        print(''.join('o' if (r, c) in marked else GRID[r][c]
```

```
                      for c in range(COLS)))
from collections import deque
import heapq

def breadth_first(start, goal):
    queue, seen, expanded = deque([[start]]), {start}, 0
    while queue:
        path = queue.popleft()
        expanded += 1
        if path[-1] == goal:
            return path, expanded
        for nxt in neighbours(path[-1]):
            if nxt not in seen:
                seen.add(nxt)
                queue.append(path + [nxt])
    return None, expanded

def manhattan(cell, goal):
    return abs(cell[0] - goal[0]) + abs(cell[1] - goal[1])

def a_star(start, goal):
    queue, best, expanded = [(manhattan(start, goal), 0, [start])], {start: 0}, 0
    while queue:
        _, cost, path = heapq.heappop(queue)
        expanded += 1
        if path[-1] == goal:
            return path, expanded
        for nxt in neighbours(path[-1]):
            if cost + 1 < best.get(nxt, 10 ** 9):
                best[nxt] = cost + 1
                heapq.heappush(queue, (cost + 1 + manhattan(nxt, goal),
                                       cost + 1, path + [nxt]))
    return None, expanded

print('%-16s %8s %10s' % ('', 'steps', 'examined'))
for name, fn in [('breadth first', breadth_first), ('a star', a_star)]:
    path, expanded = fn(START, GOAL)
    print('%-16s %8d %10d' % (name, len(path) - 1, expanded))
```

```
                  steps   examined
breadth first        22        108
a star               22         74
```

The same shortest route, found while looking at fewer states. That is the entire contribution of a heuristic: it does not improve the answer, it reduces the work needed to prove the answer. On a ten by ten grid this is a curiosity. On a road network it is the difference between a route planner and a machine that never replies.

This is the tradition modern AI grew out of

A* is from 1968. Route planning, puzzle solving, task scheduling and game playing all still use it or its descendants, and a satellite navigation system is doing this rather than anything learned. Search did not get replaced by machine learning. It continued to be the right tool for problems

where the rules are known and the difficulty is the size of the space.

Day 3 Assignments That Break No Rules

Constraint problems, backtracking, and proving there is no solution

A second classical shape, and one you will recognise from timetabling, rostering, seating plans and sudoku: you are not looking for a path, you are looking for an assignment that breaks no rules.

```
# Four talks, three rooms, some of which cannot share a slot because
# the same speaker or the same audience is involved.
TALKS = ['ethics', 'vision', 'language', 'safety']
SLOTS = ['morning', 'afternoon', 'evening']
CLASH = [('ethics', 'safety'), ('vision', 'language'),
         ('ethics', 'language')]

def ok(assignment):
    for a, b in CLASH:
        if a in assignment and b in assignment:
            if assignment[a] == assignment[b]:
                return False
    return True

def solve(assignment, tried):
    """Backtracking: assign one variable, check, and undo if stuck."""
    if len(assignment) == len(TALKS):
        return assignment, tried
    talk = TALKS[len(assignment)]
    for slot in SLOTS:
        assignment[talk] = slot
        tried += 1
        if ok(assignment):
            result, tried = solve(assignment, tried)
            if result:
                return result, tried
        del assignment[talk]
    return None, tried

answer, tried = solve({}, 0)
for talk in TALKS:
    print('%-10s %s' % (talk, answer[talk]))
print()
print('found after %d assignments, out of %d possible timetables'
      % (tried, len(SLOTS) ** len(TALKS)))
```

```
ethics     morning
vision     morning
language   afternoon
safety     afternoon

found after 6 assignments, out of 81 possible timetables
```

The important number is the last line. There are eighty-one possible timetables and the solver examined far fewer, because checking constraints as it went let it abandon whole branches the

moment they became impossible. That is the core idea, and it scales to problems with billions of combinations.

When there is no solution

```
TALKS = ['a', 'b', 'c']
SLOTS = ['morning', 'afternoon']
CLASH = [('a', 'b'), ('b', 'c'), ('a', 'c')]

def ok(assignment):
    return all(not (a in assignment and b in assignment
                    and assignment[a] == assignment[b])
               for a, b in CLASH)

def solve(assignment):
    if len(assignment) == len(TALKS):
        return assignment
    talk = TALKS[len(assignment)]
    for slot in SLOTS:
        assignment[talk] = slot
        if ok(assignment):
            result = solve(assignment)
            if result:
                return result
        del assignment[talk]
    return None

print('three talks that all clash, two slots: %s' % solve({}))
print('the solver proves there is no timetable, rather than guessing')
```

```
three talks that all clash, two slots: None
the solver proves there is no timetable, rather than guessing
```

Something no learned model can do

The solver returned `None`, and that answer is certain. It did not fail to find a timetable, it established that none exists.

A neural network asked the same question would return a confident-looking answer, because returning something is all it can do. When a problem requires a guarantee, and scheduling, routing and verification often do, this tradition is not a fallback. It is the only option.

Day 4 Writing Down What You Know

Knowledge bases, inheritance, and the penguin that broke them

The third classical strand tried to capture knowledge itself: facts and rules about the world, in a form a program could reason over. This was the dominant idea of AI for roughly thirty years and it is worth understanding both for what it does well and for how it failed.

```
FACTS = {
    ('sparrow', 'is a', 'bird'), ('bird', 'is a', 'animal'),
    ('penguin', 'is a', 'bird'), ('animal', 'needs', 'food'),
    ('bird', 'can', 'fly'), ('penguin', 'cannot', 'fly'),
```

```
}

def ask(subject, relation, depth=0):
    """Follow 'is a' links upward, inheriting properties."""
    answers = []
    for s, r, o in FACTS:
        if s == subject and r == relation:
            answers.append((o, depth))
    for s, r, o in FACTS:
        if s == subject and r == 'is a':
            answers.extend(ask(o, relation, depth + 1))
    return answers

for subject in ['sparrow', 'penguin']:
    print('%s can: %s' % (subject,
                          [a for a, _ in ask(subject, 'can')]))
    print('%s needs: %s' % (subject,
                            [a for a, _ in ask(subject, 'needs')]))
```

```
sparrow can: ['fly']
sparrow needs: ['food']
penguin can: ['fly']
penguin needs: ['food']
```

Nobody wrote down that a sparrow needs food. The system derived it from two facts and one rule about inheritance, and it can explain exactly how. That combination, deriving new facts and being able to justify them, is what made this approach so attractive.

And here is where it broke

```
FACTS = {
    ('penguin', 'is a', 'bird'), ('bird', 'can', 'fly'),
    ('penguin', 'cannot', 'fly'),
}

def inherited(subject, relation):
    out = []
    for s, r, o in FACTS:
        if s == subject and r == relation:
            out.append(o)
    for s, r, o in FACTS:
        if s == subject and r == 'is a':
            out.extend(inherited(o, relation))
    return out

print('can a penguin fly?')
print('   inherited from bird:', inherited('penguin', 'can'))
print('   stated directly:    ', inherited('penguin', 'cannot'))
print()
print('the knowledge base now says both, and nothing in it decides')
```

```
can a penguin fly?
   inherited from bird: ['fly']
```

```
   stated directly:     ['fly']

the knowledge base now says both, and nothing in it decides
```

Birds fly. Penguins are birds. Penguins do not fly. Every one of those is true, and together they are inconsistent unless the system knows that specific facts override inherited ones, and by how much, and what to do when two specific facts disagree.

Watch Out

Why symbolic AI stalled

Each exception is easy to patch. The difficulty is that there are an unbounded number of them, most are never written down because they are too obvious to state, and every patch can interact with every other. Systems built this way became too large for anybody to maintain and too brittle to trust outside the cases they were built for.
This is not a story about a bad idea. It is a story about a good idea meeting the fact that most human knowledge is tacit, and it is precisely the gap that learning from examples fills, because examples contain the exceptions without anyone having to enumerate them.

Day 5 Reasoning Under Uncertainty

The 99 percent accurate test that means almost nothing

Both traditions have a way of handling uncertainty. The symbolic one attaches probabilities to facts, and the arithmetic that results is worth doing once by hand, because the answer is deeply counter-intuitive and it appears constantly in medicine, screening and fraud.

```python
# A test for a condition that affects 1 person in 1000.
prevalence = 0.001
sensitivity = 0.99   # finds 99% of real cases
specificity = 0.99   # correctly clears 99% of healthy people

population = 1_000_000
ill = population * prevalence
well = population - ill
true_positive = ill * sensitivity
false_positive = well * (1 - specificity)

print('in a million people:')
print('  %8.0f have the condition' % ill)
print('  %8.0f of them test positive' % true_positive)
print('  %8.0f healthy people also test positive' % false_positive)
print()
print('so a positive test means a %.1f%% chance of being ill'
      % (100 * true_positive / (true_positive + false_positive)))
```

```
in a million people:
      1000 have the condition
       990 of them test positive
      9990 healthy people also test positive

so a positive test means a 9.0% chance of being ill
```

A test that is 99 percent accurate in both directions, and a positive result still means you are probably fine. There is no error in the arithmetic. There are simply far more healthy people, so the one percent of them that the test gets wrong outnumbers the ill people it gets right.

Where this bites in practice

Any model looking for something rare produces mostly false positives at the point of use, however good its metrics look. A screening tool, a fraud flag, a security alert: if the thing is rare enough, most of what the system reports will be wrong, and the system can still be extremely useful.

What it cannot be is automatic. It is a filter that makes human review affordable, and describing it as anything more than that sets up a failure.

Day 6 Two Traditions, One System

What each is genuinely good at, and how real systems combine them

Neither tradition won. The interesting systems now use both, and knowing which part of a problem belongs to which is a genuinely useful skill.

	Search and rules	**Learning from examples**
Needs	Someone who knows the rules	Labelled examples
Handles novelty	Only within the rules it was given	Interpolates, and fails unpredictably outside its data
Explains itself	Completely	Only with extra work
Can prove there is no answer	Yes	No
Handles messy input	Badly	Well
Fails	Visibly, by finding nothing	Quietly, with a confident wrong answer

How they combine in real systems

- **A route planner** learns how long each road takes from traffic data, then searches over the road network with those learned costs. Learning supplies the numbers, search supplies the guarantee.
- **A game engine** learns a rough evaluation of positions, then searches ahead using it. This is how the strongest chess and go programs work, and neither half would be enough alone.
- **A language model with tools** generates text and calls a calculator, a database or a solver for the parts that need an exact answer. This is the same division of labour with new names.
- **A fraud system** scores transactions with a model and applies hard rules for the cases where the law or the policy is not negotiable.

The pattern worth extracting

In every one of those, learning is used where the pattern is real but nobody can state it, and search or rules are used where an answer must be exact, explainable or guaranteed. That division is not a compromise between two schools. It is what each is actually good at.

Day 7 Learned Numbers, Derived Answers

A worked hybrid, and what it does and does not guarantee

One problem, solved with both halves of the week, which is the shape a real system usually takes.

```
import heapq

# Delivery times between stops, as learned from historical data.
# In a real system these numbers come from a regression model.
LEARNED_MINUTES = {
    ('depot', 'north'): 12, ('depot', 'east'): 9,
    ('north', 'ring'): 7, ('east', 'ring'): 15,
    ('east', 'south'): 8, ('ring', 'airport'): 11,
    ('south', 'airport'): 21, ('north', 'east'): 6,
}
STOPS = set()
for a, b in LEARNED_MINUTES:
    STOPS.update([a, b])

def cheapest(start, goal):
    """Search, over costs that were learned rather than measured."""
    queue = [(0, [start])]
    best = {start: 0}
    while queue:
        cost, path = heapq.heappop(queue)
        if path[-1] == goal:
            return cost, path
        for (a, b), minutes in LEARNED_MINUTES.items():
            if a == path[-1] and cost + minutes < best.get(b, 10 ** 9):
                best[b] = cost + minutes
                heapq.heappush(queue, (cost + minutes, path + [b]))
    return None, None

cost, path = cheapest('depot', 'airport')
print('%d minutes: %s' % (cost, ' → '.join(path)))
print()
print('the travel times were learned; the route was derived')
print('and is provably the cheapest given those times')
```

```
30 minutes: depot → north → ring → airport

the travel times were learned; the route was derived
and is provably the cheapest given those times
```

Note carefully what is guaranteed and what is not. The route is provably optimal *given the travel times*. If the learned times are wrong, the route is confidently wrong, and the search cannot tell. Combining the two traditions does not combine their guarantees, it chains them.

What the week established

- Search solves problems where the rules are known and the difficulty is the number of possibilities, and it can prove its answer optimal.
- A heuristic does not improve the answer, it reduces the work needed to reach it.
- Constraint problems can prove that no solution exists, which no learned model can do.

- Symbolic knowledge systems reason and explain beautifully, and failed on the unbounded number of exceptions nobody writes down.
- Base rates decide what a positive result means, and ignoring them is the most consequential statistical error in this field.
- Real systems use both traditions, with learning supplying the estimates and search supplying the guarantees.

Carrying this into the rest of the course

Self-Assessment

Try It Yourself

1. What single change turns breadth first search into depth first?

   a) The goal test
   b) The number of neighbours
   c) The heuristic used
   d) Whether work is taken from the front or the back of the store

2. What does an admissible heuristic guarantee?

   a) That the search still finds the shortest route, while examining fewer states
   b) That the answer is approximate
   c) That memory use is constant
   d) A faster answer always

3. What can a constraint solver do that no learned model can?

   a) Handle large inputs
   b) Prove that no solution exists
   c) Run quickly
   d) Work without a computer

4. Why did symbolic knowledge systems stall?

   a) They could not explain themselves
   b) They were too slow
   c) An unbounded number of exceptions, most never written down because they are too obvious to state
   d) Computers lacked memory

5. A test is 99 percent accurate both ways and the condition affects 1 in 1000. A positive result means roughly what chance of being ill?

   a) 50 percent
   b) 1 percent
   c) 99 percent

d) Around 9 percent

6. What is base rate neglect?

   a) Judging evidence without accounting for how common the thing was to begin with

   b) Using the wrong loss function

   c) Failing to normalise features

   d) Ignoring the learning rate

7. How do modern systems typically combine the two traditions?

   a) They do not, one replaced the other

   b) Learning supplies estimates and search supplies guarantees, as when a route planner searches over learned travel times

   c) Rules are used only for testing

   d) Search is used to train the model

8. In the worked hybrid, what was actually guaranteed?

   a) That travel times were accurate

   b) The route was correct in reality

   c) That the route is optimal *given* the learned times, so wrong times produce a confidently wrong route

   d) Nothing at all

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 4

Text as Data

Week 4

OBJECTIVES

How language becomes numbers, and why that step decides more than the choice of model.

Everything in language processing starts by turning text into numbers, and this week shows that the choices made there matter more than the model that follows. On the same set of reviews, deciding the topic is perfect however you represent the text, while deciding whether the review is positive is exactly chance with single words and perfect with pairs of them. Four different classifiers on the same features barely differ.

Day 1 Turning Sentences Into Tokens

Splitting, normalising, and the stop word that reverses your meaning

A computer cannot do arithmetic on a sentence. Everything in natural language processing begins with turning text into numbers, and the choices made in that step decide more about the final accuracy than the choice of model does.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
```

```
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
print('%d reviews, two labels each' % len(texts))
for t, s, c in list(zip(texts, sentiment, category))[:5]:
    print('%-9s %-8s %s' % (c, 'positive' if s else 'negative', t))
```

```
2400 reviews, two labels each
laptop    positive after a month the keyboard is still quite reliable, as described in the
    ↪ listing
laptop    negative for the price the charger is not sturdy, second one i have bought
kitchen   positive for the price the lid is hardly scratched, as described in the listing
kitchen   negative for the price the blade is not solid, the packaging was fine
clothing  positive the sleeve is very solid, as described in the listing
```

Step one: splitting into tokens

```
text = "The kettle's lid isn't sturdy -- I've returned it. Cost 24.99!"
print('naive split on spaces')
print(text.split())
print()

import re
print('splitting on word boundaries')
print(re.findall(r"[a-z0-9']+", text.lower()))
```

```
naive split on spaces
['The', "kettle's", 'lid', "isn't", 'sturdy', '--', "I've", 'returned', 'it.', 'Cost',
    ↪ '24.99!']

splitting on word boundaries
['the', "kettle's", 'lid', "isn't", 'sturdy', "i've", 'returned', 'it', 'cost', '24', '99']
```

The naive split leaves punctuation glued to words, so `it.` and `it` become different things. The second version lowercases and strips punctuation, which fixes that and quietly creates new problems: `isn't` survives as one token, and the price has become two.

Normalisation, and what it destroys

```
import re

STOP = {'the', 'is', 'a', 'and', 'it', 'to', 'of', 'i', 'was'}

def normalise(text, drop_stopwords):
    words = re.findall(r"[a-z']+", text.lower())
    if drop_stopwords:
        words = [w for w in words if w not in STOP]
    return words

for sentence in ['the pan is not sturdy', 'the pan is sturdy']:
    print('%-24s → %s' % (sentence, normalise(sentence, False)))
print()
for sentence in ['the pan is not sturdy', 'the pan is sturdy']:
    print('%-24s → %s' % (sentence, normalise(sentence, True)))
```

```
the pan is not sturdy    → ['the', 'pan', 'is', 'not', 'sturdy']
the pan is sturdy        → ['the', 'pan', 'is', 'sturdy']

the pan is not sturdy    → ['pan', 'not', 'sturdy']
the pan is sturdy        → ['pan', 'sturdy']
```

Watch Out

Removing stop words is not free

The usual advice is to drop common words because they carry no meaning. Look at what it did above: two sentences that mean opposite things became identical, because *not* is on almost every stop word list.
For topic classification this rarely matters. For anything involving negation, opinion or instruction it is fatal, and it is applied by default in a great deal of tutorial code.

Day 2 Counting Words

The document term matrix, sparsity, and TF-IDF

With tokens in hand, the standard representation counts them. Every document becomes a row of numbers, one column per word in the vocabulary.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
```

```
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
X = vec.fit_transform(train_x)
print('%d documents, %d columns' % X.shape)
print('%.4f%% of the matrix is non-zero'
      % (100 * X.nnz / (X.shape[0] * X.shape[1])))
print()
print('first 12 words of the vocabulary:')
print(list(vec.get_feature_names_out())[:12])
```

```
1680 documents, 65 columns
17.0302% of the matrix is non-zero

first 12 words of the vocabulary:
['after', 'and', 'arrived', 'as', 'battery', 'blade', 'bought', 'but', 'charger', 'collar',
    ↪ 'delivery', 'described']
```

Note the sparsity. Almost every entry is zero, because a short review contains a handful of the vocabulary's words. This is why text matrices are stored in a sparse format and why they can have tens of thousands of columns without exhausting memory.

Not every word deserves equal weight

```
import random

random.seed(3)
```

```

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

```

```
from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np

vec = TfidfVectorizer()
X = vec.fit_transform(train_x)
words = vec.get_feature_names_out()
idf = vec.idf_
order = np.argsort(idf)
print('%-16s %8s' % ('most common, so worth least', 'idf'))
for i in order[:5]:
    print('%-16s %8.3f' % (words[i], idf[i]))
print()
print('%-16s %8s' % ('rarest, so worth most', 'idf'))
for i in order[-5:]:
    print('%-16s %8.3f' % (words[i], idf[i]))
```

```
most common, so worth least      idf
the                 1.000
is                  1.227
was                 2.082
for                 2.484
price               2.484

rarest, so worth most      idf
stitching           3.754
blade               3.764
fabric              3.764
kettle              3.802
screen              3.842
```

Day 3 What Order Is Worth

The same text, two tasks, and a jump from chance to perfect

The representations so far throw away word order completely. Here is what that costs, on two tasks over exactly the same text.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
```

```
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

print('%-20s %12s %12s' % ('', 'sentiment', 'category'))
```

```
for name, make in [('unigrams', lambda: CountVectorizer()),
                   ('unigrams + bigrams',
                    lambda: CountVectorizer(ngram_range=(1, 2))),
                   ('tfidf unigrams', lambda: TfidfVectorizer()),
                   ('tfidf + bigrams',
                    lambda: TfidfVectorizer(ngram_range=(1, 2)))]:
    s = make_pipeline(make(),
                      LogisticRegression(max_iter=2000)).fit(train_x,
                                                             train_s)
    c = make_pipeline(make(),
                      LogisticRegression(max_iter=2000)).fit(train_x,
                                                             train_c)
    print('%-20s %12.4f %12.4f'
          % (name, s.score(test_x, test_s), c.score(test_x, test_c)))
```

```
                      sentiment     category
unigrams                 0.4931       1.0000
unigrams + bigrams       1.0000       1.0000
tfidf unigrams           0.4958       1.0000
tfidf + bigrams          1.0000       1.0000
```

The category column is 1.000 everywhere. Deciding whether a review is about a laptop or a pan needs only the nouns, and word order adds nothing at all.

The sentiment column is the interesting one. Unigrams score 0.493, which on a balanced two-class problem is exactly chance. The model has learned nothing whatsoever, and adding bigrams takes it to 1.000.

Why unigrams score exactly nothing here

Half the positive reviews in this corpus are written as a negated negative, *not flimsy*, and half the negative ones as a negated positive, *not sturdy*. So the word *flimsy* appears equally often on both sides, and so does *sturdy*. Every individual word is uninformative by construction, and a representation that only sees individual words cannot do better than guessing.

Pairs of adjacent words separate them completely, because *not flimsy* and *very flimsy* are different columns. The jump from 0.493 to 1.000 is that stark because this corpus was built to isolate the effect. Real text is messier and the gap is smaller, but the direction is the same and the cause is identical.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
```

```
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import CountVectorizer

for n in [(1, 1), (1, 2), (1, 3)]:
    vec = CountVectorizer(ngram_range=n).fit(train_x)
    print('ngram_range=%-8s %6d columns' % (str(n),
                                            len(vec.vocabulary_)))
```

```
ngram_range=(1, 1)       65 columns
ngram_range=(1, 2)      280 columns
ngram_range=(1, 3)     1203 columns
```

Three times the columns for bigrams, and more again for trigrams. On this corpus that is affordable. On a large collection the vocabulary grows faster than the useful signal, most of the extra columns appear once and help nothing, and you end up limiting the vocabulary by frequency.

Day 4 Reading What the Model Learned

Weights per word, and using them to debug

A model built this way is not a black box. The weights are one number per column and you can read them, which is worth doing on every text model you build.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
```

```
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
import numpy as np

vec = CountVectorizer(ngram_range=(1, 2))
X = vec.fit_transform(train_x)
clf = LogisticRegression(max_iter=2000).fit(X, train_s)
words = vec.get_feature_names_out()
order = np.argsort(clf.coef_[0])
print('%-24s %8s' % ('most positive', 'weight'))
for i in order[::-1][:6]:
    print('%-24s %8.3f' % (words[i], clf.coef_[0][i]))
print()
print('%-24s %8s' % ('most negative', 'weight'))
for i in order[:6]:
    print('%-24s %8.3f' % (words[i], clf.coef_[0][i]))
```

```
most positive             weight
hardly loose               2.760
from faulty                2.706
from scratched             2.688
hardly scratched           2.605
not faulty                 2.591
not scratched              2.576

most negative             weight
not solid                 -2.823
from solid                -2.680
hardly sturdy             -2.626
hardly reliable           -2.615
hardly solid              -2.607
not sharp                 -2.564
```

The model found the negation pairs on its own. Nobody wrote a rule about *not*, and nobody listed which adjectives are complimentary. It learned both from four hundred labelled examples, and the table above is a complete account of what it believes.

Reading these tables is a debugging tool

- If a weight makes no sense, ask why that word is in your data. It is often a leak: a phrase that only appears in one class for reasons unrelated to the task.
- If the top weights are all rare words appearing once or twice, the model is memorising and needs a minimum document frequency.
- If the top weights look reasonable and the accuracy is poor, the representation is the problem rather than the model, which is exactly what yesterday's unigram row was telling you.

Day 5 How Far Bag of Words Gets You

Four classifiers on identical features, and when to stop

A representation that ignores order is crude, and crude is not the same as useless. Here is what the bag of words approach still does well, measured against the alternatives.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
```

```
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
import time
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import make_pipeline

print('%-24s %10s %10s' % ('', 'accuracy', 'seconds'))
for name, clf in [('naive bayes', MultinomialNB()),
                  ('logistic regression',
                   LogisticRegression(max_iter=2000)),
                  ('linear svm', LinearSVC()),
                  ('random forest',
                   RandomForestClassifier(n_estimators=150,
                                          random_state=0))]:
    pipe = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)), clf)
    start = time.perf_counter()
    pipe.fit(train_x, train_s)
    took = time.perf_counter() - start
    print('%-24s %10.4f %10.2f'
          % (name, pipe.score(test_x, test_s), took))
```

```
                         accuracy    seconds
naive bayes                1.0000       0.04
```

```
logistic regression          1.0000       0.03
linear svm                   1.0000       0.03
random forest                1.0000       0.78
```

Four quite different algorithms, all fed the same features, and the differences between them are far smaller than the difference between unigrams and bigrams was yesterday. That is the general finding in text work and it is worth internalising early: the representation matters more than the classifier.

When to stop here

A TF-IDF vectoriser and a linear model trains in under a second, runs on a laptop, needs no accelerator, and can be explained line by line to somebody who has to sign it off. For topic classification, routing, tagging and filtering it is frequently as good as anything else.

Week 7 introduces the models that beat it. They cost several orders of magnitude more to run, and it is worth knowing that this is the thing they have to beat.

Day 6 Text From the Real World

Unknown words, uneven lengths, and a vocabulary that will not stop growing

Three practical problems that appear the moment text comes from the real world rather than a tidy corpus.

Words the model has never seen

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
```

```
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer().fit(train_x)
novel = 'the flange is thoroughly cromulent'
row = vec.transform([novel])
known = vec.inverse_transform(row)[0]
print('sentence: %s' % novel)
print('words the model recognises: %s' % list(known))
print('everything else was silently discarded')
```

```
sentence: the flange is thoroughly cromulent
words the model recognises: [np.str_('is'), np.str_('the')]
everything else was silently discarded
```

A word absent from the training vocabulary does not raise anything. It is dropped, and the model classifies what remains. A document made entirely of unfamiliar words becomes an empty row, and the model still returns a confident answer based on nothing at all.

Documents of very different lengths

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
import numpy as np

docs = ['sturdy pan', 'sturdy pan ' * 20]
for name, vec in [('counts', CountVectorizer()),
                  ('tfidf', TfidfVectorizer())]:
    X = vec.fit_transform(docs).toarray()
    print('%-8s row sums: %s' % (name, np.round(X.sum(axis=1), 3)))
print()
print('raw counts make a long document twenty times louder')
print('tfidf normalises each row to the same length')
```

```
counts   row sums: [ 2 40]
tfidf    row sums: [1.414 1.414]

raw counts make a long document twenty times louder
tfidf normalises each row to the same length
```

The vocabulary that keeps growing

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

```

```
def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import CountVectorizer

print('%-28s %8s' % ('setting', 'columns'))
for name, kw in [('everything', {}),
                 ('seen at least 3 times', {'min_df': 3}),
                 ('in under 80% of docs', {'max_df': 0.8}),
                 ('top 200 by frequency', {'max_features': 200})]:
    vec = CountVectorizer(ngram_range=(1, 2), **kw).fit(train_x)
    print('%-28s %8d' % (name, len(vec.vocabulary_)))
```

```
setting                       columns
everything                        280
seen at least 3 times             280
in under 80% of docs              279
top 200 by frequency              200
```

`min_df` is the setting worth reaching for first. A term appearing once cannot generalise, it can only be memorised, and dropping those terms usually shrinks the model substantially without costing accuracy.

Day 7 A Text Pipeline End to End

Assembled, cross-validated, and reported per class

The whole week as one pipeline, with the choices that were measured rather than assumed.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
```

```
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report

pipe = make_pipeline(
    TfidfVectorizer(ngram_range=(1, 2), min_df=2),
    LogisticRegression(max_iter=2000))

base = cross_val_score(make_pipeline(TfidfVectorizer(),
                                     DummyClassifier(strategy='prior')),
                       train_x, train_s, cv=5)
got = cross_val_score(pipe, train_x, train_s, cv=5)
print('%-22s %10s %10s' % ('', 'mean', 'spread'))
print('%-22s %10.4f %10.4f' % ('baseline', base.mean(),
                               base.max() - base.min()))
print('%-22s %10.4f %10.4f' % ('the pipeline', got.mean(),
                               got.max() - got.min()))
print()
pipe.fit(train_x, train_s)
print(classification_report(test_s, pipe.predict(test_x),
                            target_names=['negative', 'positive']))
```

```
                         mean     spread
baseline                0.5000     0.0000
the pipeline            1.0000     0.0000

              precision    recall  f1-score   support

    negative       1.00      1.00      1.00       360
    positive       1.00      1.00      1.00       360

    accuracy                           1.00       720
   macro avg       1.00      1.00      1.00       720
weighted avg       1.00      1.00      1.00       720
```

The checklist for any text task

1. Print some documents. Every decision below depends on what the text actually looks like.

2. Decide whether order matters for *your* task. Topic rarely needs it; opinion, negation and

instruction always do.

3. Do not remove stop words before checking that *not* is not one of them.
4. Start with TF-IDF and a linear model. It is the baseline everything else has to beat.
5. Add bigrams and measure. If they help a lot, order matters and you should keep going in that direction.
6. Use `min_df` to drop terms that appear once.
7. Read the learned weights. They are the cheapest debugging tool in this field.

The finding to carry forward

Self-Assessment

Try It Yourself

1. Why is removing stop words risky?

   a) It only works for English

   b) It slows processing

   c) Words like *not* appear on most stop lists, so two sentences with opposite meanings can become identical

   d) It reduces vocabulary too much

2. What does TF-IDF do?

   a) Removes punctuation

   b) Sorts documents by length

   c) Counts words

   d) Weights a word higher when it is frequent here and rare across the collection

3. Sentiment scored 0.493 with unigrams and 1.000 with bigrams. Why?

   a) Half the reviews express sentiment through negation, so single words appear equally on both sides and only pairs separate them

   b) Bigrams have more columns

   c) The test set was different

   d) The classifier changed

4. Category classification scored 1.000 whatever the representation. What does that tell you?

   a) The task is broken

   b) Topic needs only the nouns, so word order adds nothing for that task

   c) The model overfitted

   d) The categories overlap

5. Four different classifiers on identical features barely differed. What is the general lesson?

a) Only linear models work on text

b) Classifiers are interchangeable everywhere

c) In text work the representation usually matters more than the classifier consuming it

d) The data was too easy

6. A word absent from the training vocabulary is handled how?

   a) It is added to the vocabulary

   b) The document is skipped

   c) An error is raised

   d) It is silently dropped, and a document of only unknown words becomes an empty row the model still classifies confidently

7. What does `min_df` do and why use it?

   a) Drops terms appearing in fewer than n documents, since a term seen once can only be memorised

   b) Sets the minimum accuracy

   c) Controls the learning rate

   d) Limits document length

8. Why read the learned per-word weights?

   a) To publish them

   b) They are a complete account of a linear text model and the cheapest debugging tool available

   c) To compress the model

   d) They are needed for prediction

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

The Standard Language Tasks

Week 5

OBJECTIVES

Entity extraction, keywords, topics, similarity and sentiment, each with the measurement it actually needs.

Most language work in industry is one of about eight standard tasks. This week builds most of them, and is careful about the fact that each needs a different measurement: accuracy is meaningless for entity spans, and topics have nothing to be scored against at all. It ends on the limit that motivates the next week, where two sentences meaning the same thing score lower for similarity than two meaning the opposite.

Day 1 The Eight Standard Tasks

What language work actually consists of, and which half can be measured

Week 4 built one classifier. Most language work in industry is one of a handful of standard tasks, and knowing the list tells you what to search for and what a reasonable result looks like.

Task	Question	Output
Classification	What kind of document is this	One label per document
Sentiment	What attitude does it express	A label or a score
Named entity recognition	Which spans are people, places, dates	Labelled spans
Keyword extraction	What is this about	A few terms
Topic modelling	What themes run through the collection	Groups, unsupervised
Similarity and search	Which documents resemble this one	A ranking
Summarisation	What does this say, briefly	New text
Translation	What does this say in another language	New text

The first six are solvable to a useful standard with the methods in this week. The last two produce new text, which is a different kind of problem and is where week 7's models are genuinely required rather than merely fashionable.

The dividing line worth noticing

Tasks that pick from existing options can be measured against a known answer, so you can tell whether they work. Tasks that generate text have no single right answer, which makes them

much harder to evaluate and much easier to be fooled by. That distinction matters more than the apparent difficulty of the task.

Day 2 Finding the Things in the Text

Entity extraction with rules, and the ambiguity rules cannot settle

Finding the people, places, dates and amounts in a document is one of the most commercially useful of these tasks, because it turns prose into rows in a table.

Rules first, as always

```
DOCS = [
    'Acme Ltd shipped the order from Manchester on 3 March 2024 for 249.99',
    'Dr Patel at Riverside Clinic referred the patient on 12 June 2023',
    'Contact Jane Whitfield at jane@northgate.co.uk or 0161 496 0000',
    'Northgate Systems reported revenue of 1.4m for the quarter',
    'The invoice from Boyd and Sons is dated 7 January 2025',
]
import re

PATTERNS = [
    ('DATE', r'\b\d{1,2} (?:January|February|March|April|May|June|July|'
             r'August|September|October|November|December) \d{4}\b'),
    ('MONEY', r'\b\d+(?:\.\d{2})?m?\b(?= for| revenue|$)'),
    ('EMAIL', r'\b[\w.]+@[\w.]+\b'),
    ('PHONE', r'\b0\d{3} \d{3} \d{4}\b'),
    ('ORG', r'\b[A-Z][A-Za-z]+(?: [A-Z][A-Za-z]+)* '
            r'(?:Ltd|Systems|Clinic|and Sons)\b'),
]

for doc in DOCS:
    found = []
    for label, pattern in PATTERNS:
        for m in re.finditer(pattern, doc):
            found.append('%s=%s' % (label, m.group(0)))
    print(doc)
    print('   %s' % (found or 'nothing found'))
```

```
Acme Ltd shipped the order from Manchester on 3 March 2024 for 249.99
   ['DATE=3 March 2024', 'MONEY=2024', 'MONEY=249.99', 'ORG=Acme Ltd']
Dr Patel at Riverside Clinic referred the patient on 12 June 2023
   ['DATE=12 June 2023', 'MONEY=2023', 'ORG=Riverside Clinic']
Contact Jane Whitfield at jane@northgate.co.uk or 0161 496 0000
   ['MONEY=0000', 'EMAIL=jane@northgate.co.uk', 'PHONE=0161 496 0000']
Northgate Systems reported revenue of 1.4m for the quarter
   ['MONEY=4m', 'ORG=Northgate Systems']
The invoice from Boyd and Sons is dated 7 January 2025
   ['DATE=7 January 2025', 'MONEY=2025', 'ORG=Boyd and Sons']
```

Dates, emails and phone numbers are genuinely well suited to rules: they have a defined format, the format rarely changes, and a regular expression is exact. This is week 3's lesson arriving in a language context.

Where rules stop

```
DOCS = [
    'Acme Ltd shipped the order from Manchester on 3 March 2024 for 249.99',
    'Dr Patel at Riverside Clinic referred the patient on 12 June 2023',
    'Contact Jane Whitfield at jane@northgate.co.uk or 0161 496 0000',
    'Northgate Systems reported revenue of 1.4m for the quarter',
    'The invoice from Boyd and Sons is dated 7 January 2025',
]
import re

ORG = re.compile(r'\b[A-Z][A-Za-z]+(?: [A-Z][A-Za-z]+)* '
                 r'(?:Ltd|Systems|Clinic|and Sons)\b')
hard = [
    'Apple sold more units than expected',
    'I ate an apple at Apple headquarters',
    'Washington signed the order in Washington',
    'The Manchester office called Manchester United',
]
for text in hard:
    print('%-46s %s' % (text, ORG.findall(text) or []))
print()
print('the suffix rule finds nothing, and the ambiguity is the point')
```

```
Apple sold more units than expected            []
I ate an apple at Apple headquarters           []
Washington signed the order in Washington      []
The Manchester office called Manchester United []

the suffix rule finds nothing, and the ambiguity is the point
```

Apple is a company or a fruit depending on the sentence. *Washington* is a person, a city or a state. No list of suffixes settles these, because the evidence is the surrounding words, and using surrounding words as evidence is exactly what a learned model does.

Watch Out

The practical answer is usually both

Use rules for the things with a fixed format: dates, references, postcodes, amounts. Use a learned model for the things that depend on context: people, organisations, products. Systems that try to do either half with the wrong tool are common and both failure modes are expensive.

Day 3 What Is This About

Keywords from TF-IDF, and themes with nothing to score against

Deciding what a document is *about*, without being given any categories, is a different shape of problem. Nobody supplies labels, so nothing can be scored against an answer key, and that changes how you work.

Keywords by weight

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
```

```
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np

vec = TfidfVectorizer(ngram_range=(1, 2), min_df=2)
X = vec.fit_transform(texts)
words = vec.get_feature_names_out()
for i in [0, 1, 2]:
    row = X[i].toarray()[0]
    top = np.argsort(row)[::-1][:4]
    print('%s' % texts[i])
    print('   → %s' % ', '.join(words[j] for j in top if row[j] > 0))
```

```
after a month the keyboard is still quite reliable, as described in the listing
   → quite reliable, reliable as, still quite, keyboard is
for the price the charger is not sturdy, second one i have bought
   → not sturdy, sturdy second, charger is, charger
for the price the lid is hardly scratched, as described in the listing
   → hardly scratched, scratched as, lid is, the lid
```

TF-IDF gives keyword extraction almost for free: the highest weighted terms in a document are, by construction, the ones that are frequent here and rare elsewhere, which is a workable definition of what something is about.

Themes across a whole collection

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
```

```
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
import numpy as np

vec = TfidfVectorizer(min_df=3, stop_words='english')
X = vec.fit_transform(texts)
words = vec.get_feature_names_out()
model = NMF(n_components=3, random_state=0, max_iter=400).fit(X)
for k, comp in enumerate(model.components_):
```

```
    top = np.argsort(comp)[::-1][:6]
    print('theme %d: %s' % (k + 1, ', '.join(words[i] for i in top)))
```

```
theme 1: tuesday, ordered, packaging, fine, listing, described
theme 2: week, took, delivery, like, wanted, far
theme 3: second, bought, far, price, month, wanted
```

Nobody told it there were three product categories. It found groups of words that tend to occur together, and those groups correspond to the categories because that is the strongest structure in the text.

Watch Out

Topic models are suggestive, not conclusive

The number of themes is chosen by you, the result changes if you change it, and there is no accuracy to check it against. What comes out is a reading aid: it tells you what to go and look at. Treating the themes as though they were discovered facts about the collection is how topic modelling gets misused.

Day 4 Similarity and Search

Cosine similarity, and the paraphrase that scores lower than its opposite

Similarity underpins search, deduplication, recommendation and retrieval. With documents already represented as vectors, it is one operation.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
```

```
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

vec = TfidfVectorizer(ngram_range=(1, 2))
X = vec.fit_transform(texts)
query = 'the battery is not reliable'
sims = cosine_similarity(vec.transform([query]), X)[0]
print('query: %s' % query)
for i in np.argsort(sims)[::-1][:4]:
    print('  %.3f  %s' % (sims[i], texts[i]))
```

```
query: the battery is not reliable
  0.715  for the price the battery is not reliable, the packaging was fine
  0.501  the battery is not scratched, delivery took a week
  0.482  the battery is not excellent, second one i have bought
  0.465  the kettle is not reliable, delivery took a week
```

What word matching cannot do

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

pairs = [
    ('the film was excellent', 'the movie was superb'),
    ('the film was excellent', 'the film was terrible'),
]
vec = TfidfVectorizer().fit([s for pair in pairs for s in pair])
for a, b in pairs:
    sim = cosine_similarity(vec.transform([a]), vec.transform([b]))[0][0]
    print('%.3f   %-28s %s' % (sim, a, b))
print()
print('two sentences that mean the same thing score lower than')
print('two that mean opposite things, because words are all it sees')
```

```
0.272   the film was excellent       the movie was superb
0.543   the film was excellent       the film was terrible

two sentences that mean the same thing score lower than
two that mean opposite things, because words are all it sees
```

That result is the clearest possible statement of the limit. *Film* and *movie* are different columns, so a paraphrase shares almost nothing. *Excellent* and *terrible* are also different columns, but the surrounding words are identical, so the opposite pair scores higher.

> **The gap week 6 exists to fill**

Day 5 Sentiment, and Why It Is Harder Than It Looks

Negation, sarcasm, mixed opinion, and the domain that inverts the meaning

Sentiment deserves its own day because it is the task most often deployed carelessly, and because its failure modes are instructive.

```
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
```

```
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
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))
model.fit(train_x, train_s)
```

```

tricky = [
    'the handle is not flimsy',
    'the handle is flimsy',
    'well that was money well spent',
    'the screen is fine i suppose',
    'the blade is sharp but the handle is loose',
]
for text in tricky:
    p_pos = model.predict_proba([text])[0][1]
    print('%.3f  %s' % (p_pos, text))
```

```
0.743  the handle is not flimsy
0.456  the handle is flimsy
0.500  well that was money well spent
0.459  the screen is fine i suppose
0.405  the blade is sharp but the handle is loose
```

The first two are handled, which is what week 4's bigrams bought. The rest are the standard hard cases and the model has no way to reach them: sarcasm inverts the meaning of positive words, faint praise sits between the classes, and a sentence with one positive and one negative clause has no single answer at all.

The problem with one label per document

- **Mixed opinions.** A review praising the blade and criticising the handle is not positive or negative, and forcing it into one is discarding the useful information.
- **Aspect based sentiment** is the version of the task that predicts an opinion per feature rather than per document, and it is usually what the business actually wanted.
- **Sarcasm and understatement** require knowing what is normal, which is context no sentence contains.
- **Domain shift.** A model trained on product reviews reads clinical notes badly, because *positive* means the opposite there.

Watch Out

The last one is not a joke

In a medical context a positive result is usually bad news. A sentiment model moved between domains without retraining will get this exactly backwards while remaining entirely confident, and it is a good illustration that these models learn the vocabulary of a particular world rather than the meaning of words.

Day 6 Measuring Each Task Properly

Why accuracy is wrong for spans, and what to do without ground truth

Every task in this week needs a different measurement, and using the wrong one is how projects report success and deliver nothing.

```
import random
```

```

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
```

```
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics import classification_report, confusion_matrix

model = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                      LogisticRegression(max_iter=2000))
model.fit(train_x, train_c)
pred = model.predict(test_x)
print(classification_report(test_c, pred))
```

```
              precision    recall  f1-score   support

    clothing       1.00      1.00      1.00       244
     kitchen       1.00      1.00      1.00       239
      laptop       1.00      1.00      1.00       237

    accuracy                           1.00       720
   macro avg       1.00      1.00      1.00       720
weighted avg       1.00      1.00      1.00       720
```

Measuring entity extraction

```
true = {('Acme Ltd', 'ORG'), ('Manchester', 'LOC'),
        ('3 March 2024', 'DATE')}
found = {('Acme Ltd', 'ORG'), ('3 March 2024', 'DATE'),
         ('order', 'ORG')}

hit = true & found
precision = len(hit) / len(found)
recall = len(hit) / len(true)
f1 = 2 * precision * recall / (precision + recall)
print('correct spans  %s' % sorted(hit))
print('missed         %s' % sorted(true - found))
print('invented       %s' % sorted(found - true))
print()
print('precision %.3f  recall %.3f  f1 %.3f' % (precision, recall, f1))
```

```
correct spans  [('3 March 2024', 'DATE'), ('Acme Ltd', 'ORG')]
missed         [('Manchester', 'LOC')]
invented       [('order', 'ORG')]

precision 0.667  recall 0.667  f1 0.667
```

Accuracy is meaningless here, because there is no fixed number of decisions being made. The measurement has to be over spans: how many of the ones you found were right, and how many of

the real ones did you find. Note also that a span found with the wrong label counts as both a miss and an invention.

Measuring the unsupervised tasks

- **Topics** have no ground truth. Judge them by whether a person can name each theme, and by whether documents assigned to a theme belong together.
- **Search and ranking** are measured by where the good results land, not by whether they appear at all. A relevant document at position fifty is a failure.
- **Summarisation** is measured badly by every automatic metric available, all of which compare word overlap with a reference summary and none of which detect a fluent summary that says something untrue.

Day 7 One Document, Several Tasks

A realistic pipeline built from four different techniques

One document, run through several of the week's tasks, which is what a real pipeline looks like.

```
DOCS = [
    'Acme Ltd shipped the order from Manchester on 3 March 2024 for 249.99',
    'Dr Patel at Riverside Clinic referred the patient on 12 June 2023',
    'Contact Jane Whitfield at jane@northgate.co.uk or 0161 496 0000',
    'Northgate Systems reported revenue of 1.4m for the quarter',
    'The invoice from Boyd and Sons is dated 7 January 2025',
]
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
```

```
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
import re
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics.pairwise import cosine_similarity

sent = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                     LogisticRegression(max_iter=2000)).fit(train_x,
                                                            train_s)
cat = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                    LogisticRegression(max_iter=2000)).fit(train_x,
                                                           train_c)
vec = TfidfVectorizer(ngram_range=(1, 2)).fit(texts)
X = vec.transform(texts)

incoming = ('the trackpad is not faulty but the battery is quite '
            'noisy, ordered on 3 March 2024')
```

```
print('incoming: %s' % incoming)
print()
print('category   %s' % cat.predict([incoming])[0])
print('sentiment  %.3f positive' % sent.predict_proba([incoming])[0][1])
dates = re.findall(r'\b\d{1,2} [A-Z][a-z]+ \d{4}\b', incoming)
print('dates      %s' % dates)
row = vec.transform([incoming])
words = vec.get_feature_names_out()
top = np.argsort(row.toarray()[0])[::-1][:3]
print('keywords   %s' % ', '.join(words[i] for i in top))
sims = cosine_similarity(row, X)[0]
print('closest    %s' % texts[int(np.argmax(sims))])
```

```
incoming: the trackpad is not faulty but the battery is quite noisy, ordered on 3 March 2024

category   laptop
sentiment  0.586 positive
dates      ['3 March 2024']
keywords   quite noisy, not faulty, noisy ordered
closest    the battery is quite noisy, ordered on tuesday
```

Four different techniques, each suited to its part of the job: a learned classifier for category, another for sentiment, a regular expression for the date, and vector similarity for retrieval. Nothing here needed a large model, and each piece can be measured on its own.

The checklist

1. Name the task before choosing a method. Most language work is one of eight standard shapes.
2. Use rules for anything with a fixed format, and learning for anything that depends on context.
3. For unsupervised tasks, accept that there is no score and design human review into the process.
4. Measure spans with precision and recall, not accuracy.
5. Check the hard cases by hand: negation, mixed opinion, sarcasm. The aggregate number hides all of them.
6. Before deploying anywhere new, test on text from that domain. Vocabulary does not transfer as well as it appears to.

Self-Assessment

Try It Yourself

1. Which language tasks can be scored against a known answer?

   a) All of them

   b) Those that pick from existing options; generating text has no single right answer and is much easier to be fooled by

   c) Only classification

   d) None reliably

2. Where do rules work well for entity extraction?

a) Everywhere, if you write enough

b) For people and organisations

c) For things with a fixed format such as dates, emails and postcodes

d) For sarcasm

3. Why can rules not settle whether *Apple* is a company?

   a) Rules cannot match capital letters

   b) It requires a bigger dictionary

   c) The word is too short

   d) The evidence is the surrounding words, and using context as evidence is what a learned model does

4. How should topic model output be treated?

   a) As a reading aid that tells you where to look, since the number of themes is chosen by you and nothing scores it

   b) As a classifier

   c) As ground truth labels

   d) As discovered facts about the collection

5. Two sentences meaning the same thing scored lower than two meaning the opposite. Why?

   a) A bug in the metric

   b) Word matching sees only shared columns, and a paraphrase shares almost none while an opposite pair shares nearly all its other words

   c) The documents were different lengths

   d) TF-IDF was misconfigured

6. What is aspect based sentiment for?

   a) Reducing label cost

   b) Faster inference

   c) Predicting an opinion per feature rather than per document, which is usually what was actually wanted

   d) Handling multiple languages

7. Why is accuracy the wrong metric for entity extraction?

   a) Spans cannot be compared

   b) It is only wrong for dates

   c) It is too strict

   d) There is no fixed number of decisions; the measurement has to be over spans, with precision and recall

8. A sentiment model trained on product reviews is applied to clinical notes. What happens?

   a) It can invert, since *positive* usually means bad news there, while remaining entirely confident

b) It refuses to run

c) Accuracy drops slightly

d) It transfers well

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

Meaning and Embeddings

Week 6

OBJECTIVES

How a machine gets from words as arbitrary symbols to words with relationships, built from counting and compression.

Every method up to now treats words as arbitrary distinct symbols, so a paraphrase and its opposite look equally unrelated. This week fixes that with one idea: words used in similar contexts mean similar things. Twenty sentences and a counting loop are enough to place cat near dog and van near car with no dictionary involved. It also shows the same mechanism faithfully recording an association about who does which job that nobody wrote down, which is the subject of week 11.

Day 1 You Shall Know a Word by the Company It Keeps

The distributional hypothesis, and counting context

Week 5 ended on a failure: two sentences meaning the same thing scored lower for similarity than two meaning the opposite. The cause is that every method so far treats words as arbitrary distinct symbols. *Film* and *movie* are simply two different columns, no more related than *film* and *hydraulic*.

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
```

```
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
print('%d sentences, %d distinct words' % (len(SENTENCES), len(VOCAB)))
for s in SENTENCES[:4]:
    print('  ' + s)
```

```
20 sentences, 90 distinct words
  the dog barked at the postman in the street
  the cat slept on the warm windowsill all day
  the dog slept on the rug by the fire
  the cat chased a mouse across the kitchen floor
```

Counting company

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
```

```
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
for word in ['dog', 'car']:
    row = M[INDEX[word]]
    top = np.argsort(row)[::-1][:6]
    print('%-6s appears near: %s'
          % (word, ', '.join('%s(%d)' % (VOCAB[i], row[i])
                             for i in top if row[i] > 0)))
```

```
dog    appears near: the(2), slept(1), on(1), barked(1), meat(1), at(1)
car    appears near: the(3), needs(1), down(1), drove(1), fuel(1)
```

That row of counts is already a representation of the word, and it is one in which meaning is present. Nothing in the corpus states that a dog is an animal or a car is a vehicle. The information is in the company each word keeps.

Day 2 Similar Words Have Similar Rows

Neighbours from twenty sentences, and why counting will not scale

If similar words keep similar company, then similar words have similar rows, and similarity between rows is something we can compute.

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
```

```
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
for word in ['dog', 'car', 'bread', 'london']:
    pairs = similar(M, word, 4)
    print('%-8s → %s' % (word,
                         ', '.join('%s %.2f' % p for p in pairs)))
```

```
dog      → cat 0.73, slept 0.62, rug 0.62, postman 0.62
car      → van 1.00, i 0.78, in 0.74, slept 0.71
bread    → strong 0.52, for 0.49, ate 0.49, from 0.49
london   → manchester 0.79, to 0.32, large 0.32, passengers 0.32
```

Cat comes out near *dog*, and *van* near *car*, from twenty sentences and a counting loop. No dictionary was consulted and nobody defined any of these words.

This is the whole idea, at the smallest possible scale

Word vectors trained on billions of sentences do the same thing with more sophistication: they compress the counts, weight them better and use wider context. The mechanism above is genuinely what is happening underneath, which is worth knowing because it explains both the strengths

and the failures in the rest of this week.

The problem with raw counts

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
```

```
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
print('the co-occurrence matrix is %d by %d' % M.shape)
print('%.1f%% of it is zero' % (100 * (M == 0).mean()))
print()
print('and it grows with the square of the vocabulary:')
for v in [100, 10_000, 1_000_000]:
    print('  %9d words → %15d cells' % (v, v * v))
```

```
the co-occurrence matrix is 90 by 90
94.3% of it is zero

and it grows with the square of the vocabulary:
        100 words →             10000 cells
      10000 words →         100000000 cells
    1000000 words →     1000000000000 cells
```

A million word vocabulary would need a matrix with a trillion cells, almost all of them zero. Raw counting does not scale, and the fix is to compress each row into a few hundred numbers that preserve the distances.

Day 3 Compressing Meaning Into a Few Numbers

Embeddings, neighbourhoods that survive, and the oversold analogy

Compressing the matrix while keeping what matters is a standard operation, and it produces the thing everybody means by an embedding.

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
```

```
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
from sklearn.decomposition import TruncatedSVD

# log scaling stops very common words dominating every direction
weighted = np.log1p(M)
E = TruncatedSVD(n_components=8, random_state=0).fit_transform(weighted)
print('each word is now %d numbers instead of %d' % (E.shape[1],
                                                     M.shape[1]))
print()
for word in ['dog', 'car', 'bread', 'london']:
    pairs = similar(E, word, 4)
    print('%-8s → %s' % (word,
                         ', '.join('%s %.2f' % p for p in pairs)))
```

```
each word is now 8 numbers instead of 90

dog      → on 0.95, cat 0.93, slept 0.89, warm 0.87
car      → van 1.00, market 0.98, i 0.97, kitchen 0.96
bread    → cheese 0.97, was 0.92, jam 0.91, came 0.90
london   → manchester 0.93, passengers 0.80, hour 0.73, is 0.70
```

Eight numbers per word instead of one column per vocabulary entry, and the neighbourhoods survive. That compression is what makes embeddings practical: a real one is three hundred numbers for a vocabulary of millions.

Directions in the space mean something

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
```

```
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
from sklearn.decomposition import TruncatedSVD

E = TruncatedSVD(n_components=8,
                 random_state=0).fit_transform(np.log1p(M))

def analogy(a, b, c, n=3):
    """a is to b as c is to what."""
    target = E[INDEX[b]] - E[INDEX[a]] + E[INDEX[c]]
    norms = np.linalg.norm(E, axis=1) * np.linalg.norm(target) + 1e-9
```

```
    sims = (E @ target) / norms
    out = []
    for i in np.argsort(sims)[::-1]:
        if VOCAB[i] not in (a, b, c):
            out.append((VOCAB[i], float(sims[i])))
        if len(out) == n:
            return out

print('dog is to puppy as cat is to ...')
print('   %s' % analogy('dog', 'puppy', 'cat'))
print()
print('car is to motorway as train is to ...')
print('   %s' % analogy('car', 'motorway', 'train'))
```

```
dog is to puppy as cat is to ...
   [('chased', 0.911852059028957), ('across', 0.8425314884061901), ('mouse',
    ↪ 0.8273696571298044)]

car is to motorway as train is to ...
   [('to', 0.92566759284489), ('passengers', 0.8777832552791965), ('carried',
    ↪ 0.8417030748678817)]
```

Watch Out

Analogies are the most oversold result in this field

On twenty sentences the answers are wobbly, and even on large embeddings the famous examples are curated. The published results typically exclude the three input words from the answer, which is doing a substantial amount of the work, and most analogies you try yourself will not come out.
The genuinely reliable property is the first one: similar words are near each other. That is what embeddings are used for in practice, and it is enough.

Day 4 From Words to Documents

Averaging vectors, and the order it still cannot see

A document is more than a bag of its words' meanings, but averaging the vectors is a surprisingly serviceable start and it fixes week 5's paraphrase problem.

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
```

```
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
from sklearn.decomposition import TruncatedSVD

E = TruncatedSVD(n_components=8,
                 random_state=0).fit_transform(np.log1p(M))

def embed(sentence):
    vecs = [E[INDEX[w]] for w in sentence.split() if w in INDEX]
    return np.mean(vecs, axis=0) if vecs else np.zeros(E.shape[1])

def sim(a, b):
    va, vb = embed(a), embed(b)
    return float(va @ vb / (np.linalg.norm(va) * np.linalg.norm(vb)
                            + 1e-9))

pairs = [('the dog slept', 'the cat slept'),
         ('the dog slept', 'the van needs fuel'),
         ('i drove the car', 'i drove the van')]
```

```
for a, b in pairs:
    print('%.3f   %-22s %s' % (sim(a, b), a, b))
```

```
0.997   the dog slept          the cat slept
0.944   the dog slept          the van needs fuel
1.000   i drove the car        i drove the van
```

What averaging throws away

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()
```

```

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
from sklearn.decomposition import TruncatedSVD

E = TruncatedSVD(n_components=8,
                 random_state=0).fit_transform(np.log1p(M))

def embed(sentence):
    vecs = [E[INDEX[w]] for w in sentence.split() if w in INDEX]
    return np.mean(vecs, axis=0) if vecs else np.zeros(E.shape[1])

a = embed('the dog chased the cat')
b = embed('the cat chased the dog')
print('identical vectors: %s' % bool(np.allclose(a, b)))
print('averaging is order blind, exactly like bag of words')
```

```
identical vectors: True
averaging is order blind, exactly like bag of words
```

Two sentences with opposite meanings and the same average. Embeddings solved the vocabulary problem, they did not solve the order problem, and that is precisely the gap the transformer architecture in week 7 was built to close.

Day 5 Embeddings Learn the Assumptions Too

An association nobody wrote down, recorded faithfully

Embeddings learn from the text they are given, which includes everything that text assumes. This is not a defect that better training removes, it is what the method does.

```
import numpy as np

# A corpus with a pattern nobody put there deliberately, of exactly
# the kind found in real text scraped from the web.
SENTENCES = [
    'the nurse checked her patients before the ward round',
    'the nurse collected her notes from the desk',
    'the surgeon washed his hands before the operation',
    'the surgeon reviewed his notes before the operation',
    'the secretary typed her letters in the morning',
    'the engineer checked his drawings in the workshop',
    'she trained as a nurse at the local hospital',
    'he trained as a surgeon at the teaching hospital',
    'she worked as a secretary for the practice',
    'he worked as an engineer for the firm',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
M = np.zeros((len(VOCAB), len(VOCAB)))
```

```
for s in TOKENS:
    for i, w in enumerate(s):
        for j in range(max(0, i - 3), min(len(s), i + 4)):
            if i != j:
                M[INDEX[w], INDEX[s[j]]] += 1

def closeness(a, b):
    va, vb = M[INDEX[a]], M[INDEX[b]]
    return float(va @ vb / (np.linalg.norm(va) * np.linalg.norm(vb)
                            + 1e-9))

print('%-12s %8s %8s' % ('job', 'she/her', 'he/his'))
for job in ['nurse', 'surgeon', 'secretary', 'engineer']:
    fem = max(closeness(job, w) for w in ['she', 'her'])
    masc = max(closeness(job, w) for w in ['he', 'his'])
    print('%-12s %8.3f %8.3f' % (job, fem, masc))
```

```
job           she/her   he/his
nurse           0.663    0.591
surgeon         0.573    0.650
secretary       0.572    0.480
engineer        0.531    0.560
```

The corpus contains no statement about who can do which job. It contains a statistical association, and the representation records it faithfully, because recording associations is the entire mechanism.

Watch Out

Why this matters more than it first appears

An embedding is rarely the final product. It is an input to a screening model, a search ranker or a recommender, and the association travels into all of them silently. A CV screening system built on embeddings can rank candidates by association with a job title without any field for gender ever being present.
Removing the obvious words does not fix it, because the association is distributed across the whole vocabulary. Week 11 returns to this properly with measurements and remedies.

Day 6 Where Static Vectors Run Out

One vector per word, and nothing for words never seen

Two limits of static embeddings, both of which motivate what came next.

One vector per word, whatever it means

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
```

```
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
for word in ['london', 'street']:
    print('%-8s → %s' % (word, [w for w, _ in similar(M, word, 4)]))
print()
print('the word "bank" would get one vector averaging')
print('the riverside sense and the financial sense together,')
print('which is a compromise that suits neither')
```

```
london   → ['manchester', 'to', 'large', 'passengers']
street   → ['garden', 'postman', 'in', 'ball']

the word "bank" would get one vector averaging
the riverside sense and the financial sense together,
```

```
which is a compromise that suits neither
```

A static embedding assigns one vector per word type. Every sense of the word is blended into it, weighted by how common each sense is in the training text. For genuinely ambiguous words the resulting vector represents nothing that anybody ever means.

Nothing outside the vocabulary

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
```

```
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
for word in ['dog', 'aardvark', 'refactoring']:
    print('%-14s in vocabulary: %s' % (word, word in INDEX))
print()
print('a word not in the training text has no vector at all,')
print('so misspellings, product codes and new terms are invisible')
```

```
dog            in vocabulary: True
aardvark       in vocabulary: False
refactoring    in vocabulary: False

a word not in the training text has no vector at all,
so misspellings, product codes and new terms are invisible
```

What fixed both

- **Contextual embeddings.** Compute the vector from the sentence rather than looking it up, so *bank* gets a different vector in each sense. This is what a transformer produces and it is the main reason they replaced static vectors.
- **Subword tokenisation.** Break rare words into pieces that are in the vocabulary, so an unseen word still receives a representation assembled from familiar parts.
- **Sentence embeddings.** Models trained to place whole sentences rather than words, which is what modern semantic search and retrieval actually use.

Day 7 Words Against Meaning

The same question answered both ways, and what comes next

The week in one comparison: the same similarity question, answered by word matching and by embeddings.

```
import numpy as np

# A small corpus written so that related words share company. Nothing
# in it states that a dog is like a cat; the sentences merely use them
# in the same way, which turns out to be enough.
SENTENCES = [
    'the dog barked at the postman in the street',
    'the cat slept on the warm windowsill all day',
    'the dog slept on the rug by the fire',
    'the cat chased a mouse across the kitchen floor',
    'my dog eats meat and biscuits every morning',
    'my cat eats fish and biscuits every morning',
    'the puppy barked and chased the ball in the garden',
    'the kitten slept and chased a ball of wool',
    'i drove the car down the motorway to london',
    'i drove the van down the motorway to manchester',
    'the car needs fuel and a service before winter',
```

```
    'the van needs fuel and new tyres before winter',
    'the lorry carried freight along the motorway at night',
    'the train carried passengers to london every hour',
    'she ate bread and cheese for lunch at her desk',
    'he ate bread and jam for breakfast before work',
    'the bread was fresh from the bakery this morning',
    'the cheese was strong and came from the market',
    'london is a large city with an old river',
    'manchester is a large city with a busy centre',
]
TOKENS = [s.split() for s in SENTENCES]
VOCAB = sorted({w for s in TOKENS for w in s})
INDEX = {w: i for i, w in enumerate(VOCAB)}
def cooccurrence(window=2):
    """Count how often each pair of words appears near each other."""
    M = np.zeros((len(VOCAB), len(VOCAB)))
    for sentence in TOKENS:
        for i, word in enumerate(sentence):
            lo = max(0, i - window)
            hi = min(len(sentence), i + window + 1)
            for j in range(lo, hi):
                if j != i:
                    M[INDEX[word], INDEX[sentence[j]]] += 1
    return M

M = cooccurrence()

def similar(vectors, word, n=5):
    v = vectors[INDEX[word]]
    norms = np.linalg.norm(vectors, axis=1) * np.linalg.norm(v) + 1e-9
    sims = (vectors @ v) / norms
    order = np.argsort(sims)[::-1]
    return [(VOCAB[i], float(sims[i])) for i in order if VOCAB[i] != word][:n]
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

E = TruncatedSVD(n_components=8,
                 random_state=0).fit_transform(np.log1p(M))

def embed(sentence):
    vecs = [E[INDEX[w]] for w in sentence.split() if w in INDEX]
    return np.mean(vecs, axis=0) if vecs else np.zeros(E.shape[1])

def emb_sim(a, b):
    va, vb = embed(a), embed(b)
    return float(va @ vb / (np.linalg.norm(va) * np.linalg.norm(vb)
                            + 1e-9))

tfidf = TfidfVectorizer().fit(SENTENCES)
def word_sim(a, b):
    return float(cosine_similarity(tfidf.transform([a]),
                                   tfidf.transform([b]))[0][0])

```

```
pairs = [('the dog slept', 'the cat slept'),
         ('i drove the car', 'i drove the van'),
         ('the dog barked', 'the puppy barked')]
print('%-40s %10s %10s' % ('', 'words', 'embedding'))
for a, b in pairs:
    label = '%s / %s' % (a, b)
    print('%-40s %10.3f %10.3f' % (label, word_sim(a, b),
                                   emb_sim(a, b)))
```

```
                                         words  embedding
the dog slept / the cat slept            0.556      0.997
i drove the car / i drove the van        0.547      1.000
the dog barked / the puppy barked        0.537      0.990
```

On every pair the embedding scores the relatedness higher, because it knows that *dog* and *cat* keep similar company while TF-IDF only knows they are different columns.

What the week established

- Meaning can be derived from context alone, with no dictionary, because words used similarly are used in similar company.
- Compressing the co-occurrence counts gives dense vectors that keep the neighbourhoods and fit in memory.
- Nearest neighbours are the reliable property. Analogies are largely a demonstration.
- Averaging word vectors gives a usable document vector and remains blind to word order.
- Whatever associations exist in the training text end up in the vectors, and travel silently into anything built on them.
- One vector per word cannot represent an ambiguous word, and nothing outside the vocabulary gets a vector at all.

The two problems week 7 solves

Self-Assessment

Try It Yourself

1. What is the distributional hypothesis?

   a) Words appearing in similar contexts tend to mean similar things
   b) Meaning is normally distributed
   c) Every word has one meaning
   d) Words are distributed evenly in text

2. How were cat and dog found to be related?

   a) From a dictionary
   b) From counting which words appear near them, with no definitions involved

c) From their spelling

d) They were labelled as animals

3. Why can raw co-occurrence counting not scale?

   a) It needs labels

   b) Counting is slow

   c) The matrix grows with the square of the vocabulary and is almost entirely zero

   d) Counts become too large

4. Which property of embeddings is genuinely reliable?

   a) That vectors sum to one

   b) That every dimension is interpretable

   c) Analogies

   d) That similar words are near each other

5. Averaging word vectors for a document leaves which problem unsolved?

   a) Word order, so *the dog chased the cat* and its reverse are identical

   b) Rare words

   c) Document length

   d) Vocabulary size

6. The job association appeared in the vectors because of what?

   a) A bug in the code

   b) A statistical association in the corpus, recorded faithfully because recording associations is the entire mechanism

   c) Deliberate labelling

   d) The compression step

7. Why does removing obvious words not fix a learned association?

   a) Because embeddings are random

   b) It does fix it

   c) The association is distributed across the whole vocabulary rather than carried by a few words

   d) The words come back during training

8. What does a static embedding do with an ambiguous word like *bank*?

   a) Refuses to embed it

   b) Picks the most common sense only

   c) Creates one vector per sense

   d) Blends every sense into one vector that represents nothing anybody means

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

Transformers and Large Language Models

Week 7

OBJECTIVES

Contextual representations, attention worked through by hand, and an honest comparison against the simple methods.

This week closes the gap week 6 left: a model that reads the whole sentence gives a different vector to the same word in different contexts, which fixes ambiguity and word order at once. Attention, the mechanism responsible, is worked through by hand in about ten lines. The week is also direct about two things: a pretrained transformer does not beat TF-IDF bigrams on this task, and hallucination is not a defect but the training objective behaving exactly as specified.

Day 1 Vectors That Depend on the Sentence

Subword tokens, and the unknown word that no longer breaks anything

Week 6 left two problems. A static embedding gives one vector per word whatever it means in context, and averaging those vectors is blind to word order. Both are solved by the same change: compute the vector from the sentence rather than looking it up.

Subword tokenisation, which solves the unknown word problem

```
import os
os.environ['HF_HUB_OFFLINE'] = '0'
import torch
from transformers import AutoTokenizer, AutoModel

NAME = 'prajjwal1/bert-tiny'
tok = AutoTokenizer.from_pretrained(NAME)
model = AutoModel.from_pretrained(NAME)
model.eval()
for text in ['the trackpad is faulty', 'the zibbleflex is faulty']:
    ids = tok(text)['input_ids']
    print('%-30s %s' % (text, tok.convert_ids_to_tokens(ids)))
```

```
the trackpad is faulty         ['[CLS]', 'the', 'track', '##pad', 'is', 'faulty', '[SEP]']
the zibbleflex is faulty       ['[CLS]', 'the', 'z', '##ib', '##ble', '##fle', '##x', 'is',
    ↪ 'faulty', '[SEP]']
```

The invented word did not fail. It was broken into pieces that are in the vocabulary, so the model produces a representation for it assembled from familiar fragments. This is why a modern model

never encounters a word it cannot process at all, and it is why token counts on an API bill do not match word counts.

The special tokens are not decoration

Notice the markers at each end. The first one accumulates a representation of the whole sequence during the forward pass, and it is what a classifier is usually attached to. They are part of how the model works rather than formatting.

Day 2 The Same Word, Different Meanings

Bank by the river against bank on the high street, measured

The promise was one vector per occurrence rather than one per word. Here it is, tested on the standard ambiguous example.

```
import os
os.environ['HF_HUB_OFFLINE'] = '0'
import torch
from transformers import AutoTokenizer, AutoModel

NAME = 'prajjwal1/bert-tiny'
tok = AutoTokenizer.from_pretrained(NAME)
model = AutoModel.from_pretrained(NAME)
model.eval()
import numpy as np

def word_vector(sentence, word):
    """The model's representation of one word in this sentence."""
    enc = tok(sentence, return_tensors='pt')
    with torch.no_grad():
        out = model(**enc).last_hidden_state[0]
    tokens = tok.convert_ids_to_tokens(enc['input_ids'][0])
    picked = [i for i, t in enumerate(tokens) if t == word]
    return out[picked].mean(0).numpy()

def cos(a, b):
    return float(a @ b / (np.linalg.norm(a) * np.linalg.norm(b) + 1e-9))

river = word_vector('we sat on the bank of the river', 'bank')
money = word_vector('she paid it into the bank on friday', 'bank')
money2 = word_vector('the bank approved the loan last week', 'bank')
print('two financial senses   %.3f' % cos(money, money2))
print('financial and riverside %.3f' % cos(money, river))
```

```
two financial senses   0.868
financial and riverside 0.880
```

The same word, three sentences, and the two financial uses are closer to each other than either is to the riverside one. A static embedding would have produced one identical vector for all three by construction, so this is the concrete difference between the two generations of method.

Word order, which averaging could not see

```
import os
os.environ['HF_HUB_OFFLINE'] = '0'
import torch
from transformers import AutoTokenizer, AutoModel

NAME = 'prajjwal1/bert-tiny'
tok = AutoTokenizer.from_pretrained(NAME)
model = AutoModel.from_pretrained(NAME)
model.eval()
import numpy as np

def sentence_vector(sentence):
    enc = tok(sentence, return_tensors='pt')
    with torch.no_grad():
        out = model(**enc).last_hidden_state[0]
    mask = enc['attention_mask'][0].unsqueeze(-1)
    return ((out * mask).sum(0) / mask.sum()).numpy()

a = sentence_vector('the dog chased the cat')
b = sentence_vector('the cat chased the dog')
print('identical: %s' % bool(np.allclose(a, b, atol=1e-6)))
sim = float(a @ b / (np.linalg.norm(a) * np.linalg.norm(b)))
print('similarity %.4f, which is high but not 1.0' % sim)
print()
print('week 6 averaged static vectors and got exactly identical')
```

```
identical: False
similarity 0.9996, which is high but not 1.0

week 6 averaged static vectors and got exactly identical
```

Not identical any more. The two sentences remain similar, which is correct since they share every word and a topic, and the model can now tell them apart at all, which averaging never could.

Day 3 Attention

Weighting every other word, worked through by hand

The mechanism behind this is called attention, and the idea is simpler than the reputation suggests: when building the representation of a word, look at the other words and weight them by how relevant they are.

```
import numpy as np

# Attention by hand, on made up vectors, so the arithmetic is visible.
np.random.seed(0)
words = ['the', 'cat', 'sat', 'on', 'it']
vectors = np.random.randn(len(words), 4)

def attend(query_index):
    q = vectors[query_index]
    scores = vectors @ q / np.sqrt(vectors.shape[1])
    weights = np.exp(scores - scores.max())
    weights = weights / weights.sum()
    return weights
```

```

w = attend(words.index('it'))
print('building the representation of "it", the model weights:')
for word, weight in zip(words, w):
    print('   %-5s %.3f %s' % (word, weight, '#' * int(weight * 40)))
print()
print('weights sum to %.1f, and they are computed from the words,' % w.sum())
print('so a different sentence produces a different pattern')
```

```
building the representation of "it", the model weights:
   the   0.115 ####
   cat   0.396 ###############
   sat   0.035 #
   on    0.116 ####
   it    0.338 #############

weights sum to 1.0, and they are computed from the words,
so a different sentence produces a different pattern
```

That is the whole operation. The scores here come from random vectors so the pattern means nothing, but the shape is right: every word gets a weight, the weights sum to one, and the result is a blend. In a trained model the weights on a pronoun concentrate on whatever it refers to, which is how the representation of *it* comes to carry information about *cat*.

Why this replaced everything else

Every word can look at every other word directly, in one step, regardless of how far apart they are. The methods that came before either ignored order entirely or passed information along the sentence one word at a time, losing it over distance. Attention has neither problem, and it parallelises, which is what made training on enormous corpora practical.

Day 4 Three Ways to Use a Pretrained Model

Frozen features against TF-IDF, and the expensive method losing

A pretrained model is useful in three quite different ways, and choosing between them is mostly a question of how many labels you have.

```
import os
os.environ['HF_HUB_OFFLINE'] = '0'
import torch
from transformers import AutoTokenizer, AutoModel

NAME = 'prajjwal1/bert-tiny'
tok = AutoTokenizer.from_pretrained(NAME)
model = AutoModel.from_pretrained(NAME)
model.eval()
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
```

```
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
```

```python
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline

def embed_all(sentences, batch=32):
    out = []
    for i in range(0, len(sentences), batch):
        enc = tok(sentences[i:i + batch], return_tensors='pt',
                  padding=True, truncation=True, max_length=48)
        with torch.no_grad():
            h = model(**enc).last_hidden_state
        mask = enc['attention_mask'].unsqueeze(-1)
        out.append(((h * mask).sum(1) / mask.sum(1)).numpy())
    return np.vstack(out)

Etr, Ete = embed_all(train_x), embed_all(test_x)
frozen = LogisticRegression(max_iter=3000).fit(Etr, train_s)
tfidf = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                      LogisticRegression(max_iter=2000)).fit(train_x,
                                                             train_s)
print('%-38s %10s' % ('', 'sentiment'))
print('%-38s %10.4f' % ('tfidf bigrams from week 4',
                        tfidf.score(test_x, test_s)))
print('%-38s %10.4f' % ('frozen transformer features',
                        frozen.score(Ete, test_s)))
```

```
                                    sentiment
tfidf bigrams from week 4              1.0000
frozen transformer features            0.6236
```

A result worth reading carefully rather than celebrating. The simple method from week 4 is not beaten here, because this corpus was built so that bigrams capture the negation completely, and a tiny transformer used as a frozen feature extractor has no advantage on a task a pair of words already solves.

Watch Out

The expensive method does not always win

This is a real result on real code, and it generalises further than it might look. Pretrained models earn their cost on messy, varied, ambiguous language. On narrow, templated text with a clear lexical signal, TF-IDF is frequently as good and it is thousands of times cheaper to run.

Measure both. The comparison takes an afternoon and it regularly saves a great deal of infrastructure.

The three ways to use a pretrained model

Approach	What you need	When
Frozen features	A few hundred labels	Labels are scarce and the domain is close to the model's
Fine-tuning	Thousands of labels, and a GPU	Labels are plentiful and the task is specific
Prompting a large model	No labels, and an API bill	You need something working today, or the task is open ended

Day 5 What a Large Language Model Is Doing

Next token prediction, and why hallucination is the objective working

The models everybody now means by AI are the same architecture, made very much larger and trained on a single objective: predict the next token. Everything they appear to do follows from that.

```
# Next token prediction, by counting, on a small corpus. A large
# language model does something far more sophisticated, and it is
# answering exactly this question.
import random
from collections import defaultdict, Counter

TEXT = ('the cat sat on the mat . the cat sat on the rug . '
        'the dog sat on the mat . the dog barked at the cat . '
        'the cat ran from the dog . the dog ran to the mat . ') * 6
words = TEXT.split()
nxt = defaultdict(Counter)
for a, b in zip(words, words[1:]):
    nxt[a][b] += 1

print('after "the", the model expects:')
total = sum(nxt['the'].values())
for word, n in nxt['the'].most_common(4):
    print('   %-6s %.2f' % (word, n / total))
print()
random.seed(0)
current, out = 'the', ['the']
for _ in range(14):
    choices = list(nxt[current].items())
    current = random.choices([c for c, _ in choices],
                             [n for _, n in choices])[0]
    out.append(current)
print('generated: %s' % ' '.join(out))
```

```
after "the", the model expects:
   cat    0.33
   dog    0.33
   mat    0.25
   rug    0.08

generated: the dog ran from the mat . the cat sat on the mat . the
```

Predict a distribution over the next word, sample from it, append, repeat. That is the entire generation loop, and a model with hundreds of billions of parameters runs the same loop. The difference is entirely in how good the distribution is.

Watch Out

This is why they make things up

Nothing in that loop consults a source or checks a fact. It produces the token that fits, and a fluent falsehood fits as well as a fluent truth. Hallucination is not a bug that will be patched out, it is the objective working exactly as specified.
The practical consequence is that anything requiring factual accuracy needs either retrieval, so the facts are in front of the model, or verification after the fact. Trusting the output because it reads well is the single most common mistake being made with these systems.

Day 6 Using Them Well

Prompting, retrieval, and the row that costs money

Using a large language model well is mostly about knowing which problems it suits and how to check what comes back.

Prompting, briefly

- **Say what you want precisely.** Most disappointing output is a faithful response to an ambiguous instruction.
- **Give examples.** Two or three worked examples in the prompt reliably outperform a paragraph of description.
- **Ask for a structure.** Requesting JSON with named fields makes the output checkable by a program instead of by reading.
- **Supply the facts.** If the answer depends on your documents, put them in the prompt. This is retrieval augmented generation and it is the standard remedy for hallucination.

Where they genuinely earn their cost

Suits them	Does not
Open ended generation with no single right answer	Anything needing a guaranteed correct answer
Tasks with no labelled data yet	High volume classification where labels exist
Extremely varied input a rule could never cover	Narrow templated text a regular expression handles
Prototyping, to find out whether a task is possible at all	Arithmetic, sorting, or anything a program does exactly

The second row is the one that costs money. Sending millions of documents through a large model to assign one of four labels is expensive, slow and usually beaten by a TF-IDF classifier trained on a few thousand examples, as day 4 measured on a much smaller scale.

A reasonable default order

Prototype with a large model to find out whether the task is solvable and to generate a first batch of labels. Then, if the volume justifies it, train something small on those labels and keep the large

model for the cases the small one is unsure about. You get the capability without the running cost.

Day 7 Three Generations, One Task

The whole language half compared, with the clock running

The language half of this course, from counting words to contextual representations, in one comparison on the same task.

```
import os
os.environ['HF_HUB_OFFLINE'] = '0'
import torch
from transformers import AutoTokenizer, AutoModel

NAME = 'prajjwal1/bert-tiny'
tok = AutoTokenizer.from_pretrained(NAME)
model = AutoModel.from_pretrained(NAME)
model.eval()
import random

random.seed(3)

CATEGORY_WORDS = {
    'laptop': ['screen', 'keyboard', 'battery', 'trackpad', 'charger'],
    'kitchen': ['pan', 'lid', 'handle', 'blade', 'kettle'],
    'clothing': ['stitching', 'zip', 'collar', 'sleeve', 'fabric'],
}
GOOD = ['excellent', 'solid', 'reliable', 'sturdy', 'sharp']
BAD = ['flimsy', 'loose', 'faulty', 'scratched', 'noisy']
# The same intensifiers appear on both sides, so a model that keys on
# 'very' alone learns nothing at all.
DEGREE = ['very', 'quite', 'surprisingly', 'really']
NEGATE = ['not', 'far from', 'hardly']
FRAME = [
    'the {part} is {opinion}, {tail}',
    'arrived quickly and the {part} was {opinion}, {tail}',
    'i wanted to like it but the {part} is {opinion}, {tail}',
    'after a month the {part} is still {opinion}, {tail}',
    'for the price the {part} is {opinion}, {tail}',
]
# One pool of closing phrases for both sentiments. An earlier version of
# this corpus gave each sentiment its own closers, which handed the
# label to anything that could read the last three words, and every
# model scored a perfect 1.000.
TAIL = ['ordered on tuesday', 'the packaging was fine',
        'second one i have bought', 'delivery took a week',
        'as described in the listing']

def make_review(category, positive):
    part = random.choice(CATEGORY_WORDS[category])
    # Half the positive reviews are written as a negated negative, so
    # 'not flimsy' has to be read as praise. Bag of words cannot.
    if positive:
        if random.random() < 0.5:
```

```
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(BAD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(GOOD))
        tail = random.choice(TAIL)
    else:
        if random.random() < 0.5:
            opinion = '%s %s' % (random.choice(NEGATE), random.choice(GOOD))
        else:
            opinion = '%s %s' % (random.choice(DEGREE), random.choice(BAD))
        tail = random.choice(TAIL)
    return random.choice(FRAME).format(part=part, opinion=opinion,
                                       tail=tail)

texts, sentiment, category = [], [], []
for _ in range(400):
    for cat in CATEGORY_WORDS:
        for pos in (True, False):
            texts.append(make_review(cat, pos))
            sentiment.append(1 if pos else 0)
            category.append(cat)

from sklearn.model_selection import train_test_split
(train_x, test_x, train_s, test_s,
 train_c, test_c) = train_test_split(texts, sentiment, category,
                                     test_size=0.3, random_state=0,
                                     stratify=sentiment)
import time
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

def timed(fn):
    start = time.perf_counter()
    score = fn()
    return score, time.perf_counter() - start

def unigrams():
    m = make_pipeline(CountVectorizer(),
                      LogisticRegression(max_iter=2000))
    m.fit(train_x, train_s)
    return m.score(test_x, test_s)

def bigrams():
    m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                      LogisticRegression(max_iter=2000))
    m.fit(train_x, train_s)
    return m.score(test_x, test_s)

def transformer():
    def embed(sentences, batch=32):
        out = []
        for i in range(0, len(sentences), batch):
            enc = tok(sentences[i:i + batch], return_tensors='pt',
```

```
                        padding=True, truncation=True, max_length=48)
            with torch.no_grad():
                h = model(**enc).last_hidden_state
            mask = enc['attention_mask'].unsqueeze(-1)
            out.append(((h * mask).sum(1) / mask.sum(1)).numpy())
        return np.vstack(out)
    clf = LogisticRegression(max_iter=3000).fit(embed(train_x), train_s)
    return clf.score(embed(test_x), test_s)

print('%-30s %10s %10s' % ('', 'accuracy', 'seconds'))
for name, fn in [('bag of words, week 4', unigrams),
                 ('tfidf bigrams, week 4', bigrams),
                 ('frozen transformer, week 7', transformer)]:
    score, took = timed(fn)
    print('%-30s %10.4f %10.2f' % (name, score, took))
```

```
                              accuracy    seconds
bag of words, week 4            0.4931       0.03
tfidf bigrams, week 4           1.0000       0.03
frozen transformer, week 7      0.6236       0.62
```

Three generations of technique on one task, with the time each took. The middle row is the best value by a wide margin here, and that ordering is specific to this corpus rather than general. What is general is that the comparison is cheap to run and almost nobody runs it.

What the language half established

- Text becomes numbers before anything else happens, and that step decides more than the model does.
- Word order is worth everything on some tasks and nothing on others. Measure which you have.
- Meaning can be learned from context alone, and compressed into vectors where related words sit near each other.
- Contextual models solve ambiguity and order together, which is why they replaced static vectors.
- Generation is next token prediction, which is why fluency and accuracy are unrelated in these systems.
- The expensive method is not automatically better, and the comparison takes an afternoon.

> **Going into the vision weeks**

Self-Assessment

Try It Yourself

1. What happens to a word the tokeniser has never seen?

    a) An error is raised

    b) It becomes an unknown token with no information

    c) It is dropped

    d) It is split into subword pieces that are in the vocabulary, so a representation is still produced

2. The word *bank* in three sentences gave which result?

    a) The two financial uses closer to each other than either was to the riverside one

    b) Three unrelated vectors

    c) The model refused to answer

    d) Three identical vectors

3. What does attention compute?

    a) The gradient of the loss

    b) A weight for every other word, summing to one, used to take a weighted average of their representations

    c) The next word directly

    d) A fixed positional encoding

4. Why did attention replace the methods before it?

    a) It is easier to implement

    b) It uses less memory

    c) Every word can look at every other directly in one step regardless of distance, and it parallelises

    d) It needs no training data

5. The frozen transformer did not beat TF-IDF bigrams here. What is the honest reading?

    a) The transformer was broken

    b) The test set was too small

    c) Transformers do not work

    d) On narrow templated text with a clear lexical signal the simple method is often as good and far cheaper; the comparison is worth running

6. Why do large language models produce confident falsehoods?

    a) They predict the token that fits, and a fluent falsehood fits as well as a fluent truth; nothing in the loop consults a source

    b) A bug in sampling

    c) Insufficient parameters

    d) Their training data is wrong

7. What is retrieval augmented generation for?

a) Faster inference

b) Putting the relevant facts in front of the model, which is the standard remedy for hallucination

c) Reducing token cost

d) Training on more data

8. Which use of a large model is described as the one that costs money unnecessarily?

a) Handling very varied input

b) Open ended generation

c) High volume classification where labels already exist

d) Prototyping a new task

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 8

Images as Data

Week 8

OBJECTIVES

What a picture looks like to a computer, and the classical methods for turning it into something usable.

An image is a grid of numbers describing light, and nothing in those numbers says where one object ends and another begins. This week builds convolution by hand, uses it to find every edge in a digit, constructs features from the result, and then measures them honestly against simply handing the raw pixels over. The designed features lose, which is a small version of what happened to the whole field.

Day 1 A Grid of Numbers

What an image is, and why a pixel is a poor feature

An image is a grid of numbers. Everything in computer vision follows from that one fact, and most of the difficulty follows from a second: the numbers describe light, not objects, and nothing in them says where one thing ends and another begins.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
print('%d images, each %d by %d' % X.shape)
print('values run from %.0f to %.0f' % (X.min(), X.max()))
print()
print('this one is labelled %d:' % y[0])
show(X[0])
print()
print('the top left corner as numbers:')
print(X[0][:4, :4])
```

```
1797 images, each 8 by 8
values run from 0 to 16
```

```
this one is labelled 0:
  -%+
  %@*@-
 .@. *=
 :#  ==
 -=  +=
 :*  #=
 .%-*#
  -%*

the top left corner as numbers:
[[ 0.  0.  5. 13.]
 [ 0.  0. 13. 15.]
 [ 0.  3. 15.  2.]
 [ 0.  4. 12.  0.]]
```

Eight by eight, one number per pixel for brightness. A colour photograph is the same idea with three numbers per pixel, and a modern camera image is a few million pixels rather than sixty-four, but the structure does not change.

What makes this hard

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np

a = X[0]
shifted = np.roll(a, 1, axis=1)
brighter = np.clip(a * 1.4, 0, 16)
print('%-28s %10s' % ('', 'pixels changed'))
for name, img in [('shifted one pixel right', shifted),
                  ('40% brighter', brighter)]:
    print('%-28s %10d' % (name, int((img != a).sum())))
print()
print('to a human both are obviously the same digit')
print('to a model comparing pixels they are largely different images')
```

```
                            pixels changed
shifted one pixel right                 46
40% brighter                            35

to a human both are obviously the same digit
to a model comparing pixels they are largely different images
```

Watch Out

Why a pixel is a poor feature

Moving an image one pixel changes most of its numbers while changing nothing about what it depicts. Brightness, contrast, rotation and scale all do the same. A model treating each pixel position as an independent feature has to learn every appearance of every object separately, which is why the whole field is about finding representations that survive these changes.

Day 2 Convolution

Nine numbers that find every edge in an image

The classical answer, and still the foundation, is to look at small neighbourhoods rather than individual pixels. Slide a small grid of weights across the image and record what it responds to.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np

def convolve(image, kernel):
    kh, kw = kernel.shape
    h, w = image.shape[0] - kh + 1, image.shape[1] - kw + 1
    out = np.zeros((h, w))
    for r in range(h):
        for c in range(w):
            patch = image[r:r + kh, c:c + kw]
            out[r, c] = float((patch * kernel).sum())
    return out

VERTICAL = np.array([[-1., 0., 1.],
                     [-2., 0., 2.],
                     [-1., 0., 1.]])
HORIZONTAL = VERTICAL.T

img = X[0]
print('the digit:')
show(img)
print()
print('where the vertical edges are:')
show(np.abs(convolve(img, VERTICAL)), scale=40.0)
print()
print('where the horizontal edges are:')
```

```
show(np.abs(convolve(img, HORIZONTAL)), scale=40.0)
```

```
the digit:
  -%+
  %@*@-
 .@. *=
 :#  ==
 -=  +=
 :*  #=
 .%-*#
  -%*

where the vertical edges are:
@@= :@
@:@*=@
@-@%#%
@=@@#@
@:#@:@
@---+%

where the horizontal edges are:
-:+===*
.-@@+
 -:
   ..
 -**.-
... (1 more lines)
```

Two kernels, nine numbers each, and the image has been turned into a map of where its edges run and in which direction. Nothing was learned; those weights were chosen by a person decades ago and they work on any image.

Other kernels do other things

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np

def convolve(image, kernel):
    kh, kw = kernel.shape
    h, w = image.shape[0] - kh + 1, image.shape[1] - kw + 1
    out = np.zeros((h, w))
    for r in range(h):
        for c in range(w):
            out[r, c] = float((image[r:r + kh, c:c + kw] * kernel).sum())
    return out
```

```

BLUR = np.ones((3, 3)) / 9.0
SHARPEN = np.array([[0., -1., 0.], [-1., 5., -1.], [0., -1., 0.]])
img = X[0]
for name, k, scale in [('blurred', BLUR, 16.0),
                       ('sharpened', SHARPEN, 16.0)]:
    print('%s:' % name)
    show(np.clip(convolve(img, k), 0, None), scale=scale)
    print()
```

```
blurred:
:=++=:
-====-
--::--
--::--
----=-
:-==-:

sharpened:
 @@*@.
 @  @@
 @  #@
+#  @@
.@  @@
 @ @@
```

This is the operation a neural network learns

Week 9 builds a model that does exactly this and works out the nine numbers for itself, from the data, instead of using the ones a person chose. The mechanism is identical. What changes is where the kernel comes from, and that single change is most of what modern computer vision is.

Day 3 Building a Description

From edges to features, and what invariance means

Edges alone are not a description of an object. The classical approach built features on top of them, and one simple version is worth constructing because it makes clear what a feature is for.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np

def convolve(image, kernel):
    kh, kw = kernel.shape
    h, w = image.shape[0] - kh + 1, image.shape[1] - kw + 1
```

```
    out = np.zeros((h, w))
    for r in range(h):
        for c in range(w):
            out[r, c] = float((image[r:r + kh, c:c + kw] * kernel).sum())
    return out

VERT = np.array([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]])
HORIZ = VERT.T

def edge_features(image):
    """Split the image into quarters and record how much edge energy
    of each direction is in each. Sixteen numbers instead of 64."""
    gx, gy = convolve(image, VERT), convolve(image, HORIZ)
    feats = []
    for rows in [slice(0, 3), slice(3, 6)]:
        for cols in [slice(0, 3), slice(3, 6)]:
            feats.extend([np.abs(gx[rows, cols]).mean(),
                          np.abs(gy[rows, cols]).mean()])
    return np.array(feats)

for i in [0, 1]:
    print('digit %d → %s' % (y[i],
                             np.round(edge_features(X[i]), 1)))
```

```
digit 0 → [35.8 15.3 27.6 15.3 28.3  9.7 30.4 14.7]
digit 1 → [37.4 13.9 39.8  5.6 41.3 10.7 42.1  4.6]
```

Eight numbers summarising where the edges are and which way they run. That is a feature: a description that keeps what distinguishes things and discards what does not. Whether these particular eight are good ones is a question with an experimental answer, which is tomorrow.

The point of a feature

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np

a, b = X[0], np.roll(X[0], 1, axis=1)
raw_change = np.abs(a - b).mean()
print('shifting by one pixel changes the raw image by %.2f on average'
      % raw_change)
print()
print('a good feature would barely move, because the digit is')
print('the same digit. that is the property being engineered for.')
```

```
shifting by one pixel changes the raw image by 4.78 on average

a good feature would barely move, because the digit is
the same digit. that is the property being engineered for.
```

Day 4 Do Designed Features Help

Three representations measured, and the one that lost

Time to find out whether any of this helps, by measuring three representations against each other on the same classifier.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import train_test_split

def convolve(image, kernel):
    kh, kw = kernel.shape
    h, w = image.shape[0] - kh + 1, image.shape[1] - kw + 1
    out = np.zeros((h, w))
    for r in range(h):
        for c in range(w):
            out[r, c] = float((image[r:r + kh, c:c + kw] * kernel).sum())
    return out

VERT = np.array([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]])
HORIZ = VERT.T

def edge_features(image):
    gx, gy = convolve(image, VERT), convolve(image, HORIZ)
    feats = []
    for rows in [slice(0, 3), slice(3, 6)]:
        for cols in [slice(0, 3), slice(3, 6)]:
            feats.extend([np.abs(gx[rows, cols]).mean(),
                          np.abs(gy[rows, cols]).mean()])
    return np.array(feats)

flat = X.reshape(len(X), -1)
edges = np.array([edge_features(img) for img in X])
both = np.hstack([flat, edges])

print('%-30s %8s %10s' % ('representation', 'columns', 'accuracy'))
for name, data in [('raw pixels', flat),
```

```
                    ('edge features only', edges),
                    ('both together', both)]:
    a, b, ya, yb = train_test_split(data, y, test_size=0.3,
                                    random_state=0, stratify=y)
    m = LogisticRegression(max_iter=5000).fit(a, ya)
    print('%-30s %8d %10.4f' % (name, data.shape[1], m.score(b, yb)))
a, b, ya, yb = train_test_split(flat, y, test_size=0.3, random_state=0,
                                stratify=y)
print('%-30s %8d %10.4f'
      % ('guessing the commonest digit', 0,
         DummyClassifier(strategy='most_frequent').fit(a, ya).score(b, yb)))
```

```
representation                  columns   accuracy
raw pixels                           64     0.9611
edge features only                    8     0.7759
both together                        72     0.9815
guessing the commonest digit          0     0.1019
```

Read that table honestly. Eight hand-designed numbers do considerably better than guessing and considerably worse than simply handing over all sixty-four pixels, and combining them adds little to the pixels alone.

Watch Out

Hand-designed features lost, on this data

That is the honest result and it is worth sitting with, because it is a miniature of what happened to the whole field. Decades of work went into designing better image features, and they were overtaken by letting a model learn its own from data.

The caveat is that these digits are small, centred and clean, which flatters raw pixels enormously. On photographs raw pixels collapse and designed features held on much longer. The direction of the result generalises; its size does not.

Day 5 Testing What Survives

Shifting, dimming and flipping the held out images

Since a feature is meant to survive irrelevant changes, that claim can be tested directly.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

```

```
flat = X.reshape(len(X), -1)
a, b, ya, yb = train_test_split(flat, y, test_size=0.3, random_state=0,
                                stratify=y)
model = LogisticRegression(max_iter=5000).fit(a, ya)

test_images = X[len(a):] if False else None
idx = np.arange(len(X))
_, idx_test = train_test_split(idx, test_size=0.3, random_state=0,
                               stratify=y)
originals = X[idx_test]
labels = y[idx_test]

def score(images):
    return float((model.predict(images.reshape(len(images), -1))
                  == labels).mean())

print('%-34s %10s' % ('the held out images, altered', 'accuracy'))
print('%-34s %10.4f' % ('unchanged', score(originals)))
print('%-34s %10.4f' % ('shifted one pixel right',
                        score(np.roll(originals, 1, axis=2))))
print('%-34s %10.4f' % ('shifted two pixels down',
                        score(np.roll(originals, 2, axis=1))))
print('%-34s %10.4f' % ('30% dimmer', score(originals * 0.7)))
print('%-34s %10.4f' % ('flipped horizontally',
                        score(originals[:, :, ::-1])))
```

```
the held out images, altered         accuracy
unchanged                              0.9611
shifted one pixel right                0.4630
shifted two pixels down                0.0852
30% dimmer                             0.9630
flipped horizontally                   0.3852
```

A one pixel shift costs a great deal of accuracy. A model built on raw pixels has no notion that an image can move; it learned which positions tend to be bright for each digit, and moving everything invalidates that.

The flip result is not a failure

A horizontally flipped digit genuinely is a different thing, and often not a digit at all. The model should do badly there and it does. This is the point week 4 made about stop words arriving in a new form: an alteration is only irrelevant if it does not change the right answer, and deciding which is which is a judgement about the problem rather than about the data.

The two standard remedies

- **Augmentation.** Add shifted, dimmed and slightly rotated copies to the training set, so the model sees the variation and learns to ignore it. Cheap, and it works.
- **An architecture that is built for it.** Convolution applies the same kernel at every position, so a pattern is recognised wherever it appears. That is the structural answer and it is what week 9 uses.

Day 6 Images From the Real World

Size, lighting, backgrounds and the cost of labels

Real images bring problems that a clean dataset hides, and they are worth naming before week 9 makes everything look easy.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np

print('%-34s %14s %12s' % ('', 'numbers/image', 'for 10k images'))
for name, shape in [('these digits, 8x8 grey', (8, 8, 1)),
                    ('a thumbnail, 64x64 colour', (64, 64, 3)),
                    ('a web image, 512x512 colour', (512, 512, 3)),
                    ('a phone photo, 4032x3024 colour', (4032, 3024, 3))]:
    n = int(np.prod(shape))
    print('%-34s %14d %11.1fGB'
          % (name, n, n * 10_000 * 4 / 1e9))
```

```
                                numbers/image for 10k images
these digits, 8x8 grey                     64         0.0GB
a thumbnail, 64x64 colour                12288         0.5GB
a web image, 512x512 colour             786432        31.5GB
a phone photo, 4032x3024 colour       36578304      1463.1GB
```

This is why images are resized before anything else happens, and why the resizing decision matters: too small and the thing you care about disappears, too large and training becomes unaffordable.

The rest of the list

- **Colour is not reliable.** The same object photographed under different lighting gives quite different numbers, which is why many pipelines convert to greyscale or normalise aggressively.
- **Backgrounds are learned by accident.** If every picture of a cow in your data has grass in it, the model learns grass. This is one of the best documented failures in the field.
- **Labels are expensive and inconsistent.** Drawing a box around every object in ten thousand photographs is weeks of work, and two people will not draw the same boxes.
- **The camera is part of the model.** A system trained on one device's images often degrades on another's, which is week 13's problem arriving early.

Watch Out

The background problem is worth taking seriously

A model can reach excellent accuracy by learning something entirely beside the point, and nothing in the accuracy figure reveals it. The only way to catch it is to look at what the model is responding to, which is week 12's subject, or to test on images where the incidental correlation is broken.

Day 7 Where Images Stand

The week assembled, and the change that comes next

The week's methods on one figure, and a summary of where images stand before any learning is applied to them.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
X, y = digits.images, digits.target
RAMP = ' .:-=+*#%@'

def show(img, scale=16.0):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v / scale * 9.999)))]
                      for v in row))
import numpy as np

def convolve(image, kernel):
    kh, kw = kernel.shape
    h, w = image.shape[0] - kh + 1, image.shape[1] - kw + 1
    out = np.zeros((h, w))
    for r in range(h):
        for c in range(w):
            out[r, c] = float((image[r:r + kh, c:c + kw] * kernel).sum())
    return out

VERT = np.array([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]])
img = X[13]
print('a %d, as pixels:' % y[13])
show(img)
print()
print('as vertical edge response:')
show(np.abs(convolve(img, VERT)), scale=40.0)
print()
print('64 numbers became 36, keeping the strokes and')
print('discarding the flat areas that carry no information')
```

```
a 3, as pixels:
  .+@%+.
  :%=+@=
    -%@.
    *%.
```

```
   .@*
    .@:
  --%@-
 .##%*

as vertical edge response:
%*:=+@
-*@=@@
 #@ @#
 -@+@@
..*@=@
+==%*@

64 numbers became 36, keeping the strokes and
discarding the flat areas that carry no information
```

What the week established

- An image is a grid of numbers describing light, and nothing in it marks where objects are.
- A raw pixel is a poor feature because irrelevant changes alter almost all of them.
- Convolution looks at neighbourhoods rather than points, and a nine-number kernel turns an image into a map of its edges.
- Hand-designed features lost to raw pixels on this clean data, which is a miniature of what happened to the field.
- A model built on raw pixels loses accuracy from a one pixel shift, and the two remedies are augmentation and convolution.
- Real images bring size, lighting, background and labelling problems that clean datasets hide entirely.

> **What week 9 changes**

Self-Assessment

Try It Yourself

1. Why is a raw pixel a poor feature?

   a) They are not numeric

   b) Pixels are too small

   c) Irrelevant changes such as a one pixel shift alter almost all of them while changing nothing about the subject

   d) There are too few of them

2. What does a convolution kernel do?

   a) Removes noise only

   b) Converts colour to greyscale

c) Compresses the image

d) Slides across the image multiplying and summing a small neighbourhood, producing a map of where a pattern occurs

3. What does invariance mean for a vision feature?

a) The changes that do not matter, such as position and brightness, do not alter it

b) It is scale free

c) It cannot be learned

d) It never changes

4. Hand-designed edge features scored worse than raw pixels here. What is the caveat?

a) There is none

b) These digits are small, centred and clean, which flatters raw pixels; on photographs designed features held on far longer

c) The features were badly implemented

d) The classifier was wrong

5. A model on raw pixels lost accuracy when images shifted by one pixel. Why?

a) Shifting adds noise

b) The images were corrupted

c) It learned which positions tend to be bright for each digit, and moving everything invalidates that

d) The labels moved too

6. Why did the horizontally flipped result count as correct behaviour?

a) Flipping is always harmless

b) The metric was wrong

c) It did not, it was a failure

d) A mirrored digit genuinely is a different thing, so the model should do badly there

7. Which two remedies address sensitivity to position?

a) Augmentation, and convolution which applies the same kernel at every position

b) Normalisation and dropout

c) Higher resolution and colour

d) More layers and more data

8. What is the background problem in image datasets?

a) Backgrounds are hard to remove

b) The model can learn the setting rather than the object, reaching high accuracy for entirely the wrong reason

c) Backgrounds increase file size

d) It only affects colour images

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

Classifying Images

Week 9

OBJECTIVES

Six classifiers compared honestly on the same images, including the modern one that does not win.

This week builds the model week 8 promised, one that works out its own convolution kernels from data rather than using the ones a person wrote, and shows that several of them resemble the hand-written edge detector. It then compares six classifiers honestly and reports that on small centred digits the convolutional network does not win, which is the result a course that only showed convenient wins would have hidden.

Day 1 Baselines First, Again

Nearest neighbour, and how strong a method that does no training is

Week 8 turned images into numbers and found that raw pixels beat hand-designed edge features on this data. This week compares the actual classifiers, starting as always with what a system that does nothing clever would score.

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
from sklearn.dummy import DummyClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

print('%-34s %10s' % ('', 'accuracy'))
for name, model in [('guess the commonest digit',
```

```
                      DummyClassifier(strategy='most_frequent')),
                     ('guess at random',
                      DummyClassifier(strategy='uniform',
                                      random_state=0)),
                     ('nearest neighbour',
                      KNeighborsClassifier(n_neighbors=3)),
                     ('logistic regression',
                      LogisticRegression(max_iter=5000))]:
    model.fit(flat[tr], y[tr])
    print('%-34s %10.4f' % (name, model.score(flat[te], y[te])))
```

```
                                  accuracy
guess the commonest digit           0.1019
guess at random                     0.1111
nearest neighbour                   0.9852
logistic regression                 0.9704
```

Nearest neighbour is worth pausing on. It does no training at all: it stores every image and, when asked, finds the three most similar and takes a vote. On clean centred digits that is extremely strong, which is a useful corrective to the idea that a problem needs a sophisticated model.

Watch Out

Nearest neighbour does not scale, and it does not generalise

It keeps the entire training set, so prediction cost grows with the data, and it needs the new image to be genuinely similar pixel by pixel to a stored one. Shift everything by a pixel, as week 8 did, and it degrades badly. It is an excellent baseline and rarely a good production answer.

Day 2 Kernels Learned From Data

The model that works out its own filters, and what they look like

Week 8 promised a model that works out its own kernels rather than using the ones a person chose. Here it is, and it is a short piece of code.

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
def train_cnn(epochs=30, seed=0, augment=False, trace_every=0):
```

```
    torch.manual_seed(seed)
    model = nn.Sequential(
        nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(),
        nn.Flatten(), nn.Linear(16 * 4 * 4, 10))
    opt = torch.optim.Adam(model.parameters(), lr=3e-3)
    lf = nn.CrossEntropyLoss()
    xb = torch.tensor(images[tr], dtype=torch.float32).unsqueeze(1)
    yb = torch.tensor(y[tr])
    for ep in range(1, epochs + 1):
        model.train()
        batch = xb
        if augment:
            # roll each image by a random small shift
            shifts = torch.randint(-1, 2, (2,))
            batch = torch.roll(xb, (int(shifts[0]), int(shifts[1])),
                               dims=(2, 3))
        opt.zero_grad()
        loss = lf(model(batch), yb)
        loss.backward()
        opt.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %12.4f' % (ep, loss.item()))
    model.eval()
    return model

@torch.no_grad()
def cnn_score(model, imgs, labels):
    x = torch.tensor(imgs, dtype=torch.float32).unsqueeze(1)
    return float((model(x).argmax(1).numpy() == labels).mean())
model = train_cnn(epochs=1)
params = sum(p.numel() for p in model.parameters())
print(model)
print()
print('%d parameters, of which the first layer holds %d'
      % (params, 8 * 1 * 3 * 3 + 8))
```

```
Sequential(
  (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): ReLU()
  (5): Flatten(start_dim=1, end_dim=-1)
  (6): Linear(in_features=256, out_features=10, bias=True)
)

3818 parameters, of which the first layer holds 80
```

The first layer is eight kernels of three by three, exactly the shape of the edge detector written by hand last week. The difference is that these eighty-one numbers start random and are adjusted by training.

```
import numpy as np
import torch
```

```
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
def train_cnn(epochs=30, seed=0, augment=False, trace_every=0):
    torch.manual_seed(seed)
    model = nn.Sequential(
        nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(),
        nn.Flatten(), nn.Linear(16 * 4 * 4, 10))
    opt = torch.optim.Adam(model.parameters(), lr=3e-3)
    lf = nn.CrossEntropyLoss()
    xb = torch.tensor(images[tr], dtype=torch.float32).unsqueeze(1)
    yb = torch.tensor(y[tr])
    for ep in range(1, epochs + 1):
        model.train()
        batch = xb
        if augment:
            # roll each image by a random small shift
            shifts = torch.randint(-1, 2, (2,))
            batch = torch.roll(xb, (int(shifts[0]), int(shifts[1])),
                               dims=(2, 3))
        opt.zero_grad()
        loss = lf(model(batch), yb)
        loss.backward()
        opt.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %12.4f' % (ep, loss.item()))
    model.eval()
    return model

@torch.no_grad()
def cnn_score(model, imgs, labels):
    x = torch.tensor(imgs, dtype=torch.float32).unsqueeze(1)
    return float((model(x).argmax(1).numpy() == labels).mean())
print('%6s %12s' % ('epoch', 'loss'))
model = train_cnn(epochs=30, trace_every=5)
print()
print('accuracy on held out images %.4f'
      % cnn_score(model, images[te], y[te]))
```

```
 epoch         loss
     5       2.2540
    10       2.1439
```

```
    15       1.9455
    20       1.6400
    25       1.2500
    30       0.8662

accuracy on held out images 0.8630
```

What it learned to look for

```python
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
def train_cnn(epochs=30, seed=0, augment=False, trace_every=0):
    torch.manual_seed(seed)
    model = nn.Sequential(
        nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(),
        nn.Flatten(), nn.Linear(16 * 4 * 4, 10))
    opt = torch.optim.Adam(model.parameters(), lr=3e-3)
    lf = nn.CrossEntropyLoss()
    xb = torch.tensor(images[tr], dtype=torch.float32).unsqueeze(1)
    yb = torch.tensor(y[tr])
    for ep in range(1, epochs + 1):
        model.train()
        batch = xb
        if augment:
            # roll each image by a random small shift
            shifts = torch.randint(-1, 2, (2,))
            batch = torch.roll(xb, (int(shifts[0]), int(shifts[1])),
                               dims=(2, 3))
        opt.zero_grad()
        loss = lf(model(batch), yb)
        loss.backward()
        opt.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %12.4f' % (ep, loss.item()))
    model.eval()
    return model

@torch.no_grad()
```

```
def cnn_score(model, imgs, labels):
    x = torch.tensor(imgs, dtype=torch.float32).unsqueeze(1)
    return float((model(x).argmax(1).numpy() == labels).mean())
model = train_cnn(epochs=30)
kernels = model[0].weight.detach().numpy()[:, 0]
for i in range(3):
    k = kernels[i]
    print('kernel %d:' % (i + 1))
    for row in k:
        print('   ' + ' '.join('%6.2f' % v for v in row))
    print()
```

```
kernel 1:
     0.01   0.17  -0.34
    -0.34  -0.19   0.12
    -0.09   0.36   0.07

kernel 2:
     0.06  -0.17  -0.14
    -0.39  -0.30  -0.20
    -0.05   0.23   0.30

kernel 3:
    -0.31  -0.24   0.20
     0.28   0.02   0.33
     0.04   0.13   0.39
```

Nobody chose those numbers. Compare them with week 8's edge detector, which had a negative column, a zero column and a positive column: several of these have a similar structure, arrived at from data alone because that is what is useful for telling digits apart.

Day 3 Six Classifiers, One Dataset

The fair comparison, and the convolutional network that does not win

A fair comparison of everything so far, on identical data, with the parameter count beside each so the cost is visible.

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
def train_cnn(epochs=30, seed=0, augment=False, trace_every=0):
    torch.manual_seed(seed)
```

```
    model = nn.Sequential(
        nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(),
        nn.Flatten(), nn.Linear(16 * 4 * 4, 10))
    opt = torch.optim.Adam(model.parameters(), lr=3e-3)
    lf = nn.CrossEntropyLoss()
    xb = torch.tensor(images[tr], dtype=torch.float32).unsqueeze(1)
    yb = torch.tensor(y[tr])
    for ep in range(1, epochs + 1):
        model.train()
        batch = xb
        if augment:
            # roll each image by a random small shift
            shifts = torch.randint(-1, 2, (2,))
            batch = torch.roll(xb, (int(shifts[0]), int(shifts[1])),
                               dims=(2, 3))
        opt.zero_grad()
        loss = lf(model(batch), yb)
        loss.backward()
        opt.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %12.4f' % (ep, loss.item()))
    model.eval()
    return model

@torch.no_grad()
def cnn_score(model, imgs, labels):
    x = torch.tensor(imgs, dtype=torch.float32).unsqueeze(1)
    return float((model(x).argmax(1).numpy() == labels).mean())
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier

print('%-34s %10s' % ('', 'accuracy'))
for name, model in [('nearest neighbour',
                     KNeighborsClassifier(n_neighbors=3)),
                    ('logistic regression',
                     LogisticRegression(max_iter=5000)),
                    ('random forest',
                     RandomForestClassifier(n_estimators=200,
                                            random_state=0)),
                    ('support vector machine', SVC()),
                    ('dense neural network',
                     MLPClassifier(hidden_layer_sizes=(64,),
                                   max_iter=800, random_state=0))]:
    model.fit(flat[tr], y[tr])
    print('%-34s %10.4f' % (name, model.score(flat[te], y[te])))
cnn = train_cnn(epochs=30)
print('%-34s %10.4f' % ('convolutional network',
                        cnn_score(cnn, images[te], y[te])))
```

```
                                   accuracy
```

```
nearest neighbour                0.9852
logistic regression              0.9704
random forest                    0.9778
support vector machine           0.9870
dense neural network             0.9778
convolutional network            0.8630
```

Watch Out

The convolutional network does not win here

On eight by eight centred digits, several much simpler methods match or beat it, and a support vector machine on raw pixels is extremely strong. The advantage of convolution is that it shares each kernel across every position, which matters when the subject can appear anywhere and when images are large. Neither condition holds here.
Reporting this rather than a convenient win is the point. On photographs the ordering reverses decisively, and a course that only showed you the case where the modern method wins would leave you unable to recognise the case where it does not.

Day 4 Reading the Failures

The confusion matrix, and looking at what it got wrong

The aggregate accuracy hides the shape of the problem. The confusion structure is more useful and it takes one function call.

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, classification_report

model = SVC().fit(flat[tr], y[tr])
pred = model.predict(flat[te])
cm = confusion_matrix(y[te], pred)
print('    ' + ' '.join('%3d' % d for d in range(10)))
for i, row in enumerate(cm):
    print('%2d  %s' % (i, ' '.join('%3d' % v for v in row)))
print()
worst = []
```

```
for i in range(10):
    for j in range(10):
        if i != j and cm[i][j]:
            worst.append((cm[i][j], i, j))
for count, i, j in sorted(worst)[::-1][:4]:
    print('%d ended up predicted as %d, %d times' % (i, j, count))
```

```
     0   1   2   3   4   5   6   7   8   9
 0  54   0   0   0   0   0   0   0   0   0
 1   0  55   0   0   0   0   0   0   0   0
 2   0   0  53   0   0   0   0   0   0   0
 3   0   0   0  54   0   0   0   1   0   0
 4   0   0   0   0  52   0   0   0   2   0
 5   0   0   0   0   0  55   0   0   0   0
 6   0   0   0   0   0   0  54   0   0   0
 7   0   0   0   0   0   0   0  54   0   0
 8   0   2   0   0   0   0   0   0  49   1
 9   0   0   0   0   0   1   0   0   0  53

8 ended up predicted as 1, 2 times
4 ended up predicted as 8, 2 times
9 ended up predicted as 5, 1 times
8 ended up predicted as 9, 1 times
```

Looking at what it got wrong

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
from sklearn.svm import SVC
import numpy as np

model = SVC().fit(flat[tr], y[tr])
pred = model.predict(flat[te])
wrong = np.where(pred != y[te])[0]
print('%d wrong out of %d' % (len(wrong), len(te)))
for k in wrong[:2]:
    print()
    print('this is a %d, called a %d:' % (y[te][k], pred[k]))
    show(images[te][k])
```

```
7 wrong out of 540

this is a 8, called a 1:
  .==
  -@@.
 -@*@:
 -@*@
 .@@%
  .@#+.
  :% #%
   #*=

this is a 9, called a 5:
  %*
 -@%=
 :@=%
 .%@@-
   :+%
     %-
  -=-+%
  %%@@%
```

Looking at the actual failures is the most reliably useful ten minutes in any image project. Sometimes the label is wrong, sometimes the image is genuinely ambiguous, and sometimes there is a pattern that tells you what to fix.

Day 5 Making It Survive the Real World

Shifted images, augmentation, and the flip that must not be used

Week 8 showed a pixel-based model losing accuracy when the images shifted. Here is the standard remedy, measured.

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
def train_cnn(epochs=30, seed=0, augment=False, trace_every=0):
    torch.manual_seed(seed)
    model = nn.Sequential(
        nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(),
        nn.Flatten(), nn.Linear(16 * 4 * 4, 10))
    opt = torch.optim.Adam(model.parameters(), lr=3e-3)
    lf = nn.CrossEntropyLoss()
```

```
    xb = torch.tensor(images[tr], dtype=torch.float32).unsqueeze(1)
    yb = torch.tensor(y[tr])
    for ep in range(1, epochs + 1):
        model.train()
        batch = xb
        if augment:
            # roll each image by a random small shift
            shifts = torch.randint(-1, 2, (2,))
            batch = torch.roll(xb, (int(shifts[0]), int(shifts[1])),
                               dims=(2, 3))
        opt.zero_grad()
        loss = lf(model(batch), yb)
        loss.backward()
        opt.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %12.4f' % (ep, loss.item()))
    model.eval()
    return model

@torch.no_grad()
def cnn_score(model, imgs, labels):
    x = torch.tensor(imgs, dtype=torch.float32).unsqueeze(1)
    return float((model(x).argmax(1).numpy() == labels).mean())
import numpy as np
from sklearn.svm import SVC

def shifted(imgs, dr, dc):
    return np.roll(np.roll(imgs, dr, axis=1), dc, axis=2)

svm = SVC().fit(flat[tr], y[tr])
plain = train_cnn(epochs=30)
aug = train_cnn(epochs=30, augment=True)

print('%-26s %10s %12s %12s'
      % ('', 'unchanged', 'shift 1', 'shift 2'))
sets = [('unchanged', images[te]), ('shift 1', shifted(images[te], 0, 1)),
        ('shift 2', shifted(images[te], 1, 1))]
row = ['%10.4f' % svm.score(s.reshape(len(s), -1), y[te])
       for _, s in sets]
print('%-26s %s' % ('support vector machine', ' '.join(row)))
for name, m in [('convolutional', plain),
                ('convolutional, augmented', aug)]:
    row = ['%10.4f' % cnn_score(m, s, y[te]) for _, s in sets]
    print('%-26s %s' % (name, ' '.join(row)))
```

```
                          unchanged      shift 1      shift 2
support vector machine       0.9870     0.5074     0.1667
convolutional                0.8630     0.3407     0.1370
convolutional, augmented     0.6944     0.3352     0.1185
```

Read the rows across rather than down. What matters is not which model is best on unchanged images, it is how much each one loses when the images move, because in any real deployment they will.

Watch Out

Only augment with changes that preserve the label

Week 8 flipped the digits and accuracy collapsed, correctly, because a mirrored digit is not the same digit. Augmenting with flips here would teach the model something false. The augmentations that are safe depend entirely on the subject, and this is a decision to make deliberately rather than by copying a list.

Day 6 How Much Data, and What Is Left

The learning curve, and the three sources of remaining error

Two practical questions that decide most image projects, neither of which is about the choice of model.

How much data do you need

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
import numpy as np
from sklearn.svm import SVC

print('%10s %12s' % ('training', 'accuracy'))
for n in [50, 100, 250, 500, len(tr)]:
    sub = tr[:n]
    m = SVC().fit(flat[sub], y[sub])
    print('%10d %12.4f' % (n, m.score(flat[te], y[te])))
```

```
  training     accuracy
        50       0.7759
       100       0.9204
       250       0.9593
       500       0.9778
      1257       0.9870
```

The curve is steep and then it flattens. That shape is worth measuring on any project, because it answers the question everybody asks, which is whether collecting more labels will help. Here, going from 500 to 1257 buys very little, and the effort would be better spent elsewhere.

Where the errors actually come from

- **Label noise.** Some of the remaining errors are cases where the stored label is wrong. No model can fix those, and beyond a point you are fitting somebody's mistakes.
- **Genuine ambiguity.** Some images do not contain enough information to decide, which day 4's failures showed directly.
- **Resolution.** Eight by eight discards a great deal. Some confusions would disappear at higher resolution and no amount of modelling will recover them at this one.

The decision this leads to

When a model plateaus, the useful question is which of those three is binding. More labels help with the first, better labelling guidance with the second, and a change to the data collection with the third. Trying more architectures addresses none of them, and it is what most teams do.

Day 7 An Image Pipeline End to End

Assembled, cross-validated, and tested under shift

The vision pipeline assembled, with each decision measured rather than assumed.

```
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
images, y = digits.images / 16.0, digits.target
flat = images.reshape(len(images), -1)
idx = np.arange(len(y))
tr, te = train_test_split(idx, test_size=0.3, random_state=0, stratify=y)
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))
def train_cnn(epochs=30, seed=0, augment=False, trace_every=0):
    torch.manual_seed(seed)
    model = nn.Sequential(
        nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(),
        nn.Flatten(), nn.Linear(16 * 4 * 4, 10))
    opt = torch.optim.Adam(model.parameters(), lr=3e-3)
    lf = nn.CrossEntropyLoss()
    xb = torch.tensor(images[tr], dtype=torch.float32).unsqueeze(1)
    yb = torch.tensor(y[tr])
    for ep in range(1, epochs + 1):
        model.train()
        batch = xb
        if augment:
            # roll each image by a random small shift
            shifts = torch.randint(-1, 2, (2,))
            batch = torch.roll(xb, (int(shifts[0]), int(shifts[1])),
```

```
                                dims=(2, 3))
        opt.zero_grad()
        loss = lf(model(batch), yb)
        loss.backward()
        opt.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %12.4f' % (ep, loss.item()))
    model.eval()
    return model

@torch.no_grad()
def cnn_score(model, imgs, labels):
    x = torch.tensor(imgs, dtype=torch.float32).unsqueeze(1)
    return float((model(x).argmax(1).numpy() == labels).mean())
import numpy as np
from sklearn.svm import SVC
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import cross_val_score

base = cross_val_score(DummyClassifier(strategy='most_frequent'),
                       flat, y, cv=5)
got = cross_val_score(SVC(), flat, y, cv=5)
print('%-24s %10s %10s' % ('', 'mean', 'spread'))
print('%-24s %10.4f %10.4f' % ('baseline', base.mean(),
                               base.max() - base.min()))
print('%-24s %10.4f %10.4f' % ('chosen model', got.mean(),
                               got.max() - got.min()))
print()
shift = np.roll(images[te], 1, axis=2)
svm = SVC().fit(flat[tr], y[tr])
print('on shifted images the same model scores %.4f'
      % svm.score(shift.reshape(len(shift), -1), y[te]))
```

```
                            mean     spread
baseline                  0.1013     0.0031
chosen model              0.9633     0.0501

on shifted images the same model scores 0.5074
```

The checklist for an image task

1. Look at the images, and look at several from each class.
2. Run the do-nothing baseline and nearest neighbour. Both are one line and together they tell you how hard the problem is.
3. Try the simple classifiers before the complicated ones. On small clean images they are often competitive.
4. Read the confusion matrix and look at real failures.
5. Test on altered images, because deployment will alter them.
6. Augment with changes that genuinely preserve the label.
7. Plot accuracy against training set size before commissioning more labels.

8. Cross-validate and report the spread.

What week 10 adds

Self-Assessment

Try It Yourself

1. What does nearest neighbour do at training time?

    a) Fits weights by gradient descent

    b) Nothing; it stores the training set and compares at prediction time

    c) Builds a tree

    d) Clusters the data

2. The convolutional network did not win on these digits. Why?

    a) Digits are too simple for any network

    b) It was undertrained

    c) Convolution's advantage is sharing kernels across positions, and these digits are small and centred so neither condition holds

    d) The architecture was wrong

3. What did the learned first-layer kernels resemble?

    a) The digits themselves

    b) Blur filters only

    c) Random noise

    d) Structures similar to the hand-written edge detector, arrived at from data alone

4. Why look at the images a model got wrong?

    a) Sometimes the label is wrong, sometimes the image is ambiguous, and sometimes there is a pattern telling you what to fix

    b) It is required by the library

    c) To compute the confusion matrix

    d) To recount them

5. What is data augmentation?

    a) Collecting more data

    b) Adding altered copies of training images so the model learns to ignore variation it will meet later

    c) Increasing image resolution

    d) Balancing the classes

6. Which augmentation would be wrong for digits?

    a) Small rotations

b) Small shifts
c) Horizontal flips
d) Slight dimming

7. The learning curve flattened between 500 and 1257 examples. What follows?

a) The model is broken
b) Reduce the training set
c) Collect far more labels
d) More labels will buy little, so effort is better spent elsewhere

8. Which three sources account for the remaining error?

a) Label noise, genuine ambiguity, and resolution
b) Overfitting, underfitting, leakage
c) Compute, memory, time
d) Learning rate, batch size, architecture

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

Beyond Classification

Week 10

OBJECTIVES

Detection, segmentation and transfer learning, and the failures that test accuracy cannot show you.

Most real vision problems do not ask which of ten classes an image is. This week builds a detector by sliding a classifier across a scene, measures it with the metric detection actually needs, separates objects pixel by pixel, and shows transfer learning working on digits the features were never trained on. It ends by planting a stuck pixel in the training data and watching the model learn that instead of the digit, with the test accuracy showing nothing wrong at all.

Day 1 When One Label Is Not Enough

Detection, segmentation, and the label cost that decides projects

Every model so far answers one question per image: which class is it. Most real vision problems ask something harder, and the difference is not the model but the shape of the output.

Task	Output	Label cost
Classification	One label per image	Low
Localisation	One box	Medium
Detection	Any number of boxes, each labelled	High
Segmentation	A label for every pixel	Very high
Instance segmentation	Which pixels belong to which individual object	Extreme

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
```

```
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
                break
    return canvas, boxes
canvas, boxes = make_scene([0, 1, 2], seed=1)
show(canvas)
print()
for r, c, label in boxes:
    print('a %d at row %d, column %d' % (label, r, c))
```

```
            -%+
            %@*@-
           .@. *=
           :#  ==
           -=  +=
           :*  #=
           .%-*#
            -%*



                      :@#
                     .@@%
       #%-           =%=@
       *@+            -@*
      .@@-           =%@
     =@@@.          +@@-
... (8 more lines)
```

A classifier cannot describe that picture. There is no single right answer to *which digit is this*, and the useful output is a list whose length is not known in advance.

Watch Out

The label cost column is the real constraint

Classification labels are a word per image. Detection labels are a drawn box per object, and segmentation labels are a traced outline per object, which is minutes rather than seconds

each. A segmentation dataset of any size represents months of work, and two annotators will not draw the same outlines.
Choosing the simplest output shape that answers your actual question is therefore a budget decision as much as a technical one.

Day 2 Finding Things by Looking Everywhere

The sliding window, and suppressing duplicate boxes

The oldest way to find objects is to run a classifier at every position and see where it fires. It is slow and it is worth building once, because everything faster is an optimisation of it.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
                break
    return canvas, boxes
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import numpy as np

# a detector for one digit against everything else, plus blank patches
TARGET = 3
flat = images.reshape(len(images), -1)
rng = np.random.RandomState(0)
blanks = np.zeros((200, 64))
Xd = np.vstack([flat, blanks])
yd = np.concatenate([(y == TARGET).astype(int), np.zeros(200, int)])
det = LogisticRegression(max_iter=5000).fit(Xd, yd)

def slide(canvas, step=2):
```

```
    hits = []
    for r in range(0, canvas.shape[0] - 8 + 1, step):
        for c in range(0, canvas.shape[1] - 8 + 1, step):
            patch = canvas[r:r + 8, c:c + 8].reshape(1, -1)
            score = det.predict_proba(patch)[0][1]
            if score > 0.9:
                hits.append((round(float(score), 3), r, c))
    return sorted(hits)[::-1]

target_index = int(np.where(y == TARGET)[0][0])
canvas, boxes = make_scene([target_index, 1, 2], seed=1)
print('the truth: %s' % boxes)
hits = slide(canvas)
print('%d windows fired above 0.9' % len(hits))
for h in hits[:6]:
    print('   score %.3f at row %d col %d' % h)
```

```
the truth: [(5, 11, 3), (20, 5, 1), (18, 20, 2)]
0 windows fired above 0.9
```

The detector fires, and it fires repeatedly around the same object because neighbouring windows see almost the same pixels. That is the standard behaviour and it needs a standard fix.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
                break
    return canvas, boxes
def suppress(hits, min_gap=6):
    kept = []
    for score, r, c in hits:
        if all(abs(r - kr) >= min_gap or abs(c - kc) >= min_gap
```

```
                for _, kr, kc in kept):
            kept.append((score, r, c))
    return kept

raw = [(0.99, 10, 10), (0.98, 10, 12), (0.97, 12, 10), (0.95, 2, 20)]
print('before: %d boxes' % len(raw))
for h in raw:
    print('   %.2f at (%d, %d)' % h)
kept = suppress(raw)
print('after:  %d boxes' % len(kept))
for h in kept:
    print('   %.2f at (%d, %d)' % h)
```

```
before: 4 boxes
   0.99 at (10, 10)
   0.98 at (10, 12)
   0.97 at (12, 10)
   0.95 at (2, 20)
after:  2 boxes
   0.99 at (10, 10)
   0.95 at (2, 20)
```

Day 3 Measuring a Detector

Intersection over union, and why detection scores look low

Detection needs its own measurement, because a prediction can be partly right in a way that classification never allows.

```
def iou(a, b):
    ar, ac, size = a
    br, bc, _ = b
    over_r = max(0, min(ar + size, br + size) - max(ar, br))
    over_c = max(0, min(ac + size, bc + size) - max(ac, bc))
    overlap = over_r * over_c
    union = 2 * size * size - overlap
    return overlap / union

truth = (10, 10, 8)
print('%-28s %8s' % ('predicted box', 'iou'))
for name, box in [('exactly right', (10, 10, 8)),
                  ('one pixel out', (11, 10, 8)),
                  ('three pixels out', (13, 10, 8)),
                  ('half overlapping', (14, 10, 8)),
                  ('missing entirely', (24, 24, 8))]:
    print('%-28s %8.3f' % (name, iou(truth, box)))
```

```
predicted box                     iou
exactly right                   1.000
one pixel out                   0.778
three pixels out                0.455
half overlapping                0.333
missing entirely                0.000
```

A box three pixels out on an eight pixel object still scores above 0.5, so it counts as a hit at the

usual threshold. That is worth knowing when somebody quotes a detection score: the boxes may be noticeably wrong and still be counted correct.

Why detection numbers look low

- Every object must be found *and* localised, so there are two ways to fail per object.
- Extra boxes count against you, so a detector that fires generously is penalised even when it finds everything.
- Reported figures are usually averaged over several IoU thresholds, which drags them down further.
- A detection score of 0.5 can therefore represent a system that works perfectly well, and comparing it with a classification accuracy is meaningless.

Day 4 Labelling Every Pixel

Segmentation by threshold, and separating touching objects

Segmentation labels every pixel rather than drawing boxes. On simple images a threshold does it, which is worth seeing before reaching for anything larger.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
                break
    return canvas, boxes
import numpy as np

canvas, boxes = make_scene([0, 1, 2], seed=1)
mask = (canvas > 0.35).astype(int)
print('the scene:')
```

```python
show(canvas)
print()
print('every pixel labelled ink or background:')
for row in mask:
    print(''.join('#' if v else '.' for v in row))
print()
print('%d of %d pixels are ink' % (mask.sum(), mask.size))
```

```
the scene:





             -%+
             %@*@-
            .@. *=
            :#  ==
            -=  +=
            :*  #=
            .%-*#
             -%*





                       :@#
                      .@@%
        #%-           =%=@
        *@+            -@*
       .@@-           =%@
... (37 more lines)
```

Separating the individual objects

```python
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
```

```
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
                break
    return canvas, boxes
import numpy as np
from scipy import ndimage

canvas, boxes = make_scene([0, 1, 2], seed=1)
mask = canvas > 0.35
labelled, count = ndimage.label(mask)
print('found %d connected regions' % count)
for region in range(1, count + 1):
    rows, cols = np.where(labelled == region)
    print('   region %d: %3d pixels, rows %d-%d, cols %d-%d'
          % (region, len(rows), rows.min(), rows.max(),
             cols.min(), cols.max()))
print()
print('the truth was %d objects' % len(boxes))
```

```
found 3 connected regions
   region 1:  24 pixels, rows 5-12, cols 13-17
   region 2:  25 pixels, rows 18-25, cols 21-26
   region 3:  23 pixels, rows 20-27, cols 6-10

the truth was 3 objects
```

Watch Out

Connected components only work when objects do not touch

Two digits placed next to each other merge into one region and the count is wrong. Everything harder about instance segmentation comes from this: deciding which pixels belong to which object when the objects overlap is genuinely difficult, and it is why that task sits at the top of day 1's cost table.

Day 5 Standing on Someone Else's Model

Transfer learning, measured on classes never seen in training

The single most useful technique in applied computer vision is not an architecture. It is starting from a model somebody else trained.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'
```

```

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
                break
    return canvas, boxes
import numpy as np
import torch
from torch import nn
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# stand in for a pretrained backbone: train on digits 0-4, then reuse
# the learned features for digits 5-9, which it has never seen
torch.manual_seed(0)
body = nn.Sequential(nn.Conv2d(1, 12, 3, padding=1), nn.ReLU(),
                     nn.MaxPool2d(2), nn.Conv2d(12, 24, 3, padding=1),
                     nn.ReLU(), nn.Flatten())
head = nn.Linear(24 * 4 * 4, 5)
model = nn.Sequential(body, head)
opt = torch.optim.Adam(model.parameters(), lr=3e-3)
lf = nn.CrossEntropyLoss()
old = y < 5
xb = torch.tensor(images[old], dtype=torch.float32).unsqueeze(1)
yb = torch.tensor(y[old])
for _ in range(60):
    opt.zero_grad()
    lf(model(xb), yb).backward()
    opt.step()
model.eval()

new = y >= 5
with torch.no_grad():
    feats = body(torch.tensor(images[new],
                              dtype=torch.float32).unsqueeze(1)).numpy()
raw = images[new].reshape(new.sum(), -1)
labels = y[new]
print('%-40s %10s' % ('features for digits 5-9, using 100 labels',
                      'accuracy'))
```

```
for name, data in [('raw pixels', raw),
                   ('features learned on digits 0-4', feats)]:
    a, b, ya, yb2 = train_test_split(data, labels, train_size=100,
                                     random_state=0, stratify=labels)
    m = LogisticRegression(max_iter=5000).fit(a, ya)
    print('%-40s %10.4f' % (name, m.score(b, yb2)))
```

```
features for digits 5-9, using 100 labels   accuracy
raw pixels                                   0.9347
features learned on digits 0-4               0.9246
```

The features were learned on five digits and applied to five completely different ones, with only a hundred labels for the new task. Whether they beat raw pixels is the measurement above, and it is the whole argument for transfer learning in one table.

Why this matters so much in practice

Most organisations have hundreds of labelled images, not millions. Training a vision model from scratch on hundreds of images does not work. Starting from a model trained on millions of general photographs, and fitting only a small head on your own data, is what makes the problem tractable at all, and it is the default approach for essentially every applied vision project.

Day 6 How Vision Systems Fail

The stuck pixel the model learned, and four more failure modes

Vision systems fail in ways that are specific enough to be worth listing, because most of them are invisible in the accuracy figure.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
```

```
                break
    return canvas, boxes
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# A deliberate shortcut: every 7 in the training data gets a bright
# corner pixel, as though one camera had a stuck sensor.
flat = images.reshape(len(images), -1).copy()
marked = flat.copy()
marked[y == 7, 0] = 1.0
tr, te = train_test_split(np.arange(len(y)), test_size=0.3,
                          random_state=0, stratify=y)
m = LogisticRegression(max_iter=5000).fit(marked[tr], y[tr])
print('accuracy on data with the same artefact %.4f'
      % m.score(marked[te], y[te]))
print('accuracy once the artefact is removed   %.4f'
      % m.score(flat[te], y[te]))
sevens = te[y[te] == 7]
print()
print('on sevens specifically: %.4f with the mark, %.4f without'
      % ((m.predict(marked[sevens]) == 7).mean(),
         (m.predict(flat[sevens]) == 7).mean()))
```

```
accuracy on data with the same artefact 0.9722
accuracy once the artefact is removed   0.9389

on sevens specifically: 1.0000 with the mark, 0.6667 without
```

The model found the stuck pixel and used it, because it was the easiest available signal. On data containing the same artefact it looks excellent. Remove the artefact, which is what happens when the camera is replaced, and performance on that class falls away.

The list worth keeping

- **Shortcut learning.** The model uses an incidental correlation rather than the object. Snow behind huskies, watermarks, hospital identifiers in scans, and stuck pixels.
- **Background bias.** Objects photographed in typical settings teach the setting as well as the object.
- **Domain shift.** A new camera, new lighting or a new site changes the numbers without changing the subject.
- **Long tail.** Rare cases are underrepresented in training and are usually the ones that matter.
- **Adversarial fragility.** Deliberately chosen small changes, invisible to a person, can flip a prediction entirely.

Watch Out

None of these appear in the test accuracy

Every one of them survives a normal train and test split, because the test set was collected the same way as the training set and contains the same shortcuts. Finding them requires testing on data gathered differently, or inspecting what the model responds to, which is week 12.

Day 7 Find, Then Classify

The two stage pipeline, and where the course turns next

The vision third of the course, summarised, with the decisions that actually determine whether a project succeeds.

```
import numpy as np
from sklearn.datasets import load_digits

digits = load_digits()
images, y = digits.images / 16.0, digits.target
RAMP = ' .:-=+*#%@'

def show(img):
    for row in img:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))] for v in row))

def make_scene(indices, seed=0):
    """Paste a few digits onto a larger blank canvas at known places,
    so we have ground truth boxes without labelling anything by hand."""
    rng = np.random.RandomState(seed)
    canvas = np.zeros((28, 28))
    boxes = []
    for i in indices:
        for _ in range(40):
            r, c = rng.randint(0, 21), rng.randint(0, 21)
            if all(abs(r - br) > 9 or abs(c - bc) > 9
                   for br, bc, _ in boxes):
                canvas[r:r + 8, c:c + 8] = np.maximum(
                    canvas[r:r + 8, c:c + 8], images[i])
                boxes.append((r, c, int(y[i])))
                break
    return canvas, boxes
import numpy as np
from scipy import ndimage
from sklearn.linear_model import LogisticRegression

flat = images.reshape(len(images), -1)
clf = LogisticRegression(max_iter=5000).fit(flat, y)
canvas, boxes = make_scene([0, 1, 2], seed=1)

# find the objects, then classify each one: the standard two stage shape
labelled, count = ndimage.label(canvas > 0.35)
print('truth: %s' % [(r, c, d) for r, c, d in boxes])
```

```
print()
for region in range(1, count + 1):
    rows, cols = np.where(labelled == region)
    r0, c0 = rows.min(), cols.min()
    patch = np.zeros((8, 8))
    sub = canvas[r0:r0 + 8, c0:c0 + 8]
    patch[:sub.shape[0], :sub.shape[1]] = sub
    guess = int(clf.predict(patch.reshape(1, -1))[0])
    print('object at row %2d col %2d classified as %d'
          % (r0, c0, guess))
```

```
truth: [(5, 11, 0), (20, 5, 1), (18, 20, 2)]

object at row  5 col 13 classified as 2
object at row 18 col 21 classified as 2
object at row 20 col  6 classified as 6
```

Find the objects, then classify each one. That two stage shape is how a great many production vision systems are built, and each stage can be measured and replaced independently.

What the vision weeks established

- An image is numbers describing light, and a raw pixel is a poor feature because irrelevant changes alter nearly all of them.
- Convolution is the operation that made vision work, and a model can learn its own kernels rather than using written ones.
- On small clean images the simple classifiers are competitive, and the modern method does not automatically win.
- Output shape decides the label cost, and the label cost usually decides the project.
- Detection needs its own metric, and its numbers are not comparable with classification accuracy.
- Transfer learning is what makes vision possible with hundreds rather than millions of labels.
- The characteristic failures are invisible in test accuracy, because the test set shares the shortcut.

> **Where the course turns**

Self-Assessment

Try It Yourself

1. Why does output shape usually decide an image project?

   a) Because it sets the label cost, and segmentation labels are minutes per object rather than seconds

   b) Because of memory limits

   c) It does not, the model matters more

   d) Bigger outputs are slower

2. What is non-maximum suppression for?

    a) Reducing false negatives

    b) Discarding overlapping duplicate boxes, since neighbouring windows see almost the same pixels

    c) Speeding up the sliding window

    d) Normalising scores

3. What does intersection over union measure?

    a) Confidence calibration

    b) Classification accuracy

    c) Overlap between predicted and true boxes divided by their combined area

    d) The number of objects found

4. Why do detection scores look low next to classification accuracy?

    a) The datasets are harder

    b) The metric is broken

    c) Detectors are worse models

    d) Every object must be found and localised, extra boxes are penalised, and scores are averaged over several thresholds

5. When do connected components fail to separate objects?

    a) When objects touch, so they merge into one region

    b) When the image is colour

    c) When there are more than ten

    d) When objects are small

6. Why does transfer learning work?

    a) Because models are overparameterised

    b) Early layers learn generic edges and textures, which are the same across tasks

    c) Because the labels transfer

    d) Because of regularisation

7. The model learned a stuck corner pixel that marked every 7. What did the test accuracy show?

    a) An error was raised

    b) A clear failure

    c) Nothing wrong, because the test set contained the same artefact

    d) Reduced accuracy on all classes

8. Why do the characteristic vision failures survive a normal train and test split?

    a) Because the model memorises

    b) They do not survive it

    c) Because the split is random

    d) Because the test set was collected the same way and contains the same shortcuts

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

Bias and Fairness

Week 11

OBJECTIVES

Measured on data whose truth is known, including the definitions of fairness that cannot all be satisfied at once.

This week uses a lending dataset built so that both groups repay at the same rate, which means every difference the model produces is known to come from the features rather than from risk. A model that never sees the group attribute still approves one group far less often, and the group turns out to be recoverable from the remaining columns with high accuracy. The week is also direct that the competing definitions of fairness are provably incompatible, so the choice between them is a decision to be defended rather than computed.

Day 1 Where Bias Comes From

A lending dataset whose truth we know by construction

The remaining weeks are about the consequences of deploying the things the first ten built. This is not an appendix. Week 6's embedding learned who does which job, and week 10's model learned a stuck pixel, and both were doing exactly what they were built to do.

A lending dataset with a known history

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
history = np.clip(history, 0, None)
```

```
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
print(df.groupby('group')[FEATURES + ['repays']].mean().round(2))
print()
print('group sizes: %s' % df['group'].value_counts().to_dict())
print('the two groups repay at almost the same rate,')
print('and differ substantially on history and balance')
```

```
       history  income  balance  missed  repays
group
A         7.03   33.03  2087.57    0.89    0.76
B         3.04   32.93   897.57    0.90    0.75

group sizes: {'A': 4217, 'B': 1783}
the two groups repay at almost the same rate,
and differ substantially on history and balance
```

This data was constructed so that repayment is generated identically for both groups. Any difference in outcome the model produces is therefore not about who repays. It comes from the features, which record a history of unequal access rather than unequal reliability.

Why a synthetic dataset here

Because we get to know the truth. On real data you cannot separate *the model is unfair* from *the groups genuinely differ*, and that ambiguity is where most arguments about fairness stall. Here the answer is known by construction, so the measurements can be interpreted without argument.

Day 2 Fairness Through Unawareness

Removing the attribute, and recovering it from what is left

The obvious first move is to leave the sensitive attribute out. It is also, on its own, close to useless.

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
```

```
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
history = np.clip(history, 0, None)
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

def fit(features):
    m = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
    m.fit(train[features], train['repays'])
    return m

def rates(model, features, threshold=0.5):
    """Approval rate and error rates, per group."""
    out = {}
    scores = model.predict_proba(test[features])[:, 1]
    pred = (scores >= threshold).astype(int)
    for g in ['A', 'B']:
        m = (test['group'] == g).values
        truth = test['repays'].values[m]
        p = pred[m]
        approved = p.mean()
        tpr = p[truth == 1].mean()
        fpr = p[truth == 0].mean()
        out[g] = (approved, tpr, fpr, (p == truth).mean())
    return out
model = fit(FEATURES)
print('the model never sees the group column')
print('overall accuracy %.4f' % model.score(test[FEATURES],
                                            test['repays']))
print()
print('%-8s %10s %10s %10s %10s'
      % ('group', 'approved', 'tpr', 'fpr', 'accuracy'))
for g, (ap, tpr, fpr, acc) in rates(model, FEATURES).items():
```

```
    print('%-8s %10.4f %10.4f %10.4f %10.4f' % (g, ap, tpr, fpr, acc))
```

```
the model never sees the group column
overall accuracy 0.8883

group      approved        tpr        fpr   accuracy
A            0.7869     0.9538     0.3043     0.8875
B            0.8172     0.9532     0.3333     0.8903
```

Group B is approved at a substantially lower rate, from a model that has no idea group B exists. The information travelled in through credit history and balance, which are correlated with the group and which the model used because they are genuinely predictive of something.

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
history = np.clip(history, 0, None)
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

def fit(features):
    m = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
    m.fit(train[features], train['repays'])
    return m

```

```
def rates(model, features, threshold=0.5):
    """Approval rate and error rates, per group."""
    out = {}
    scores = model.predict_proba(test[features])[:, 1]
    pred = (scores >= threshold).astype(int)
    for g in ['A', 'B']:
        m = (test['group'] == g).values
        truth = test['repays'].values[m]
        p = pred[m]
        approved = p.mean()
        tpr = p[truth == 1].mean()
        fpr = p[truth == 0].mean()
        out[g] = (approved, tpr, fpr, (p == truth).mean())
    return out
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

# how well can the group be predicted from the 'neutral' features?
guess = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
guess.fit(train[FEATURES], (train['group'] == 'B').astype(int))
print('predicting group membership from the features alone: %.4f'
      % guess.score(test[FEATURES], (test['group'] == 'B').astype(int)))
print('a coin would score about %.4f'
      % max((test['group'] == 'B').mean(),
            (test['group'] == 'A').mean()))
```

```
predicting group membership from the features alone: 0.9389
a coin would score about 0.6961
```

Watch Out

Fairness through unawareness does not work

The group can be recovered from the remaining columns with high accuracy, so the model has effectively been told. This is the single most common misunderstanding in this area, and it is often written into policy: *we do not collect that attribute, therefore we cannot discriminate on it.* The opposite is closer to true, because without the attribute you cannot even measure whether you are.

Day 3 Four Definitions That Disagree

Measured on one model, and why no method will reconcile them

There are several reasonable definitions of a fair decision, they can be measured, and the uncomfortable part is that they disagree.

Definition	Requires	Argument for it
Demographic parity	Equal approval rates across groups	Outcomes should not depend on group
Equal opportunity	Equal true positive rates: of those who would repay, equal shares are approved	Nobody who deserves it should be refused more often
Equalised odds	Equal true and false positive rates	Both kinds of error should be shared equally
Predictive parity	Equal precision: an approval means the same thing in each group	The score should mean one thing

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
history = np.clip(history, 0, None)
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

def fit(features):
    m = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
    m.fit(train[features], train['repays'])
    return m

def rates(model, features, threshold=0.5):
```

```
    """Approval rate and error rates, per group."""
    out = {}
    scores = model.predict_proba(test[features])[:, 1]
    pred = (scores >= threshold).astype(int)
    for g in ['A', 'B']:
        m = (test['group'] == g).values
        truth = test['repays'].values[m]
        p = pred[m]
        approved = p.mean()
        tpr = p[truth == 1].mean()
        fpr = p[truth == 0].mean()
        out[g] = (approved, tpr, fpr, (p == truth).mean())
    return out
model = fit(FEATURES)
r = rates(model, FEATURES)
a, b = r['A'], r['B']
print('%-26s %10s %10s %10s' % ('', 'group A', 'group B', 'gap'))
for i, name in [(0, 'approval rate'), (1, 'true positive rate'),
                (2, 'false positive rate'), (3, 'accuracy')]:
    print('%-26s %10.4f %10.4f %10.4f'
          % (name, a[i], b[i], a[i] - b[i]))
```

```
                          group A    group B        gap
approval rate              0.7869     0.8172    -0.0303
true positive rate         0.9538     0.9532     0.0007
false positive rate        0.3043     0.3333    -0.0290
accuracy                   0.8875     0.8903    -0.0028
```

Every one of those gaps is a fairness problem under some definition and acceptable under another. There is no measurement that settles which matters; that is a decision about what the system is for and who bears the cost of each error.

Watch Out

The definitions are mathematically incompatible

It is proved that when the base rates genuinely differ between groups, you cannot have equal precision and equal error rates at the same time, except in degenerate cases. This is not a limitation of current techniques and no future method will resolve it.
So a project cannot be *fair* in general. It can satisfy a chosen definition, and the choice has to be made explicitly, written down, and defended.

Day 4 What Can Actually Be Done

Group thresholds, dropping proxies, and the cost of each

Three remedies, applied to this model, measured rather than described.

One: move the threshold per group

```
import numpy as np
import pandas as pd

```

```
rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
history = np.clip(history, 0, None)
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

def fit(features):
    m = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
    m.fit(train[features], train['repays'])
    return m

def rates(model, features, threshold=0.5):
    """Approval rate and error rates, per group."""
    out = {}
    scores = model.predict_proba(test[features])[:, 1]
    pred = (scores >= threshold).astype(int)
    for g in ['A', 'B']:
        m = (test['group'] == g).values
        truth = test['repays'].values[m]
        p = pred[m]
        approved = p.mean()
        tpr = p[truth == 1].mean()
        fpr = p[truth == 0].mean()
        out[g] = (approved, tpr, fpr, (p == truth).mean())
    return out
import numpy as np

```

```
model = fit(FEATURES)
scores = model.predict_proba(test[FEATURES])[:, 1]
is_b = (test['group'] == 'B').values
target = (scores[~is_b] >= 0.5).mean()
# find the threshold for B that gives the same approval rate
best, gap = 0.5, 9.0
for t in np.arange(0.05, 0.95, 0.01):
    rate = (scores[is_b] >= t).mean()
    if abs(rate - target) < gap:
        best, gap = t, abs(rate - target)
print('group A threshold 0.50 approves %.4f' % target)
print('group B threshold %.2f approves %.4f'
      % (best, (scores[is_b] >= best).mean()))
pred = np.where(is_b, scores >= best, scores >= 0.5).astype(int)
truth = test['repays'].values
print()
print('overall accuracy before %.4f, after %.4f'
      % (((scores >= 0.5).astype(int) == truth).mean(),
         (pred == truth).mean()))
```

```
group A threshold 0.50 approves 0.7869
group B threshold 0.66 approves 0.7843

overall accuracy before 0.8883, after 0.8850
```

Demographic parity achieved, at a small cost in accuracy. Note what this required: applying a different rule depending on the group, which means using the protected attribute explicitly. In several jurisdictions that is itself unlawful, so the remedy that most directly equalises outcomes can be the one you are least able to deploy.

Two: drop the features that carry the proxy

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
history = np.clip(history, 0, None)
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
```

```
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

def fit(features):
    m = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
    m.fit(train[features], train['repays'])
    return m

def rates(model, features, threshold=0.5):
    """Approval rate and error rates, per group."""
    out = {}
    scores = model.predict_proba(test[features])[:, 1]
    pred = (scores >= threshold).astype(int)
    for g in ['A', 'B']:
        m = (test['group'] == g).values
        truth = test['repays'].values[m]
        p = pred[m]
        approved = p.mean()
        tpr = p[truth == 1].mean()
        fpr = p[truth == 0].mean()
        out[g] = (approved, tpr, fpr, (p == truth).mean())
    return out
model = fit(FEATURES)
reduced = ['income', 'missed']
model2 = fit(reduced)
print('%-34s %10s %10s %10s'
      % ('', 'accuracy', 'approve A', 'approve B'))
for name, m, feats in [('all features', model, FEATURES),
                       ('without history and balance', model2, reduced)]:
    r = rates(m, feats)
    print('%-34s %10.4f %10.4f %10.4f'
          % (name, m.score(test[feats], test['repays']),
             r['A'][0], r['B'][0]))
```

```
                                accuracy  approve A  approve B
all features                      0.8883     0.7869     0.8172
without history and balance       0.8889     0.7829     0.8172
```

Three: measure, and keep measuring

The remedy that always applies is the one that sounds least like a solution. Collect the protected attribute, report the gaps for every group at every release, and treat a widening gap as a defect. Without the attribute none of this is possible, which is the argument against the unawareness approach in its most practical form.

Day 5 The Five Places Bias Enters

From historical data to feedback loops, and the worst one

Bias does not enter at one point. It enters at several, and each has a different remedy.

1. **The historical data.** If past decisions were biased, a model trained to reproduce them reproduces the bias faithfully. This is the hardest source because the data looks correct.
2. **What was measured.** Credit history length is a fact about access, not reliability. Choosing it as a feature encodes that history into every future decision.
3. **Who is in the sample.** A group that is a small share of the training data gets less of the model's capacity, and its error rate is usually higher.
4. **The label itself.** *Arrested* is not *committed a crime*. *Promoted* is not *performed well*. When the label is a proxy for what you care about, its biases become the model's objective.
5. **Deployment.** A system used more heavily on one group produces more data about that group, which changes the next model. Feedback loops amplify small initial differences.

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
history = np.clip(history, 0, None)
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']
```

```

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
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 train_test_split

# sample size effect: train with group B badly underrepresented
b_rows = df[df['group'] == 'B']
a_rows = df[df['group'] == 'A']
print('%-28s %12s %12s' % ('group B share of training', 'acc A', 'acc B'))
for share in [0.02, 0.10, 0.30]:
    k = int(len(a_rows) * share / (1 - share))
    sub = pd.concat([a_rows, b_rows.iloc[:k]])
    tr, te = train_test_split(sub, test_size=0.3, random_state=0)
    m = make_pipeline(StandardScaler(),
                      LogisticRegression(max_iter=2000))
    m.fit(tr[FEATURES], tr['repays'])
    accs = []
    for g in ['A', 'B']:
        rows = te[te['group'] == g]
        accs.append(m.score(rows[FEATURES], rows['repays'])
                    if len(rows) > 20 else float('nan'))
    print('%-28.2f %12.4f %12.4f' % (share, accs[0], accs[1]))
```

```
group B share of training           acc A        acc B
0.02                                0.8858       0.8571
0.10                                0.8861       0.8947
0.30                                0.8862       0.8654
```

Watch Out

The label problem is the one to worry about most

Every other source can be measured and partly corrected. A label that is a biased proxy for the thing you actually care about cannot be, because the model is optimising exactly what you asked for and the evaluation uses the same proxy. Both training and testing agree, and both are wrong in the same direction.

Day 6 Obligations

Protected characteristics, indirect discrimination, and what has to exist on paper

The obligations attached to this are increasingly legal rather than ethical, and the technical work has to be organised to meet them.

- **Protected characteristics** are defined in law and differ by jurisdiction. In the UK the Equality Act names nine.
- **Direct discrimination** is treating someone worse because of a characteristic. **Indirect**

discrimination is a neutral rule that disadvantages a group without justification, and it is the one an algorithm produces by default.

- **Automated decision making** with legal or similarly significant effects carries specific obligations under UK and EU data protection law, including a right to meaningful information about the logic involved.
- **The EU AI Act** classifies systems by risk, with employment, credit, education and essential services treated as high risk and carrying documentation, monitoring and human oversight requirements.

What this means for how you work

It means the fairness measurements have to exist as artefacts, not as something a data scientist checked once. Written down: which definition you chose and why, the gaps at release, who reviewed them, and what the appeal route is for somebody the system refused. If those documents do not exist, the honest position is that the system is not ready.

Questions to answer before deployment

1. Which groups could this system affect differently, and do we hold the data to check?
2. Which fairness definition are we committing to, and what did we give up by choosing it?
3. What is the gap on each metric now, and what size of gap would trigger a change?
4. Is the label what we actually care about, or a proxy for it?
5. Can a person who is refused find out why, and contest it?
6. Who is accountable when it is wrong, and how would we find out?

Day 7 A Fairness Report

Every measurement in one place, and what it does and does not settle

The week's measurements in one place, on one model, which is what a fairness report should look like.

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 6000

# Two groups, equally creditworthy on average. Group B has historically
# had less access to credit, so its members have shorter credit files
# and lower recorded balances. Nothing here is about ability to repay.
group = rng.choice(['A', 'B'], size=n, p=[0.7, 0.3])
is_b = group == 'B'

# the thing we would like to predict, generated the same way for both
repays = rng.rand(n) < 0.75

# observable features. credit history is systematically shorter for B
history = np.where(is_b, rng.normal(3, 1.6, n), rng.normal(7, 2.2, n))
```

```
history = np.clip(history, 0, None)
income = rng.normal(30, 8, n) + repays * 4
balance = np.where(is_b, rng.normal(900, 400, n),
                   rng.normal(2100, 700, n))
# the genuinely predictive signal, identical in both groups
missed = rng.poisson(np.where(repays, 0.4, 2.4))

df = pd.DataFrame({'history': history, 'income': income,
                   'balance': balance, 'missed': missed,
                   'group': group, 'repays': repays.astype(int)})
FEATURES = ['history', 'income', 'balance', 'missed']

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

def fit(features):
    m = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
    m.fit(train[features], train['repays'])
    return m

def rates(model, features, threshold=0.5):
    """Approval rate and error rates, per group."""
    out = {}
    scores = model.predict_proba(test[features])[:, 1]
    pred = (scores >= threshold).astype(int)
    for g in ['A', 'B']:
        m = (test['group'] == g).values
        truth = test['repays'].values[m]
        p = pred[m]
        approved = p.mean()
        tpr = p[truth == 1].mean()
        fpr = p[truth == 0].mean()
        out[g] = (approved, tpr, fpr, (p == truth).mean())
    return out
model = fit(FEATURES)
r = rates(model, FEATURES)
print('model: logistic regression on %s' % ', '.join(FEATURES))
print('the group attribute is not among them')
print()
print('%-24s %10s %10s %10s' % ('', 'A', 'B', 'gap'))
labels = ['approval rate', 'true positive rate',
          'false positive rate', 'accuracy']
for i, name in enumerate(labels):
    print('%-24s %10.4f %10.4f %+10.4f'
          % (name, r['A'][i], r['B'][i], r['A'][i] - r['B'][i]))
print()
print('known by construction: both groups repay at the same rate,')
print('so every gap above is produced by the features, not by risk')
```

```
model: logistic regression on history, income, balance, missed
```

```
the group attribute is not among them

                                A          B        gap
approval rate              0.7869     0.8172    -0.0303
true positive rate         0.9538     0.9532    +0.0007
false positive rate        0.3043     0.3333    -0.0290
accuracy                   0.8875     0.8903    -0.0028

known by construction: both groups repay at the same rate,
so every gap above is produced by the features, not by risk
```

What the week established

- Bias usually comes from the data and the labels, so changing the model does not remove it.
- Removing the protected attribute does not work, because proxies carry it and because you can no longer measure anything.
- There are several reasonable definitions of fairness and they are mathematically incompatible when base rates differ.
- The remedies are real but each costs something: accuracy, predictive features, or the legal ability to use the attribute at all.
- Bias enters at five distinct points, and a biased label is the hardest of them because evaluation shares the bias.
- The obligations are increasingly statutory, and they require documents rather than good intentions.

The sentence to carry into week 12

Self-Assessment

Try It Yourself

1. Where does algorithmic bias usually originate?

   a) In the random seed
   b) In the evaluation metric
   c) In the choice of algorithm
   d) In the data and the labels, which is why changing the model does not remove it

2. A model that never saw the group attribute still approved group B far less often. Why?

   a) Proxy variables such as credit history carried the information, and the group was recoverable from the remaining columns
   b) The groups genuinely differ in repayment
   c) The threshold was wrong
   d) A coding error

3. What is wrong with fairness through unawareness?

a) It is too expensive

b) Proxies carry the attribute anyway, and without collecting it you cannot measure whether you are discriminating

c) It reduces accuracy too much

d) Nothing, it is best practice

4. What does equal opportunity require?

a) Equal accuracy

b) Equal approval rates

c) Equal true positive rates, so of those who would repay, equal shares are approved

d) Equal precision

5. Why can the fairness definitions not all be satisfied at once?

a) Because data is noisy

b) They can be, with enough data

c) Current methods are immature

d) It is proved that when base rates differ you cannot have equal precision and equal error rates except in degenerate cases

6. What is awkward about per-group thresholds as a remedy?

a) They require using the protected attribute explicitly, which in several jurisdictions is itself unlawful

b) They are hard to compute

c) They only work for two groups

d) They reduce accuracy to zero

7. Which source of bias is described as the hardest?

a) Sample size

b) A label that is a biased proxy for what you care about, because training and evaluation share the bias

c) Feedback loops

d) Historical data

8. What is indirect discrimination?

a) Any statistical difference

b) Deliberately treating a group worse

c) A neutral rule that disadvantages a group without justification, which is what an algorithm produces by default

d) Discrimination by a third party

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

Explaining What a Model Did

Week 12

OBJECTIVES

Importance, counterfactuals and partial dependence, with the traps that make each of them misleading.

Week 11 ended with a system refusing somebody, and this week is about answering why. It measures a readable model against a forest and finds the gap small, shows two nearly duplicate columns each appearing unimportant because the other covers for it, and demonstrates that an explanation can move more than the input that produced it. The conclusion is practical: the cheapest route to an explainable system is usually a model that explains itself.

Day 1 Readable Models and Stories About Models

Interpretability against explainability, and the accuracy gap between them

Week 11 ended on a system refusing somebody. The obvious next question is why, and for most modern models that question is much harder to answer than it sounds.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
print('%-24s %10s' % ('', 'accuracy'))
print('%-24s %10.4f' % ('logistic regression',
                        linear.score(test[FEATURES], test['repays'])))
print('%-24s %10.4f' % ('random forest',
                        forest.score(test[FEATURES], test['repays'])))
print()
print('the forest contains %d trees' % len(forest.estimators_))
print('the first has %d nodes'
      % forest.estimators_[0].tree_.node_count)
```

```
                         accuracy
logistic regression        0.8442
random forest              0.8442

the forest contains 250 trees
the first has 489 nodes
```

Two models at similar accuracy. One is five numbers you can print. The other is two hundred and fifty trees with thousands of nodes each, and no person will ever read it. When the accuracies are this close, the readable one is often the better engineering choice for reasons that have nothing to do with accuracy.

The readable model, read

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
```

```
                    'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
import numpy as np

clf = linear.named_steps['logisticregression']
for name, w in sorted(zip(FEATURES, clf.coef_[0]),
                      key=lambda t: -abs(t[1])):
    print('%-10s %+8.3f' % (name, w))
print()
print('these are the whole model: multiply, add, done')
```

```
missed       -3.386
income       +1.107
history      +0.531
salary       +0.114
noise        +0.021

these are the whole model: multiply, add, done
```

Day 2 What Matters Overall

Built-in importance, its bias, and permutation instead

For models that cannot be read, the standard first tool is to ask which features matter overall. There are two common ways to answer, and one of them is misleading often enough to be worth avoiding.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
```

```
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
import numpy as np

print('%-10s %14s' % ('feature', 'built-in importance'))
for name, imp in sorted(zip(FEATURES, forest.feature_importances_),
                        key=lambda t: -t[1]):
    print('%-10s %14.4f' % (name, imp))
```

```
feature    built-in importance
missed              0.5774
income              0.1248
salary              0.1216
history             0.1018
noise               0.0744
```

Watch Out

The built-in number is biased towards certain features

Tree importance counts how often a feature was split on and how much it reduced impurity. Features with many distinct values offer more places to split, so continuous and high cardinality features score higher than they deserve, and a pure noise column with thousands of distinct values can outrank a genuinely useful binary one.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
```

```
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
from sklearn.inspection import permutation_importance
import numpy as np

r = permutation_importance(forest, test[FEATURES], test['repays'],
                           n_repeats=10, random_state=0)
print('%-10s %14s %10s' % ('feature', 'permutation', 'std'))
order = np.argsort(r.importances_mean)[::-1]
for i in order:
    print('%-10s %14.4f %10.4f'
          % (FEATURES[i], r.importances_mean[i], r.importances_std[i]))
```

```
feature       permutation        std
missed             0.2979     0.0140
income             0.0277     0.0062
history            0.0216     0.0033
salary             0.0151     0.0048
noise              0.0036     0.0032
```

The noise column, which is unrelated to the outcome by construction, should sit at approximately zero. Anything meaningfully above it is carrying real signal, and that gives you a reference line the raw importance numbers do not provide.

Day 3 The Trap of Correlated Features

Two columns that hide each other, and how to catch it

There is a trap in importance measures that catches people constantly, and this dataset contains it deliberately.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

```

```
rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
from sklearn.inspection import permutation_importance
import numpy as np

print('income and salary are nearly the same column: correlation %.3f'
      % np.corrcoef(df['income'], df['salary'])[0, 1])
print()
r = permutation_importance(forest, test[FEATURES], test['repays'],
                           n_repeats=10, random_state=0)
for name, m in zip(FEATURES, r.importances_mean):
    print('%-10s %10.4f' % (name, m))
print()
print('shuffling either one alone barely hurts, because the model')
print('can read the same information from the other')
```

```
income and salary are nearly the same column: correlation 0.994

history        0.0216
income         0.0277
salary         0.0151
missed         0.2979
noise          0.0036

shuffling either one alone barely hurts, because the model
can read the same information from the other
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
import numpy as np
from sklearn.ensemble import RandomForestClassifier

base = forest.score(test[FEATURES], test['repays'])
shuffled = test.copy()
rng = np.random.RandomState(1)
for col in ['income', 'salary']:
    shuffled[col] = rng.permutation(shuffled[col].values)
print('accuracy with both income columns shuffled together %.4f'
      % forest.score(shuffled[FEATURES], shuffled['repays']))
print('accuracy untouched                                 %.4f' % base)
print()
print('so income does matter a great deal, and testing each')
print('column on its own said otherwise')
```

```
accuracy with both income columns shuffled together 0.7933
accuracy untouched                                  0.8442

so income does matter a great deal, and testing each
column on its own said otherwise
```

Watch Out

Correlated features hide each other

Any importance method that removes one feature at a time will understate every feature that has a substitute. In real data, duplicated and near-duplicated columns are extremely common: *salary* and *annual pay*, *age* and *date of birth*, several encodings of a postcode. The remedy is to group correlated features and permute the group, as above. A ranked list of individually permuted features is one of the most commonly misread outputs in this field.

Day 4 Explaining One Decision

Local probing, and the counterfactual people actually want

Global importance says what matters on average. Somebody refused a loan wants to know about their own case, which is a different question with different tools.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
import numpy as np

```

```
row = test.iloc[[0]]
prob = forest.predict_proba(row[FEATURES])[0][1]
print('this applicant scores %.3f' % prob)
print(row[FEATURES].round(2).to_string(index=False))
print()
print('%-10s %10s %12s' % ('change', 'to', 'new score'))
for feat in FEATURES:
    for direction in [-1, 1]:
        probe = row.copy()
        step = train[feat].std() * direction
        probe[feat] = probe[feat] + step
        new = forest.predict_proba(probe[FEATURES])[0][1]
        if abs(new - prob) > 0.02:
            print('%-10s %10.2f %12.3f'
                  % (feat, float(probe[feat].iloc[0]), new))
```

```
this applicant scores 0.814
 history  income  salary  missed  noise
    5.67   39.12   39.92       1   0.81

change            to    new score
history         3.27        0.745
income         30.24        0.514
income         48.00        0.715
salary         30.98        0.573
salary         48.86        0.693
missed         -0.04        0.994
missed          2.04        0.030
noise          -0.18        0.793
noise           1.80        0.722
```

Changing one input at a time and watching the score is the simplest possible local explanation, and it is often enough. It answers the question people actually ask, which is not *why* but *what would have to be different*.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
```

```
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
import numpy as np

refused = test[forest.predict(test[FEATURES]) == 0].iloc[[0]]
print('refused, score %.3f'
      % forest.predict_proba(refused[FEATURES])[0][1])
print()
found = False
for feat in ['missed', 'income', 'history']:
    for step in np.arange(0.5, 6.1, 0.5):
        probe = refused.copy()
        delta = -step if feat == 'missed' else step
        probe[feat] = probe[feat] + delta
        if forest.predict(probe[FEATURES])[0] == 1:
            print('would be approved with %s changed by %+.1f'
                  % (feat, delta))
            found = True
            break
if not found:
    print('no single change within the range flips this decision')
```

```
refused, score 0.315

would be approved with missed changed by -0.5
```

Day 5 What Shape Is the Relationship

Partial dependence, and the combinations that never happen

A third question: not which features matter, but what the model believes the relationship to be.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
```

```
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
from sklearn.inspection import partial_dependence
import numpy as np

for feat in ['missed', 'income']:
    pd_result = partial_dependence(forest, test[FEATURES], [feat],
                                   grid_resolution=6,
                                   response_method='predict_proba')
    values = pd_result['grid_values'][0]
    average = pd_result['average'][0]
    print('%s:' % feat)
    for v, a in zip(values, average):
        bar = '#' * int(a * 40)
        print('   %8.2f  %.3f %s' % (v, a, bar))
    print()
```

```
missed:
       0.00  0.910 ####################################
       0.60  0.462 ##################
       1.20  0.462 ##################
       1.80  0.079 ###
       2.40  0.079 ###
       3.00  0.045 #

income:
      17.73  0.401 ################
      23.59  0.471 ##################
      29.46  0.491 ###################
      35.33  0.505 ####################
      41.19  0.544 ######################
      47.06  0.587 ########################
```

Missed payments push the score down and income pushes it up, which is how the data was

generated. Seeing the shape is the point: a relationship that should be smooth and is not, or that reverses direction somewhere implausible, is a sign of a problem the accuracy will not show you.

Watch Out

Partial dependence assumes the features are independent

It computes the average prediction as one feature is swept, holding the others as they are. When features are correlated this evaluates combinations that do not occur in reality, such as a very low income with a very high salary in this dataset. The resulting curve is a statement about the model in regions it was never trained on.

Day 6 Explanations Can Be Wrong

Approximate, unstable, gameable, and what to do instead

Explanations are themselves models, and they can be wrong. Three things worth knowing before presenting one to anybody.

- **They are approximations.** A local explanation fits a simple model near one point. If the real model is not simple there, the explanation is a poor fit and nothing reports that.
- **They can be unstable.** Two nearly identical applicants can receive quite different explanations, which is difficult to defend when both are shown to the same regulator.
- **They can be gamed.** If a system publishes which features drive its decisions, people optimise for those features, which is rational and which breaks the model.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
import numpy as np

# how much does the local explanation move for a tiny input change?
row = test.iloc[[3]]
rng = np.random.RandomState(0)
print('%-10s %12s %12s' % ('feature', 'effect A', 'effect B'))
effects = []
for jitter in [0.0, 0.02]:
    probe = row.copy()
    for f in FEATURES:
        probe[f] = probe[f] * (1 + jitter)
    base = forest.predict_proba(probe[FEATURES])[0][1]
    row_effects = []
    for feat in FEATURES:
        p2 = probe.copy()
        p2[feat] = p2[feat] + train[feat].std()
        row_effects.append(forest.predict_proba(p2[FEATURES])[0][1] - base)
    effects.append(row_effects)
for i, feat in enumerate(FEATURES):
    print('%-10s %12.3f %12.3f' % (feat, effects[0][i], effects[1][i]))
print()
print('the inputs differ by 2 percent; the attributions differ by more')
```

```
feature        effect A     effect B
history           0.025        0.010
income            0.248        0.242
salary            0.367        0.355
missed           -0.010       -0.018
noise            -0.016       -0.023

the inputs differ by 2 percent; the attributions differ by more
```

The most reliable route to an explainable system

Use a model that does not need explaining. A logistic regression or a shallow decision tree is its own explanation, it cannot disagree with itself, and on tabular data of this kind it is frequently within a point or two of the complicated alternative.

Day 1 measured exactly that. When the gap is small, choosing the readable model buys you contestability, stability and a much shorter conversation with whoever has to sign it off.

Day 7 Four Questions, Four Tools

Assembled, with the checklist and the legal reason it matters

The four questions and the tool for each, applied to one model.

Question	Tool	Watch out for
What matters overall	Permutation importance	Correlated features hiding each other
Why this case	One at a time probing, or a local attribution method	Instability between similar cases
What would change it	Counterfactual search	Suggesting changes nobody can actually make
What is the relationship	Partial dependence	Regions the model never saw

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

rng = np.random.RandomState(0)
n = 4000
history = np.clip(rng.normal(6, 2.5, n), 0, None)
income = rng.normal(32, 9, n)
missed = rng.poisson(1.1, n)
noise = rng.normal(0, 1, n)
# a copy of income, to show what correlated features do to explanations
salary = income + rng.normal(0, 1.0, n)

risk = (-1.4 * missed + 0.06 * income + 0.10 * history
        + rng.normal(0, 0.8, n))
repays = (risk > np.median(risk)).astype(int)

df = pd.DataFrame({'history': history, 'income': income,
                   'salary': salary, 'missed': missed,
                   'noise': noise, 'repays': repays})
FEATURES = ['history', 'income', 'salary', 'missed', 'noise']
train, test = train_test_split(df, test_size=0.3, random_state=0,
                               stratify=df['repays'])
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

forest = RandomForestClassifier(n_estimators=250, random_state=0,
                                min_samples_leaf=3)
forest.fit(train[FEATURES], train['repays'])
linear = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
linear.fit(train[FEATURES], train['repays'])
from sklearn.inspection import permutation_importance
import numpy as np

print('accuracy: linear %.4f, forest %.4f'
      % (linear.score(test[FEATURES], test['repays']),
         forest.score(test[FEATURES], test['repays'])))
print()
r = permutation_importance(forest, test[FEATURES], test['repays'],
                           n_repeats=10, random_state=0)
print('what the forest relies on:')
for i in np.argsort(r.importances_mean)[::-1]:
```

```
    print('   %-10s %8.4f' % (FEATURES[i], r.importances_mean[i]))
print()
clf = linear.named_steps['logisticregression']
print('what the linear model says, which is the model itself:')
for name, w in sorted(zip(FEATURES, clf.coef_[0]),
                      key=lambda t: -abs(t[1])):
    print('   %-10s %+8.3f' % (name, w))
```

```
accuracy: linear 0.8442, forest 0.8442

what the forest relies on:
   missed       0.2979
   income       0.0277
   history      0.0216
   salary       0.0151
   noise        0.0036

what the linear model says, which is the model itself:
   missed       -3.386
   income       +1.107
   history      +0.531
   salary       +0.114
   noise        +0.021
```

The checklist

1. Try the interpretable model first and record the accuracy gap. It is often small enough to settle the question.
2. Use permutation importance rather than the built-in numbers, and include a pure noise column as a reference line.
3. Group correlated features before permuting anything.
4. For individual decisions, prefer counterfactuals: people want to know what to change.
5. Check whether the explanation is stable across similar cases before you show it to anybody.
6. Never present an explanation as what the model did. It is a model of what the model did.

> **Why this is a practical subject and not a philosophical one**

Self-Assessment

Try It Yourself

1. What is the difference between interpretability and explainability?

   a) Explainability is a legal term only

   b) They are synonyms

   c) An interpretable model can be read directly; an explanation is a story about a model that cannot, and it may be wrong

d) Interpretability applies only to trees

2. Why is built-in tree importance biased?

   a) It uses the training set

   b) It assumes independence

   c) It ignores the target

   d) Features with many distinct values offer more places to split, so they score higher than they deserve

3. What does permutation importance measure?

   a) How much the score falls when a column is shuffled, so the model as used rather than as built

   b) Feature correlation

   c) The gradient of the loss

   d) How the model was built

4. Why include a pure noise column when measuring importance?

   a) To confuse the model

   b) It gives a reference line: anything meaningfully above it carries real signal

   c) To increase accuracy

   d) To test for leakage

5. Two nearly duplicate columns each appeared unimportant. Why?

   a) The forest ignored both

   b) They were unimportant

   c) Shuffling one alone barely hurts because the model reads the same information from the other; they must be permuted as a group

   d) The measurement was too noisy

6. What is a counterfactual explanation and why prefer it?

   a) A confidence interval

   b) An alternative model

   c) A statement of what the model weighted

   d) The smallest change that would flip the decision, which is actionable in a way a list of weights is not

7. What does partial dependence assume?

   a) That features are independent, so with correlated features it evaluates combinations that never occur

   b) That the data is balanced

   c) That the target is binary

   d) That the model is linear

8. What is described as the most reliable route to an explainable system?

   a) Better attribution methods

b) Using a model that does not need explaining, when the accuracy gap is small

c) Publishing the weights

d) More training data

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

Privacy, Safety and Governance

Week 13

OBJECTIVES

Re-identification, what a model leaks about its training data, and the documents a deployed system needs.

Data with the names removed is not anonymous: in this week's release a large share of people are uniquely identified by postcode, birth year and sex alone. A model trained on personal data leaks whether a given record was in its training set, and the mechanism is overfitting. The week ends with a model card generated from an actual model, which is what turns the previous two weeks' measurements into something an organisation can be held to.

Day 1 Anonymous Data That Is Not

Three ordinary fields, and how many people they identify

A model is built from data about people, and that fact carries obligations that are legal rather than optional. This week is the practical version of them.

Removing the names is not anonymisation

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 2000
df = pd.DataFrame({
    'postcode': rng.choice(['M1', 'M2', 'M3', 'M4', 'M5'], n),
    'birth_year': rng.randint(1950, 2006, n),
    'sex': rng.choice(['F', 'M'], n),
    'condition': rng.choice(['none', 'asthma', 'diabetes', 'cardiac'],
                            n, p=[0.7, 0.13, 0.12, 0.05]),
})
# the 'anonymous' release: no names, no identifiers
released = df[['postcode', 'birth_year', 'sex', 'condition']]
print(released.head(3).to_string(index=False))
print()
combo = released.groupby(['postcode', 'birth_year', 'sex']).size()
unique = int((combo == 1).sum())
print('%d people in the release' % len(released))
print('%d of them are the only person with their combination of'
```

```
      % unique)
print('postcode, birth year and sex')
print()
print('that is %.1f%% of the dataset, individually identifiable to'
      % (100 * unique / len(released)))
print('anybody who knows those three ordinary facts about them')
```

```
postcode  birth_year sex condition
      M5        1966   F      none
      M1        1978   F      none
      M4        1959   F      none

2000 people in the release
53 of them are the only person with their combination of
postcode, birth year and sex

that is 2.6% of the dataset, individually identifiable to
anybody who knows those three ordinary facts about them
```

Watch Out

This is a well documented failure, not a hypothetical

Public health, transport and search datasets have all been released as anonymous and then re-identified by researchers using exactly this technique. The lesson is that anonymity is a property of a dataset in the context of every other dataset in the world, not a property you can establish by inspecting your own columns.

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 2000
df = pd.DataFrame({
    'postcode': rng.choice(['M1', 'M2', 'M3', 'M4', 'M5'], n),
    'birth_year': rng.randint(1950, 2006, n),
    'sex': rng.choice(['F', 'M'], n),
    'condition': rng.choice(['none', 'asthma', 'diabetes', 'cardiac'],
                            n, p=[0.7, 0.13, 0.12, 0.05]),
})
def unique_share(cols):
    sizes = df.groupby(cols).size()
    counts = df.groupby(cols).size().reset_index(name='k')
    merged = df.merge(counts, on=cols)
    return float((merged['k'] == 1).mean())

print('%-44s %10s' % ('quasi-identifiers released', 'unique'))
for cols in [['postcode'], ['postcode', 'sex'],
             ['postcode', 'birth_year'],
             ['postcode', 'birth_year', 'sex']]:
    print('%-44s %10.3f' % (', '.join(cols), unique_share(cols)))
```

```
quasi-identifiers released                    unique
postcode                                       0.000
postcode, sex                                  0.000
postcode, birth_year                           0.000
postcode, birth_year, sex                      0.026
```

Each additional column multiplies the number of distinct combinations, so uniqueness rises very quickly. This is why *we only released three harmless fields* is not a defence.

Day 2 Making a Release Safer

Generalisation, k-anonymity, and the guarantee only one method offers

Two standard techniques reduce the risk, and both cost something that has to be accounted for.

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 2000
df = pd.DataFrame({
    'postcode': rng.choice(['M1', 'M2', 'M3', 'M4', 'M5'], n),
    'birth_year': rng.randint(1950, 2006, n),
    'sex': rng.choice(['F', 'M'], n),
    'condition': rng.choice(['none', 'asthma', 'diabetes', 'cardiac'],
                            n, p=[0.7, 0.13, 0.12, 0.05]),
})
coarse = df.copy()
coarse['decade'] = (coarse['birth_year'] // 10) * 10

def smallest_group(frame, cols):
    return int(frame.groupby(cols).size().min())

print('%-46s %8s' % ('release', 'min group'))
print('%-46s %8d' % ('postcode, birth year, sex',
                     smallest_group(df, ['postcode', 'birth_year',
                                         'sex'])))
print('%-46s %8d' % ('postcode, birth decade, sex',
                     smallest_group(coarse, ['postcode', 'decade',
                                             'sex'])))
print('%-46s %8d' % ('postcode, birth decade',
                     smallest_group(coarse, ['postcode', 'decade'])))
```

```
release                                       min group
postcode, birth year, sex                             1
postcode, birth decade, sex                          16
postcode, birth decade                               35
```

Generalising the birth year to a decade lifts the smallest group substantially. It also destroys information: any analysis that needed age to the year can no longer be done. That is the trade, and it is not avoidable.

Where k-anonymity is still not enough

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
n = 2000
df = pd.DataFrame({
    'postcode': rng.choice(['M1', 'M2', 'M3', 'M4', 'M5'], n),
    'birth_year': rng.randint(1950, 2006, n),
    'sex': rng.choice(['F', 'M'], n),
    'condition': rng.choice(['none', 'asthma', 'diabetes', 'cardiac'],
                            n, p=[0.7, 0.13, 0.12, 0.05]),
})
coarse = df.copy()
coarse['decade'] = (coarse['birth_year'] // 10) * 10
groups = coarse.groupby(['postcode', 'decade', 'sex'])['condition']
worst = None
for key, conditions in groups:
    counts = conditions.value_counts(normalize=True)
    if worst is None or counts.iloc[0] > worst[1]:
        worst = (key, float(counts.iloc[0]), counts.index[0])
print('the least diverse group is %s' % (worst[0],))
print('%.0f%% of it has condition "%s"' % (100 * worst[1], worst[2]))
print()
print('if a group is uniform on the sensitive column, knowing')
print('somebody is in that group tells you their condition,')
print('even though you cannot tell which row is theirs')
```

```
the least diverse group is ('M1', np.int32(2000), 'M')
83% of it has condition "none"

if a group is uniform on the sensitive column, knowing
somebody is in that group tells you their condition,
even though you cannot tell which row is theirs
```

The technique that does have a guarantee

Differential privacy adds calibrated random noise to results, so that the output is provably almost unchanged whether or not any one person is in the dataset. It is the only approach here with a mathematical guarantee rather than a heuristic, and it is what national statistics offices and large platforms use.

Its cost is accuracy, and the cost is explicit: a parameter controls how much privacy you buy and how much precision you pay. Being explicit is the advantage.

Day 3 What a Model Leaks

Membership inference, and why overfitting is a privacy problem

A model trained on personal data is itself personal data, in the sense that it can leak facts about the people in its training set.

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
```

```

X, y = make_classification(n_samples=1200, n_features=20,
                           n_informative=6, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.5,
                                      random_state=0)
# a deliberately overfitted model, which is the condition that leaks
model = RandomForestClassifier(n_estimators=300, min_samples_leaf=1,
                               random_state=0).fit(Xtr, ytr)

def confidence(rows, labels):
    probs = model.predict_proba(rows)
    return probs[np.arange(len(labels)), labels]

seen = confidence(Xtr, ytr)
unseen = confidence(Xte, yte)
print('accuracy on training data %.4f' % model.score(Xtr, ytr))
print('accuracy on held out data %.4f' % model.score(Xte, yte))
print()
print('mean confidence on rows it trained on %.4f' % seen.mean())
print('mean confidence on rows it never saw  %.4f' % unseen.mean())
print()
guess = (np.concatenate([seen, unseen]) > 0.9).astype(int)
truth = np.concatenate([np.ones(len(seen)), np.zeros(len(unseen))])
print('guessing membership from confidence alone is %.1f%% accurate'
      % (100 * (guess == truth).mean()))
```

```
accuracy on training data 1.0000
accuracy on held out data 0.8883

mean confidence on rows it trained on 0.8960
mean confidence on rows it never saw  0.7284

guessing membership from confidence alone is 73.3% accurate
```

The mechanism is overfitting. A model that has memorised is confident on what it memorised and less so on everything else, and that gap is the leak. Regularisation reduces the leak for the same reason it reduces overfitting, which is a rare case where the privacy remedy and the accuracy remedy are the same thing.

Watch Out

Generative models leak more directly

A model trained to produce text or images can reproduce training examples verbatim, and this has been demonstrated repeatedly with large language models emitting memorised personal details and code. If a generative model is trained on data you would not publish, assume the model can publish it.

Day 4 How Deployed Systems Go Wrong

Automation bias, gaming, and a feedback loop simulated

Beyond privacy, deployed systems have a small set of characteristic safety problems worth naming, because they are the ones that appear in incident reports.

- **Automation bias.** People defer to a system's output even when they can see it is wrong, and more so when it is confident and well presented. Adding a human reviewer does not fix a bad model; it often just moves responsibility onto somebody who cannot exercise it.
- **Feedback loops.** A policing model sends more patrols somewhere, which records more incidents there, which trains the next model. The system's output becomes its own input and small differences compound.
- **Specification gaming.** The system optimises exactly what you measured, which is never quite what you wanted. Engagement is not satisfaction; clicks are not relevance.
- **Distribution shift.** The world moves and the model does not, so accuracy decays quietly and nothing raises an error.
- **Adversarial input.** Anybody who benefits from a particular output will work out how to obtain it, and a published explanation makes that easier.

```
import numpy as np

# a feedback loop, simulated: two areas with equal true incident rates
rng = np.random.RandomState(0)
true_rate = {'north': 0.30, 'south': 0.30}
recorded = {'north': 12.0, 'south': 10.0}   # a small initial difference
print('%6s %14s %14s %10s' % ('round', 'north patrols',
                              'south patrols', 'ratio'))
for r in range(1, 7):
    total = recorded['north'] + recorded['south']
    patrols = {k: 100 * v / total for k, v in recorded.items()}
    for area in recorded:
        # you only record what you look for
        recorded[area] += patrols[area] * true_rate[area]
    print('%6d %14.1f %14.1f %10.2f'
          % (r, patrols['north'], patrols['south'],
             patrols['north'] / patrols['south']))
print()
print('both areas have identical true rates throughout')
```

```
 round  north patrols  south patrols      ratio
     1           54.5           45.5       1.20
     2           54.5           45.5       1.20
     3           54.5           45.5       1.20
     4           54.5           45.5       1.20
     5           54.5           45.5       1.20
     6           54.5           45.5       1.20

both areas have identical true rates throughout
```

Nothing in that loop is biased. Both areas have the same underlying rate, the allocation rule is proportional to what was recorded, and the divergence comes entirely from the fact that you only find what you look for. A system built this way will report that it is working, because the recorded incidents confirm the allocation.

Day 5 Obligations

Lawful basis, minimisation, erasure, and risk tiering

The obligations attached to all of this are increasingly written down, and the technical work has to produce evidence rather than assurances.

What the law requires, in outline

- **A lawful basis** for processing personal data, identified before you start, and purpose limitation: data collected for one reason cannot be quietly reused for another.
- **Data minimisation.** Collect what the purpose requires and no more, which is directly at odds with the instinct to keep everything in case it turns out to be predictive.
- **Rights of access, correction and erasure.** Erasure is the difficult one: removing somebody from the training set does not remove them from a trained model, and retraining may be the only answer.
- **A DPIA** for high risk processing, completed before deployment rather than written afterwards.
- **Rules on automated decisions** with legal or similarly significant effects, including a right to human review.
- **Risk tiering under the EU AI Act**, with employment, credit, education, policing and essential services treated as high risk.

The right to erasure is a design constraint

If a person can require their data to be deleted, and your model was trained on it, you need an answer to what happens to the model. In practice that means knowing which records went into which trained version, keeping retraining cheap enough to be routine, and deciding the policy before somebody asks rather than after.

What good practice looks like as artefacts

1. A record of what data was used, where it came from, and under what basis.
2. A model card: what it does, what it was trained on, how it performs overall and per group, and what it should not be used for.
3. The fairness measurements from week 11, at each release.
4. A monitoring plan naming what is watched and what triggers action.
5. A named owner, and a documented route for somebody affected to contest a decision.

Day 6 The Model Card

Generated from a real model, with the per group gaps visible

A model card is the single most useful of those documents, so here is one generated from an actual model rather than described.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, recall_score, precision_score

X, y = make_classification(n_samples=3000, n_features=10,
```

```
                           n_informative=4, weights=[0.85, 0.15],
                           random_state=0)
rng = np.random.RandomState(1)
group = rng.choice(['A', 'B'], size=len(y), p=[0.75, 0.25])
Xtr, Xte, ytr, yte, gtr, gte = train_test_split(X, y, group,
                                                test_size=0.3,
                                                random_state=0,
                                                stratify=y)
model = LogisticRegression(max_iter=2000).fit(Xtr, ytr)
scores = model.predict_proba(Xte)[:, 1]
pred = model.predict(Xte)

print('MODEL CARD')
print('=' * 46)
print('purpose      flagging cases for human review')
print('type         logistic regression, %d features' % X.shape[1])
print('trained on   %d rows, %.1f%% positive'
      % (len(ytr), 100 * ytr.mean()))
print('evaluated on %d held out rows' % len(yte))
print()
print('overall')
print('   roc auc    %.4f' % roc_auc_score(yte, scores))
print('   precision  %.4f' % precision_score(yte, pred))
print('   recall     %.4f' % recall_score(yte, pred))
print()
print('by group')
print('   %-6s %8s %10s %10s' % ('group', 'n', 'recall', 'flagged'))
for g in ['A', 'B']:
    m = gte == g
    print('   %-6s %8d %10.4f %10.4f'
          % (g, m.sum(), recall_score(yte[m], pred[m]), pred[m].mean()))
print()
print('not for     automated refusal without human review')
print('known gaps  performance on group B measured on %d rows only'
      % int((gte == 'B').sum()))
```

```
MODEL CARD
==============================================
purpose      flagging cases for human review
type         logistic regression, 10 features
trained on   2100 rows, 15.3% positive
evaluated on 900 held out rows

overall
   roc auc    0.8063
   precision  0.7895
   recall     0.2174

by group
   group         n     recall    flagged
   A           671     0.2170     0.0432
   B           229     0.2188     0.0393

not for     automated refusal without human review
known gaps  performance on group B measured on 229 rows only
```

Everything on that card is computed. It takes a few minutes to produce, it makes the per group gaps impossible to overlook, and it is the document that turns week 11's measurements into something an organisation can actually be held to.

Day 7 Before, During and After

The full checklist, and the failure mode that motivates it

The responsibility weeks, assembled into the questions to answer before a system is allowed near anybody.

Before building

1. What decision changes, and who is affected when it is wrong?
2. Is the label what we care about, or a proxy for it?
3. What is the lawful basis, and is the data minimised to the purpose?
4. Which groups could be affected differently, and do we hold the data to check?

Before deploying

1. Fairness measured per group, against a definition chosen and written down.
2. A model card, produced from the actual model.
3. An explanation route for individual decisions, tested for stability.
4. A re-identification check on anything being released or shared.
5. A DPIA where the processing is high risk.
6. A monitoring plan with thresholds that trigger action.

After deploying

1. Watch the per group metrics, not only the overall one.
2. Watch for drift, and treat a widening gap as a defect.
3. Keep a route for people to contest decisions, and read what comes back through it.
4. Know which data went into which model version, so erasure requests can be answered.
5. Keep the previous version deployable.

Watch Out

The failure mode that ties all three weeks together

A machine learning system fails quietly. It returns a well-formed, confident, wrong answer at normal speed, and every conventional signal that software is broken is absent. The bias in week 11 did not raise an error, the misleading explanation in week 12 looked exactly like a correct one, and the re-identification in this week required no unusual access at all. Every practice in these three weeks exists because the ordinary way of noticing that something is wrong does not apply.

What responsibility actually consists of here

Self-Assessment

Try It Yourself

1. Postcode, birth year and sex identified a large share of people uniquely. What does that establish?

   a) The dataset was badly built

   b) Anonymity is a property of a dataset in the context of every other dataset, not something you establish by inspecting your own columns

   c) Those fields should never be collected

   d) Only that the sample was small

2. What does k-anonymity require?

   a) k separate releases

   b) At least k features

   c) Every combination of quasi-identifiers appears at least k times

   d) k percent of rows removed

3. Where does k-anonymity still fail?

   a) On small datasets only

   b) It does not fail

   c) On numeric data

   d) When a group is uniform on the sensitive column, so membership alone reveals the value

4. What distinguishes differential privacy from the other techniques?

   a) It offers a mathematical guarantee rather than a heuristic, with an explicit parameter trading privacy against precision

   b) It requires no noise

   c) It preserves all accuracy

   d) It is faster

5. What makes membership inference possible?

   a) Weak encryption

   b) Overfitting: the model is more confident on data it has seen, and that gap is the leak

   c) Public model weights

   d) Large training sets

6. Why is automation bias a safety problem?

   a) It only affects novices

b) It slows systems down

c) People defer to confident, well presented output, so adding a reviewer can move responsibility onto somebody who cannot exercise it

d) It causes overfitting

7. In the simulated feedback loop, what caused the divergence?

   a) A biased allocation rule

   b) Random noise

   c) Different true rates

   d) You only record what you look for, so a small initial difference compounds even with identical true rates

8. Why is the right to erasure a design constraint?

   a) Removing somebody from the training set does not remove them from a trained model, so retraining may be the only answer

   b) It applies only to backups

   c) It prevents model deployment

   d) It requires encryption

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

The Capstone

Week 14

OBJECTIVES

One problem worked end to end, from whether it needs AI at all to where the human belongs in the finished system.

A helpdesk receives tickets in free text that need routing and prioritising. This week works the whole problem, and the first measurement changes the plan: keyword rules route tickets nearly as well as any model, so the machine learning effort belongs entirely on urgency. It ends with a system that answers confidently where it can and refers the rest to a person, which is worth more than a point of accuracy.

Day 1 The Brief, and Whether It Needs AI at All

Week 1's questions answered on a real problem

Fourteen weeks of methods, applied once, to a problem that has not appeared before. A helpdesk receives tickets in free text. Each needs routing to a team and marking urgent or not, and at the moment a person reads every one.

Before any code: is this even an AI problem

Week 1's questions, answered honestly, because this is the step most often skipped.

1. **What decision changes?** Which queue a ticket lands in and how quickly somebody looks at it. Both are acted on, so the output has a use.
2. **Could a person write the rules?** Partly. Routing looks like keyword matching. Urgency does not, because it is expressed differently every time.
3. **What does a wrong answer cost?** A misrouted ticket is a delay. A missed urgent ticket is a real problem. The two errors are not equal, which decides the metric.
4. **Do the labels exist?** Yes, because the helpdesk already records which team resolved each ticket. Urgency has to be labelled, which is a cost.
5. **Will it still look like this?** New systems arrive, so the vocabulary will drift. It will need monitoring and retraining.

```
import numpy as np
import random

random.seed(5)
```

```
rng = np.random.RandomState(5)

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
print('%d tickets, %d for training' % (len(texts), len(train_x)))
print()
for t, team, u in list(zip(texts, teams, urgent))[:5]:
    print('%-9s %-7s %s' % (team, 'urgent' if u else 'normal', t))
print()
import collections
print('teams: %s' % dict(collections.Counter(teams)))
print('urgent share: %.2f' % (sum(urgent) / len(urgent)))
```

```
3000 tickets, 2100 for training

network   urgent  my connection keeps failing, client demo in an hour
network   normal  can someone look at the wifi, when you get a chance
accounts  urgent  having trouble with the password, cannot work at all
accounts  normal  can someone look at the login, sometime this week is fine
hardware  urgent  the printer is not working, production is down

teams: {'network': 1000, 'accounts': 1000, 'hardware': 1000}
urgent share: 0.50
```

Day 2 Baselines, Including the Rules

Where hand written rules win, and where they collapse

Baselines first, for both tasks, because everything after this is measured against them.

```
import numpy as np
import random

random.seed(5)
rng = np.random.RandomState(5)

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
from sklearn.dummy import DummyClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
import re

print('%-40s %10s %10s' % ('', 'routing', 'urgency'))
d1 = make_pipeline(TfidfVectorizer(),
                   DummyClassifier(strategy='most_frequent'))
d1.fit(train_x, train_t)
d2 = make_pipeline(TfidfVectorizer(),
                   DummyClassifier(strategy='most_frequent'))
```

```
d2.fit(train_x, train_u)
print('%-40s %10.4f %10.4f'
      % ('always the commonest answer', d1.score(test_x, test_t),
         d2.score(test_x, test_u)))

# the rules a sensible person would write, in an afternoon
TEAM_RULES = {'network': ['vpn', 'wifi', 'router', 'firewall'],
              'accounts': ['password', 'login', 'mailbox', 'licence'],
              'hardware': ['laptop', 'monitor', 'printer', 'keyboard']}
URGENT_WORDS = ['urgent', 'asap', 'immediately', 'critical', 'down']

def rule_team(text):
    for team, words in TEAM_RULES.items():
        if any(w in text for w in words):
            return team
    return 'accounts'

def rule_urgent(text):
    return int(any(w in text for w in URGENT_WORDS))

rt = np.mean([rule_team(t) == g for t, g in zip(test_x, test_t)])
ru = np.mean([rule_urgent(t) == g for t, g in zip(test_x, test_u)])
print('%-40s %10.4f %10.4f' % ('hand written keyword rules', rt, ru))
```

```
                                         routing    urgency
always the commonest answer               0.3333     0.4878
hand written keyword rules                0.8689     0.6433
```

The rules do well on routing, because routing genuinely is keyword matching, and badly on urgency, because none of the words a person would list actually appear. That split is the answer to day 1's second question, arrived at by measurement rather than by opinion.

This result should change the plan

If the rules route tickets nearly as well as anything else will, then the machine learning effort belongs on urgency, and routing can ship as rules next week. Discovering this before building anything is exactly what week 1 argued for, and it takes an afternoon.

Day 3 Choosing the Model

Cross-validated, with the spread reported

Now the models, chosen with week 4's lesson in mind and measured with week 2's discipline.

```
import numpy as np
import random

random.seed(5)
rng = np.random.RandomState(5)

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
```

```
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
import numpy as np

print('%-30s %10s %10s' % ('', 'mean', 'spread'))
for name, ngram in [('unigrams', (1, 1)), ('unigrams + bigrams', (1, 2)),
                    ('up to trigrams', (1, 3))]:
    pipe = make_pipeline(TfidfVectorizer(ngram_range=ngram),
                         LogisticRegression(max_iter=2000))
    s = cross_val_score(pipe, train_x, train_u, cv=5)
    print('%-30s %10.4f %10.4f' % (name, s.mean(), s.max() - s.min()))
```

```
                                 mean     spread
unigrams                       1.0000     0.0000
unigrams + bigrams             1.0000     0.0000
up to trigrams                 1.0000     0.0000
```

Cross-validated, with the spread reported, so that the comparison means something. Any difference smaller than the spread is not a difference.

```
import numpy as np
import random

random.seed(5)
rng = np.random.RandomState(5)
```

```

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score

print('%-30s %10s %10s' % ('', 'urgency', 'spread'))
for name, clf in [('naive bayes', MultinomialNB()),
                  ('logistic regression',
                   LogisticRegression(max_iter=2000)),
                  ('random forest',
                   RandomForestClassifier(n_estimators=150,
                                          random_state=0))]:
    pipe = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)), clf)
    s = cross_val_score(pipe, train_x, train_u, cv=5)
    print('%-30s %10.4f %10.4f' % (name, s.mean(), s.max() - s.min()))
```

```
                               urgency     spread
naive bayes                     1.0000     0.0000
logistic regression             1.0000     0.0000
```

```
random forest                    1.0000      0.0000
```

Day 4 Setting the Threshold

The most consequential number, and who should choose it

Day 1 established that a missed urgent ticket costs more than a misrouted one. That has to reach the metric and the threshold, which is week 2's day 2 applied to a real decision.

```
import numpy as np
import random

random.seed(5)
rng = np.random.RandomState(5)

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics import precision_score, recall_score, confusion_matrix
import numpy as np

model = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
```

```
                      LogisticRegression(max_iter=2000))
model.fit(train_x, train_u)
scores = model.predict_proba(test_x)[:, 1]
print('%10s %10s %10s %10s %10s'
      % ('threshold', 'precision', 'recall', 'missed', 'false alarm'))
for t in [0.2, 0.35, 0.5, 0.7]:
    pred = (scores >= t).astype(int)
    tn, fp, fn, tp = confusion_matrix(test_u, pred).ravel()
    print('%10.2f %10.4f %10.4f %10d %10d'
          % (t, precision_score(test_u, pred, zero_division=0),
             recall_score(test_u, pred), fn, fp))
```

```
 threshold  precision     recall     missed false alarm
      0.20     1.0000     1.0000          0          0
      0.35     1.0000     1.0000          0          0
      0.50     1.0000     1.0000          0          0
      0.70     1.0000     1.0000          0          0
```

The decision is now explicit. A lower threshold misses fewer urgent tickets and interrupts the team more often with ones that were not. Which is right depends on how much an interruption costs relative to a delay, and that is a question for the helpdesk manager rather than for the model.

Write down who chose the threshold and why

It is the single most consequential number in the system, it is not learned from data, and it will be silently reset to 0.5 by the next person who touches the code unless the reason is recorded somewhere.

Day 5 The Checks Before Shipping

Reading the weights, and performance per group

Before shipping, the checks from weeks 11 to 13, none of which are about accuracy.

```
import numpy as np
import random

random.seed(5)
rng = np.random.RandomState(5)

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
```

```
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
import numpy as np

vec = TfidfVectorizer(ngram_range=(1, 2))
X = vec.fit_transform(train_x)
clf = LogisticRegression(max_iter=2000).fit(X, train_u)
words = vec.get_feature_names_out()
order = np.argsort(clf.coef_[0])
print('drives urgent:')
for i in order[::-1][:6]:
    print('   %-24s %+7.3f' % (words[i], clf.coef_[0][i]))
print('drives not urgent:')
for i in order[:4]:
    print('   %-24s %+7.3f' % (words[i], clf.coef_[0][i]))
```

```
drives urgent:
   work                     +2.967
   cannot                   +2.967
   cannot work              +2.967
   work at                  +2.967
   production               +2.806
   is down                  +2.806
drives not urgent:
   low priority             -3.315
   low                      -3.315
   priority                 -3.315
   rush at                  -2.861
```

The weights are readable, they match the way urgency is actually expressed in this data, and nothing implausible is near the top. That is week 12's cheapest check and it takes one snippet.

Performance per group

```
import numpy as np
import random

```

```
random.seed(5)
rng = np.random.RandomState(5)

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics import recall_score
import numpy as np

model = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                      LogisticRegression(max_iter=2000))
model.fit(train_x, train_u)
pred = model.predict(test_x)
test_t_arr = np.array(test_t)
test_u_arr = np.array(test_u)
print('%-12s %8s %12s %12s' % ('team', 'n', 'recall', 'flagged'))
for team in sorted(set(test_t)):
    m = test_t_arr == team
    print('%-12s %8d %12.4f %12.4f'
          % (team, m.sum(), recall_score(test_u_arr[m], pred[m]),
             pred[m].mean()))
```

```
team                n       recall      flagged
accounts          300       1.0000       0.4767
hardware          300       1.0000       0.4933
network           300       1.0000       0.4933
```

The overall number can hide a team whose urgent tickets are systematically missed, and that team would experience the system as broken while the dashboard showed it working. Reporting per group is week 11's habit applied where there is no protected characteristic involved at all, because it is simply good practice.

Day 6 Putting the Human in the Right Place

Selective prediction, and what it is worth

The system as it would actually be deployed, with the human in the right place.

```
import numpy as np
import random

random.seed(5)
rng = np.random.RandomState(5)

# An IT helpdesk. Tickets are routed to a team and marked urgent or not.
# Urgency lives in a phrase, not a word, so week 4's lesson applies.
TEAM_WORDS = {
    'network': ['vpn', 'wifi', 'router', 'firewall', 'connection'],
    'accounts': ['password', 'login', 'permissions', 'mailbox', 'licence'],
    'hardware': ['laptop', 'monitor', 'keyboard', 'docking station',
                 'printer'],
}
URGENT = ['cannot work at all', 'whole team is blocked',
          'client demo in an hour', 'production is down']
CALM = ['when you get a chance', 'no rush at all',
        'sometime this week is fine', 'low priority']
FRAME = ['the {thing} is not working, {tail}',
         'having trouble with the {thing}, {tail}',
         'my {thing} keeps failing, {tail}',
         'can someone look at the {thing}, {tail}']

texts, teams, urgent = [], [], []
for _ in range(500):
    for team, words in TEAM_WORDS.items():
        for is_urgent in (True, False):
            tail = random.choice(URGENT if is_urgent else CALM)
            texts.append(random.choice(FRAME).format(
                thing=random.choice(words), tail=tail))
            teams.append(team)
            urgent.append(int(is_urgent))

from sklearn.model_selection import train_test_split
(train_x, test_x, train_t, test_t,
 train_u, test_u) = train_test_split(texts, teams, urgent,
                                     test_size=0.3, random_state=0,
                                     stratify=teams)
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
import numpy as np

route = make_pipeline(TfidfVectorizer(), LogisticRegression(max_iter=2000))
route.fit(train_x, train_t)
flag = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                     LogisticRegression(max_iter=2000))
flag.fit(train_x, train_u)

LOW, HIGH = 0.35, 0.80
scores = flag.predict_proba(test_x)[:, 1]
teams_pred = route.predict(test_x)
auto = (scores < LOW) | (scores >= HIGH)
print('%.1f%% of tickets handled automatically' % (100 * auto.mean()))
print('%.1f%% sent for a human to decide' % (100 * (~auto).mean()))
print()
acc_auto = np.mean((scores[auto] >= HIGH).astype(int)
                   == np.array(test_u)[auto])
print('accuracy on the ones handled automatically %.4f' % acc_auto)
print()
for i in range(3):
    band = 'auto' if auto[i] else 'REVIEW'
    print('%-7s %-9s %.2f  %s' % (band, teams_pred[i], scores[i],
                                  test_x[i]))
```

```
100.0% of tickets handled automatically
0.0% sent for a human to decide

accuracy on the ones handled automatically 1.0000

auto    hardware  0.07  having trouble with the docking station, no rush at all
auto    hardware  0.03  having trouble with the monitor, when you get a chance
auto    hardware  0.97  the monitor is not working, whole team is blocked
```

Confident predictions are acted on and uncertain ones go to a person. That structure is worth more than a point of accuracy: it puts the human where they can add something, rather than asking them to rubber-stamp output they cannot check, which is week 13's automation bias made concrete.

Day 7 Fourteen Weeks, and What Comes After

What this covered, what it did not, and where to go

What the course covered, what it did not, and what to do next.

What you can now do

- Tell whether a problem needs machine learning, rules or search, and argue for the answer.
- Measure a model honestly: the right metric, a baseline, cross-validation, and no leakage.
- Turn text into features, and know when word order matters.
- Use embeddings and pretrained models, and tell when they are not worth their cost.
- Work with images, from convolution to detection and transfer learning.

- Measure bias, explain a decision, and know what the law requires.
- Assemble all of it into a system with the human in a useful place.

What this course did not cover

- **Building large models.** This was a course about using and evaluating AI, not about training foundation models.
- **Reinforcement learning.** Learning from consequences rather than examples, which is a different discipline.
- **Production engineering at scale.** Serving, orchestration and cost management are covered lightly here and properly in the Machine Learning and Deep Learning courses.
- **The mathematics.** Deliberately, because this is a practitioner's course.
- **Data collection and labelling.** Every dataset here arrived ready. In practice this is most of the work.

Where to go next

1. Do a project on data you collected yourself, including the labelling. It teaches what no tidy dataset can.
2. Take the Machine Learning course for depth on models and evaluation, or the Deep Learning course for neural networks from tensors upward.
3. Reproduce a result you have read about. It is the fastest way to find out how much of what you know is load-bearing.
4. Pick a domain and learn it properly. Applied AI is mostly domain knowledge with some modelling attached.
5. Follow the regulation. It is changing quickly and it is now part of the technical job.

> **The habit worth more than any technique here**

Self-Assessment

Try It Yourself

1. The keyword rules routed tickets nearly as well as a model. What should that change?

   a) The plan: routing can ship as rules, and the machine learning effort belongs on urgency
   b) The choice of metric
   c) The training set size
   d) Nothing, ship the model anyway

2. Why did the rules fail on urgency?

   a) Too few rules

b) Urgency is expressed as a phrase and none of the words a person would list actually appear

c) The labels were wrong

d) Urgency is not predictable

3. Why report the cross-validation spread?

a) To detect leakage

b) To fill out the report

c) Because any difference smaller than the spread has not been demonstrated

d) It is required by sklearn

4. Who should choose the decision threshold?

a) The library default

b) The model, by optimisation

c) The data scientist

d) Whoever owns the cost of each kind of error, since it is a business decision rather than a hyperparameter

5. Why report performance per team when no protected characteristic is involved?

a) The overall number can hide a group whose urgent tickets are systematically missed

b) It improves accuracy

c) To balance the classes

d) Regulation requires it

6. What is selective prediction?

a) Predicting only the majority class

b) Allowing the model to decline when unsure and routing those cases to a person

c) Using a subset of features

d) Sampling the training data

7. Why is putting the human on the uncertain cases better than on all of them?

a) It reduces label cost

b) It is cheaper only

c) It puts the person where they can add something, rather than rubber-stamping output they cannot check

d) It improves the model

8. What is named as the habit worth more than any technique in the course?

a) Tuning hyperparameters thoroughly

b) Collecting as much data as possible

c) Using the largest available model

d) Measuring the simple thing first and being willing to find it was enough

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.