



CONTENTS

1	Decision Trees for Classification	3
1.1	Measuring Impurity	3
1.2	Choosing a Split	4
1.3	A Tree in Python	5
1.4	Stopping and Overfitting	7
2	Bagging and Random Forests	11
2.1	The Wisdom of Crowds	11
2.2	Bootstrap Aggregating	12
2.2.1	Why Bootstrap Samples Repeat	12
2.3	Random Forests	13
2.4	Out-of-Bag Error	13
2.5	A Forest in Python	13
2.6	What You Give Up	15
3	Boosting	19
3.1	Weak Learners	19
3.2	AdaBoost	19
3.3	AdaBoost on Penguins	20
3.4	Gradient Boosting	22
3.5	Cautions	25
4	Generalized Linear Models	29
4.1	Beyond the Normal Distribution	29

4.2	Three Components	29
4.3	Link Functions	30
4.4	Counts and the Poisson Distribution	30
4.5	Fitting a GLM	31
4.6	Poisson Regression in Python	32
5	Clustering with k-Means	37
5.1	Unsupervised Learning	37
5.2	The Algorithm	37
5.3	Starting Well	38
5.4	Clustering Penguins	38
5.5	Did It Find the Species?	40
5.6	Choosing k	41
A	Answers to Exercises	45
	Index	51

Decision Trees for Classification

Remember the game Twenty Questions? “Is it alive?” “Is it bigger than a breadbox?” Good players ask questions that split the possibilities roughly in half, and they adapt each question to the answers so far. A *decision tree* classifies the same way: it asks a series of yes-or-no questions about the features, and each answer leads to the next question or to a final answer.

Decision trees are popular because people can read them. A doctor can follow a tree of symptoms; a bank can explain why a loan was refused. Here is the tree that our program will learn for penguin species:

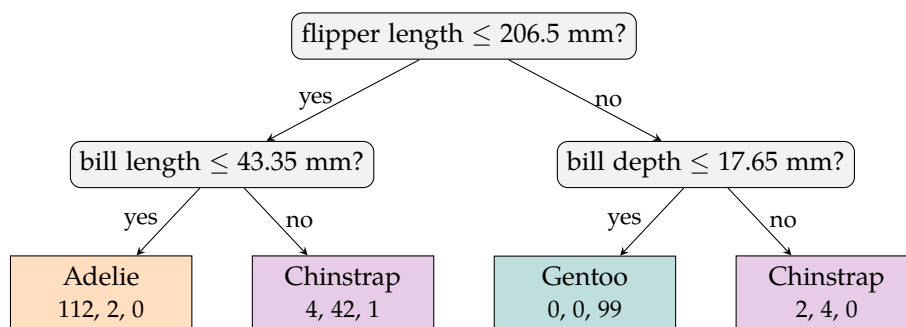


Figure 1.1: A depth-2 decision tree for penguin species, learned from 266 penguins. Each leaf shows how many Adelie, Chinstrap, and Gentoo training penguins ended up there.

To classify a penguin, start at the top and follow the answers. A penguin with 215 mm flippers and a 15 mm deep bill goes right, then left: Gentoo. The *internal nodes* ask questions, and the *leaves* give answers; each leaf predicts the most common class among the training penguins that landed there.

1.1 Measuring Impurity

How does the computer choose a question? A good question produces groups that are each mostly one class, or *pure*. We need a number for how mixed a group is.

The *Gini impurity* of a node is

$$G = 1 - \sum_i p_i^2$$

where p_i is the fraction of the node’s points in class i . A pure node has $G = 0$. A node

split evenly between two classes has $G = 1 - (0.5^2 + 0.5^2) = 0.5$. One way to read it: G is the chance that two points drawn at random from the node (with replacement) belong to different classes.

A popular alternative is the *entropy*, measured in bits:

$$H = - \sum_i p_i \log_2 p_i$$

The decrease in entropy from a split is called the *information gain*. Both measures are 0 for a pure node and largest for an even mix, and in practice they almost always pick the same splits.

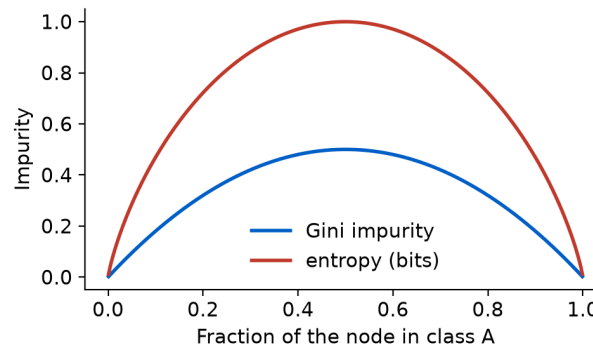


Figure 1.2: Gini impurity and entropy for a node with two classes.

Fraction in class A	Gini	Entropy (bits)
0.5	0.50	1.00
0.6	0.48	0.97
0.8	0.32	0.72
0.9	0.18	0.47
1.0	0	0

1.2 Choosing a Split

At each node, the tree tries every feature and every threshold and keeps the split with the lowest *weighted* impurity of the two children, where each child is weighted by its share of the points.

Example: A node holds 6 points of class A and 4 of class B. A split sends 4 A and 1 B to the left and 2 A and 3 B to the right. Does it help?

Solution: Before the split, $G = 1 - (0.6^2 + 0.4^2) = 0.48$. The left child has $G = 1 - (0.8^2 + 0.2^2) = 0.32$, and the right has $G = 1 - (0.4^2 + 0.6^2) = 0.48$. Each child has 5 of the 10

points, so the weighted impurity is

$$\frac{5}{10}(0.32) + \frac{5}{10}(0.48) = 0.40$$

The split lowers the impurity from 0.48 to 0.40.

For the penguins, the 266 training penguins (118 Adelie, 48 Chinstrap, 100 Gentoo) have $G = 1 - [(118/266)^2 + (48/266)^2 + (100/266)^2] \approx 0.629$. The best first question, “flipper length ≤ 206.5 mm?”, sends 161 penguins (116, 44, 1) left, with $G \approx 0.406$, and 105 (2, 4, 99) right, with $G \approx 0.109$. The weighted impurity is

$$\frac{161}{266}(0.406) + \frac{105}{266}(0.109) \approx 0.289$$

No other feature and threshold does better. The left group still mixes Adelies and Chinstraps, so the tree asks a second question there, and so on.

Which thresholds are worth trying? Only the midpoints between consecutive values in the data: any threshold between 206 and 207 splits the penguins the same way, so 206.5 stands for all of them. This whole procedure, always taking the best split right now without looking ahead, is called a *greedy* algorithm.

1.3 A Tree in Python

Here is `tree.py`. A tree is stored as nested dictionaries: an internal node holds a feature, a threshold, and two subtrees; a leaf holds a class. The function `grow` calls itself to build each subtree, which is called *recursion*.

```

1  import numpy as np
2  from penguins import load_penguins
3
4  data = load_penguins()
5  names, y = np.unique(data["species"], return_inverse=True)
6  features = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]
7  X = np.column_stack([data[f] for f in features])
8  order = np.random.default_rng(1).permutation(len(y))
9  dev, test = order[:266], order[266:]
10
11 def gini(labels):
12     p = np.bincount(labels, minlength=3) / len(labels)
13     return 1 - np.sum(p**2)
14
15 def best_split(X, y):
16     """Try every feature and every threshold; return the lowest weighted Gini."""
17     best = (gini(y), None, None) # (impurity, feature, threshold)
18     for j in range(X.shape[1]):
19         values = np.unique(X[:, j])

```

```

20         for t in (values[:-1] + values[1:]) / 2: # midpoints between values
21             left = X[:, j] <= t
22             g = (left.sum() * gini(y[left]) + (~left).sum() * gini(y[~left])) /
                ↪ len(y)
23             if g < best[0]:
24                 best = (g, j, t)
25         return best
26
27 def grow(X, y, depth, max_depth):
28     majority = np.bincount(y, minlength=3).argmax()
29     if depth == max_depth or gini(y) == 0:
30         return {"leaf": majority, "counts": np.bincount(y, minlength=3)}
31     g, j, t = best_split(X, y)
32     if j is None: # no split helps
33         return {"leaf": majority, "counts": np.bincount(y, minlength=3)}
34     left = X[:, j] <= t
35     return {"feature": j, "threshold": t,
36           "left": grow(X[left], y[left], depth + 1, max_depth),
37           "right": grow(X[~left], y[~left], depth + 1, max_depth)}
38
39 def predict_one(node, x):
40     while "leaf" not in node:
41         node = node["left"] if x[node["feature"]] <= node["threshold"] else
            ↪ node["right"]
42     return node["leaf"]
43
44 def predict(tree, X):
45     return np.array([predict_one(tree, x) for x in X])
46
47 def show(node, indent="", label=""):
48     if "leaf" in node:
49         print(f"{indent}{label}{names[node['leaf']] {node['counts']}")
50     else:
51         print(f"{indent}{label}{features[node['feature']] <=
            ↪ {node['threshold']:.2f}?"")
52         show(node["left"], indent + "    ", "yes: ")
53         show(node["right"], indent + "    ", "no: ")
54
55 tree = grow(X[dev], y[dev], 0, max_depth=2)
56 show(tree)
57 for depth in [1, 2, 3, 4, 6, 10]:
58     tree = grow(X[dev], y[dev], 0, depth)
59     train_acc = np.mean(predict(tree, X[dev]) == y[dev])
60     test_acc = np.mean(predict(tree, X[test]) == y[test])
61     print(f"max depth {depth:2d}: training {train_acc:.3f}, test {test_acc:.3f}")

```

```

flipper_length_mm <= 206.50?
yes: bill_length_mm <= 43.35?
    yes: Adelie [112  2  0]
    no: Chinstrap [ 4 42 1]
no: bill_depth_mm <= 17.65?
    yes: Gentoo [ 0  0 99]

```

```
no: Chinstrap [2 4 0]
max depth 1: training 0.808, test 0.701
max depth 2: training 0.966, test 0.955
max depth 3: training 0.974, test 0.955
max depth 4: training 0.992, test 0.955
max depth 6: training 0.996, test 0.955
max depth 10: training 1.000, test 0.955
```

Notice that we did not standardize anything. A tree only ever asks whether one feature is above or below a threshold, so the units do not matter. That is a real advantage over k-nearest neighbors.

1.4 Stopping and Overfitting

Keep splitting and the tree will eventually put every training penguin in a leaf of its own class: training accuracy reaches 100% at depth 10. But look at Figure 1.3. The deep tree carves out thin slivers to capture individual penguins, which is memorization, not learning.

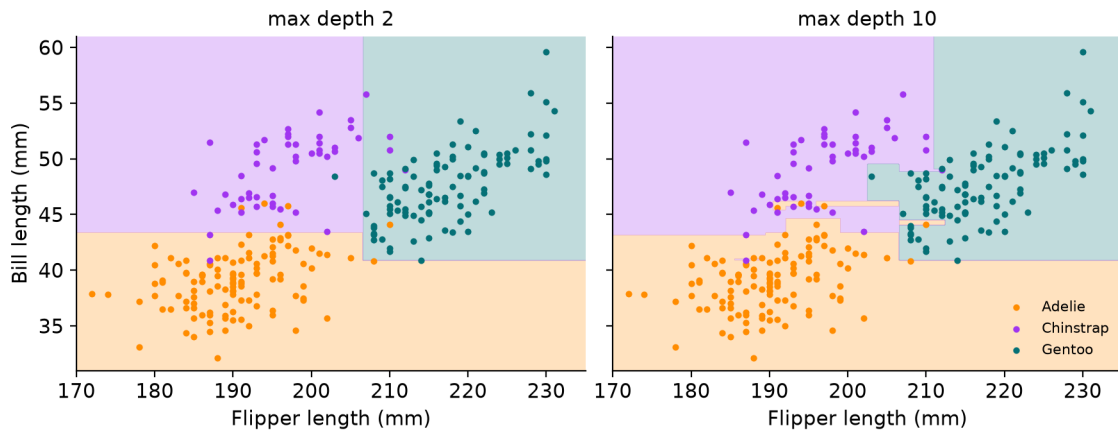


Figure 1.3: Trees grown on flipper length and bill length only. Every boundary is horizontal or vertical, because each question tests one feature.

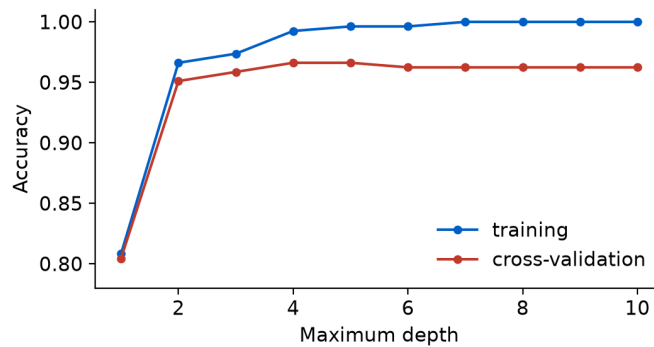


Figure 1.4: Training and 5-fold cross-validation accuracy for trees of different depths, using all four features.

On the 67 test penguins, every depth from 2 to 10 scores the same, 95.5%; cross-validation peaks at depth 4 or 5 and drifts down a little after that. The extra depth buys almost nothing, and a depth-2 tree you can draw on an index card is just as good. To control overfitting, limit the depth, require a minimum number of points in each leaf, or grow a big tree and then *prune* the branches that do not help on validation data. As always, choose these limits by validation, not by training accuracy.

Strength	Weakness
easy to read and explain	a small change in the data can produce a very different tree
no scaling needed	boundaries are always parallel to the axes
handles numbers and categories	a single tree is rarely the most accurate model

The last two weaknesses have a famous fix: grow many trees and let them vote. That is the next chapter.

Exercise 1 Compute impurity

A node holds 3 points of class A and 1 point of class B. What is its Gini impurity?

Working Space

Answer on Page 45

Exercise 2 Is the split better?

A node with $G = 0.48$ is split into two children of equal size with $G = 0.2$ and $G = 0.3$. Is the split an improvement?

*Working Space**Answer on Page 45***Exercise 3** Pick the better split

Using the node from the worked example (6 A and 4 B), a second candidate split sends 5 A and 0 B to the left and 1 A and 4 B to the right. Which split should the tree choose: this one or the one in the worked example?

*Working Space**Answer on Page 45***Exercise 4** Information gain

Find the entropy of the node with 6 A and 4 B, and the information gain of the split in the worked example (4 A and 1

Working Space

B left, 2 A and 3 B right).

Answer on Page 45

Exercise 5 Follow the tree

Use the tree in Figure 1.1 to classify three penguins: (a) flipper 190 mm, bill length 48 mm, bill depth 18 mm; (b) flipper 220 mm, bill length 47 mm, bill depth 14.5 mm; (c) flipper 210 mm, bill length 50 mm, bill depth 19 mm.

Working Space

Answer on Page 45

Exercise 6 No flippers

Remove "flipper_length_mm" from the list of features in tree.py. What question does the tree ask first now? How do the test accuracies compare?

Working Space

Answer on Page 46

Bagging and Random Forests

At a county fair, hundreds of people guess the weight of an ox. Most guesses are off by a lot, but in a famous 1906 contest, the *median* of the guesses landed within 1% of the true weight, closer than nearly every individual guess. When many people make mistakes that are not the same mistakes, the mistakes cancel.

The last chapter ended with a weakness of decision trees: change the training data a little, and you may get a very different tree. This chapter turns that weakness into a strength. Grow many different trees and let them vote.

2.1 The Wisdom of Crowds

Suppose you have a panel of classifiers, each right 70% of the time, and their mistakes are *independent*. Take a majority vote. The number of voters who are right follows a binomial distribution, so the chance that the majority is right is a binomial tail probability:

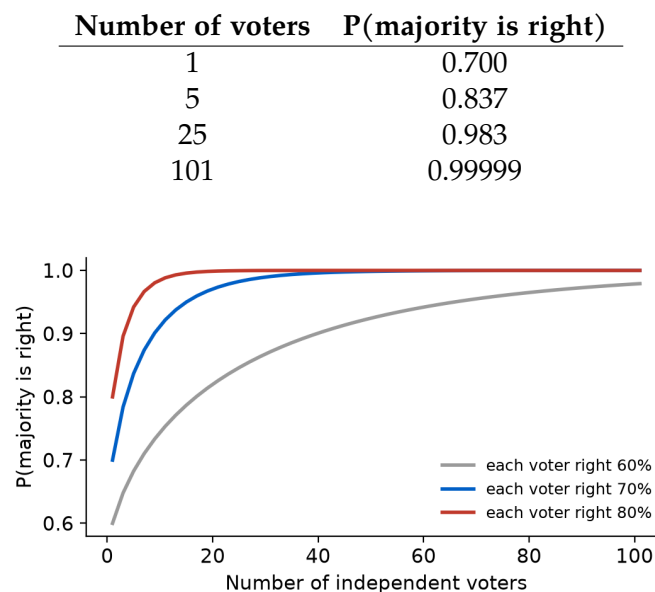


Figure 2.1: The chance that a majority of independent voters is right, for voters of three different accuracies.

The catch is the word “independent.” If all 101 voters are copies of the same classifier,

they make the same mistakes, and the vote is exactly as good as one of them. An ensemble helps only as much as its members *disagree*. The two methods in this chapter are ways to build trees that disagree.

2.2 Bootstrap Aggregating

Bagging, short for bootstrap aggregating, trains each tree on a different *bootstrap sample*: n points drawn from the original n points *with replacement*, the same resampling you used for bootstrap intervals in the Simulation-Based Inference chapter. For classification, the trees vote; for regression, their predictions are averaged.

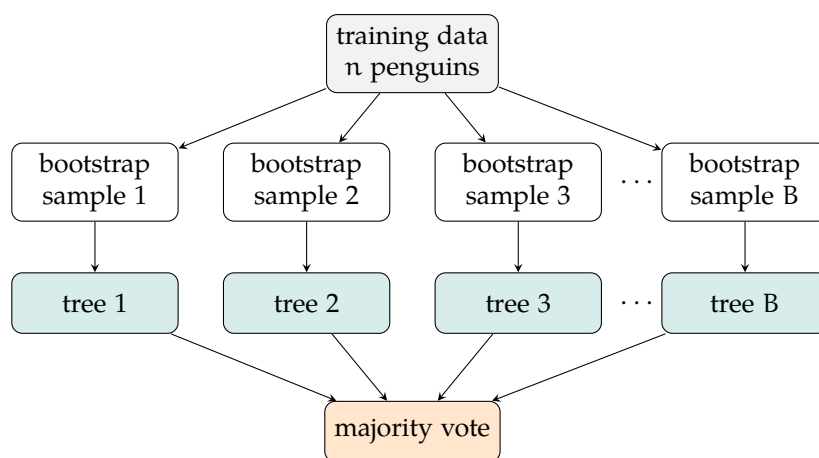


Figure 2.2: Bagging: each tree learns from its own bootstrap sample, and the trees vote.

2.2.1 Why Bootstrap Samples Repeat

Sampling with replacement means some points appear more than once and others not at all. A given point is missed by one draw with probability $1 - \frac{1}{n}$, so it is missed by all n draws with probability $(1 - \frac{1}{n})^n$. For large n , this approaches $\frac{1}{e} \approx 0.368$. So each bootstrap sample contains about $1 - \frac{1}{e} \approx 63.2\%$ of the distinct original points.

Example: For $n = 10$ points, a given point is missed by all 10 draws with probability $(0.9)^{10} \approx 0.349$, so about 65% of the distinct points appear in each bootstrap sample. For our 266 development penguins, the miss probability is $(1 - \frac{1}{266})^{266} \approx 0.367$, already very close to $\frac{1}{e}$.

2.3 Random Forests

Bagged trees still tend to look alike. If one feature is a very strong predictor, almost every tree puts it at the top, and the trees make similar mistakes. A *random forest* adds one more source of randomness: each time a tree chooses a split, it may consider only a random subset of m features. Hiding the best feature from some splits forces the trees to find other patterns, so they disagree more. A common choice is $m = \sqrt{p}$ for classification, where p is the number of features, and $m = p/3$ for regression.

Random forests, introduced by Leo Breiman in 2001, are one of the most reliable methods in machine learning: they need little tuning, no scaling, and rarely overfit badly as you add trees.

2.4 Out-of-Bag Error

About a third of the points are left out of each bootstrap sample. So each point can be predicted by the trees that never saw it, about a third of the forest. The accuracy of these *out-of-bag* (OOB) predictions is an honest estimate of how the forest does on new data, and it comes free with training. You can tune a random forest without setting aside a separate validation set. (Keep the test set anyway, for the final report.)

2.5 A Forest in Python

The file `trees.py` contains the tree code from the last chapter, with one change: `grow` takes an optional `max_features`, and when it is set, each split looks at only that many randomly chosen features:

```

1 columns = range(X.shape[1])
2 if max_features is not None:                # random forest: a random subset
3     columns = rng.choice(X.shape[1], size=max_features, replace=False)
4 g, j, t = best_split(X, y, columns)

```

Then `forest.py` grows 200 trees for each method, keeping track of the test votes and the out-of-bag votes as it goes:

```

1 import numpy as np
2 from penguins import load_penguins
3 from trees import grow, predict
4
5 data = load_penguins()
6 names, y = np.unique(data["species"], return_inverse=True)
7 features = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]
8 X = np.column_stack([data[f] for f in features])

```

```

9 order = np.random.default_rng(1).permutation(len(y))
10 dev, test = order[:266], order[266:]
11 X_dev, y_dev, n = X[dev], y[dev], len(dev)
12
13 def forest(n_trees, max_features, seed=0):
14     rng = np.random.default_rng(seed)
15     test_votes = np.zeros((len(test), 3))
16     oob_votes = np.zeros((n, 3))
17     single = [] # accuracy of each tree alone
18     for b in range(1, n_trees + 1):
19         sample = rng.integers(0, n, size=n) # bootstrap: n draws with
20         ↪ replacement
21         tree = grow(X_dev[sample], y_dev[sample], max_features=max_features,
22         ↪ rng=rng)
23         guess = predict(tree, X[test])
24         test_votes[np.arange(len(test)), guess] += 1
25         single.append(np.mean(guess == y[test]))
26         out = np.setdiff1d(np.arange(n), sample) # penguins this tree never saw
27         oob_votes[out, predict(tree, X_dev[out])] += 1
28         if b in (1, 10, 50, 200):
29             seen = oob_votes.sum(axis=1) > 0
30             oob = np.mean(oob_votes[seen].argmax(axis=1) == y_dev[seen])
31             acc = np.mean(test_votes.argmax(axis=1) == y[test])
32             print(f" {b:3d} trees: out-of-bag {oob:.3f}, test {acc:.3f}")
33         print(f" single trees on the test set: {min(single):.3f} to
34         ↪ {max(single):.3f},"
35         ↪ f" average {np.mean(single):.3f}")
36
37 print("Bagging (every split sees all 4 features)")
38 forest(200, max_features=None)
39 print("Random forest (every split sees 2 random features)")
40 forest(200, max_features=2)

```

```

Bagging (every split sees all 4 features)
  1 trees: out-of-bag 0.918, test 0.955
 10 trees: out-of-bag 0.958, test 0.970
 50 trees: out-of-bag 0.970, test 0.955
200 trees: out-of-bag 0.962, test 0.955
single trees on the test set: 0.866 to 0.970, average 0.946
Random forest (every split sees 2 random features)
  1 trees: out-of-bag 0.918, test 0.955
 10 trees: out-of-bag 0.954, test 0.955
 50 trees: out-of-bag 0.974, test 0.970
200 trees: out-of-bag 0.970, test 0.970
single trees on the test set: 0.761 to 1.000, average 0.916

```

The line `test_votes[np.arange(len(test)), guess] += 1` adds one vote in row *i*, column `guess[i]`, for every test penguin *i* at once. `np.setdiff1d` finds the penguins that are not in the bootstrap sample.

Method	Single trees (average)	OOB, 200 trees	Test, 200 trees
one tree, all the data	—	—	0.955
bagging	0.946	0.962	0.955
random forest	0.916	0.970	0.970

Look at the single-tree column. Each random-forest tree is *worse* on its own than a bagged tree (91.6% against 94.6%), because it was sometimes denied the best feature. Their test accuracies range from 76% to 100%; they really do disagree. Yet the forest as a whole is the best of the three. That is the wisdom-of-crowds effect: weaker but more independent voters make a stronger majority.

The differences here are small, because penguin species are easy to tell apart. (A single extra correct test penguin is 1.5 percentage points.) On harder problems with many features, the gap between one tree and a forest is usually much larger.

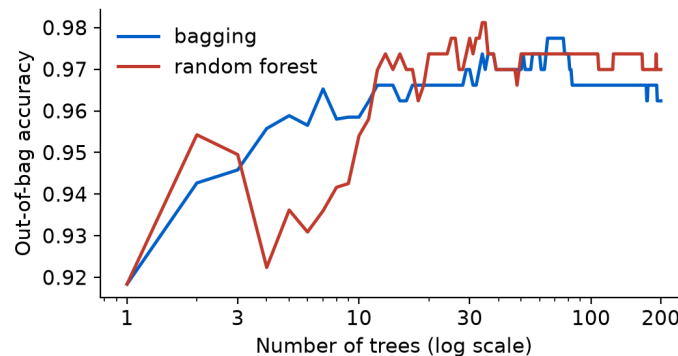


Figure 2.3: Out-of-bag accuracy as trees are added. It is noisy at first and then settles; adding more trees does not cause overfitting.

2.6 What You Give Up

A forest of 200 trees cannot be drawn on an index card. You lose the main attraction of a single tree, its readability. Forests can still tell you which features matter most: add up how much each feature reduces the impurity across all the splits in all the trees, a quantity called *feature importance*. Forests are also slower: 200 trees take 200 times as long to train and to predict.

Exercise 7 Probability a point is left out

A data set has 5 points. What is the probability that a particular point is not chosen in any of the 5 draws of a boot-

Working Space

strap sample?

Answer on Page 46

Exercise 8 **Choosing m**

A classification problem has 16 features. How many features should each split consider in a random forest?

Working Space

Answer on Page 46

Exercise 9 **Three voters**

Three independent classifiers are each right with probability 0.8. What is the probability that their majority vote is right?

Working Space

Answer on Page 46

Exercise 10 **Copies do not help**

Explain why a forest of 100 trees grown on the *same* data, with no bootstrap sampling and no random features, would be

Working Space

no better than one tree.

Answer on Page 46

Exercise 11 Simulate a bootstrap

Use `rng.integers(0, n, size=n)` to draw 1000 bootstrap samples of size $n = 266$. For each, compute the fraction of distinct points with `len(np.unique(sample)) / n`. What is the average? Compare with $1 - (1 - \frac{1}{266})^{266}$.

Working Space

Answer on Page 46

Exercise 12 One feature at a time

Run `forest.py` with `max_features=1`, so each split considers a single random feature. How do the average single-tree accuracy and the forest's test accuracy change? Why?

Working Space

Answer on Page 47

Boosting

After a practice test, a good student does not reread the whole textbook. They look at the problems they missed and study those. Then they take another practice test, look at what they missed *this* time, and repeat.

Boosting trains a sequence of models the same way. Each new model concentrates on the examples the models before it got wrong, and the final prediction combines them all. Random forests build many trees independently and let them vote; boosting builds them one after another, each correcting the last.

3.1 Weak Learners

Boosting works with *weak learners*: models that are only a little better than guessing. The usual choice is a *decision stump*, a decision tree with a single question, like “body mass > 3712.5 g?” A stump alone is a poor classifier. The surprise, proved in the 1990s, is that a weighted vote of many weak learners can be as strong as you like.

	Random forest	Boosting
Trees built	independently	one after another
Each tree	deep, overfits on its own	small, underfits on its own
Main effect	averages away variance	chips away at bias
Too many trees	harmless	can overfit

3.2 AdaBoost

AdaBoost (adaptive boosting), invented by Yoav Freund and Robert Schapire in 1995, works with labels $y_i \in \{-1, +1\}$. It keeps a weight w_i for each training point.

1. Start with equal weights, $w_i = \frac{1}{n}$.
2. For $t = 1, \dots, T$: train a weak learner h_t on the weighted data, compute its weighted error $\varepsilon_t = \sum_{i: h_t(x_i) \neq y_i} w_i$, and give it a vote of strength

$$\alpha_t = \frac{1}{2} \ln \frac{1 - \varepsilon_t}{\varepsilon_t}$$

3. Increase the weights of the misclassified points and decrease the others,

$$w_i \leftarrow w_i e^{-\alpha_t y_i h_t(x_i)}$$

then divide by the total so the weights sum to 1.

The final classifier is the sign of the weighted vote, $H(x) = \text{sign}(\sum_t \alpha_t h_t(x))$.

The exponent does the bookkeeping. When h_t is right, $y_i h_t(x_i) = +1$ and the weight shrinks by the factor $e^{-\alpha_t}$; when it is wrong, $y_i h_t(x_i) = -1$ and the weight grows by e^{α_t} . A learner with a small error gets a large α_t , a strong vote.

Example: Four points have weights $\frac{1}{4}$ each. A weak learner misclassifies only the fourth point. Find α_1 and the new weights.

Solution: $\varepsilon_1 = \frac{1}{4}$, so

$$\alpha_1 = \frac{1}{2} \ln \frac{3/4}{1/4} = \frac{1}{2} \ln 3 \approx 0.549$$

The three correct points get weight $\frac{1}{4} e^{-0.549} \approx 0.144$ before normalizing, and the misclassified point gets $\frac{1}{4} e^{0.549} \approx 0.433$. The total is $3(0.144) + 0.433 = 0.866$, so after normalizing, the misclassified point has weight 0.5 and each correct point has weight $\frac{1}{6} \approx 0.167$. Half of the next learner's attention goes to the one point that was missed.

3.3 AdaBoost on Penguins

Male and female penguins are hard to tell apart, especially across three species of different sizes. Here is `adaboost.py`, which boosts stumps to predict a penguin's sex from its four measurements, training on 233 penguins and testing on 100:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 y = np.where(data["sex"] == "male", 1, -1)           # labels are +1 and -1
6 features = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]
7 X = np.column_stack([data[f] for f in features])
8 order = np.random.default_rng(1).permutation(len(y))
9 train, test = order[:233], order[233:]
10
11 def best_stump(X, y, w):
12     """The one-question rule with the smallest weighted error."""
13     best = (np.inf, None, None, None)                # (error, feature, threshold,
14     ↪ sign)
15     for j in range(X.shape[1]):
16         values = np.unique(X[:, j])
17         for t in (values[:-1] + values[1:]) / 2:
```

```

17         guess = np.where(X[:, j] > t, 1, -1)
18         for sign in (1, -1):                                # "big means male" or the
19             ↪ reverse
20             error = np.sum(w[sign * guess != y])
21             if error < best[0]:
22                 best = (error, j, t, sign)
23         return best
24
25 def stump_predict(stump, X):
26     _, j, t, sign = stump
27     return sign * np.where(X[:, j] > t, 1, -1)
28
29 w = np.full(len(train), 1 / len(train))
30 score_train, score_test = np.zeros(len(train)), np.zeros(len(test))
31 for t in range(1, 201):
32     stump = best_stump(X[train], y[train], w)
33     error = stump[0]
34     alpha = 0.5 * np.log((1 - error) / error)
35     h = stump_predict(stump, X[train])
36     w = w * np.exp(-alpha * y[train] * h)
37     w = w / w.sum()
38     score_train += alpha * h
39     score_test += alpha * stump_predict(stump, X[test])
40     if t <= 3:
41         print(f"round {t}: {features[stump[1]]} > {stump[2]:.2f} means "
42               f"'male' if stump[3] == 1 else 'female'; error {error:.3f}, alpha "
43               ↪ {alpha:.3f}")
44     if t in (1, 10, 50, 200):
45         acc_train = np.mean(np.sign(score_train) == y[train])
46         acc_test = np.mean(np.sign(score_test) == y[test])
47         print(f"{t:3d} stumps: training {acc_train:.3f}, test {acc_test:.3f}")

```

```

round 1: body_mass_g > 3712.50 means male; error 0.270, alpha 0.496
  1 stumps: training 0.730, test 0.710
round 2: bill_depth_mm > 17.95 means male; error 0.236, alpha 0.588
round 3: body_mass_g > 5025.00 means male; error 0.258, alpha 0.529
 10 stumps: training 0.936, test 0.880
 50 stumps: training 0.970, test 0.890
200 stumps: training 1.000, test 0.860

```

The first stump, “heavier than 3712.5 g means male,” is right 73% of the time; its mistakes are mostly heavy Gentoo females and light Adelie males. The weights shift toward those penguins, and the second stump, on bill depth, is chosen because it handles them better. The third stump asks about mass again, but with a much higher threshold, aimed at the Gentoos. The running totals score_train and score_test hold $\sum_t \alpha_t h_t(x)$ for every penguin, so checking accuracy after any round is just a matter of taking signs.

Ten stumps already get 88% of the test penguins right. But look at the end: with 200 stumps, the training accuracy is a perfect 100%, and the test accuracy has slipped to 86%. Boosting kept focusing harder and harder on the few training penguins it could not get

right, and eventually it memorized them.

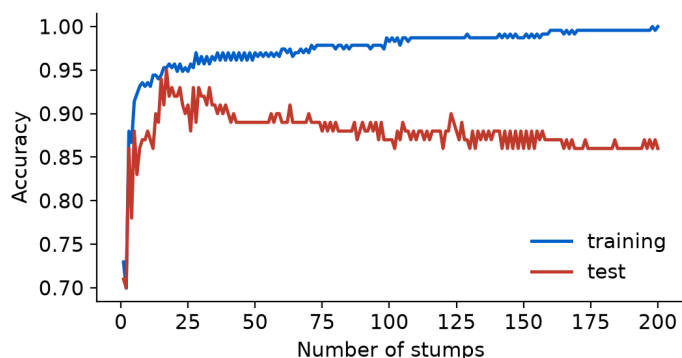


Figure 3.1: AdaBoost accuracy as stumps are added.

The test curve in Figure 3.1 peaks early and then sags. You might be tempted to pick the number of stumps at its peak, but that would be choosing a setting by looking at the test set, which turns the test set into a validation set and its score into an overestimate. Choose the number of rounds with a separate validation set or cross-validation.

3.4 Gradient Boosting

AdaBoost reweights the points. *Gradient boosting* takes a different route that works for regression, too: each new tree is trained to predict the *errors* of the ensemble so far, and its predictions are added on, scaled down by a *learning rate*.

Example: Three houses sell for 3, 5, and 10 (in hundreds of thousands of dollars). Start by predicting the mean, 6, for every house. The residuals are -3 , -1 , and 4 . Suppose the first tree learns these residuals exactly. With a learning rate of 0.5 , each prediction moves halfway toward its target: $6 + 0.5(-3) = 4.5$, $6 + 0.5(-1) = 5.5$, and $6 + 0.5(4) = 8$. The new residuals are -1.5 , -0.5 , and 2 , half as big. The next tree fits these, and so on.

Why “gradient”? For squared error, $L = \frac{1}{2}(y - \hat{y})^2$, the negative gradient with respect to the prediction, $-\partial L / \partial \hat{y} = y - \hat{y}$, is exactly the residual. So fitting each tree to the residuals is gradient descent, as in the Linear Regression and Gradient Descent chapter, except that each step adds a whole tree instead of nudging a few parameters. Other losses (such as log loss, for classification) give other “residuals,” and the same machinery works.

Figure 3.2 shows gradient boosting with stumps on a single feature, flipper length. Each stump adds one small step, and the steps pile up into a staircase that follows the data.

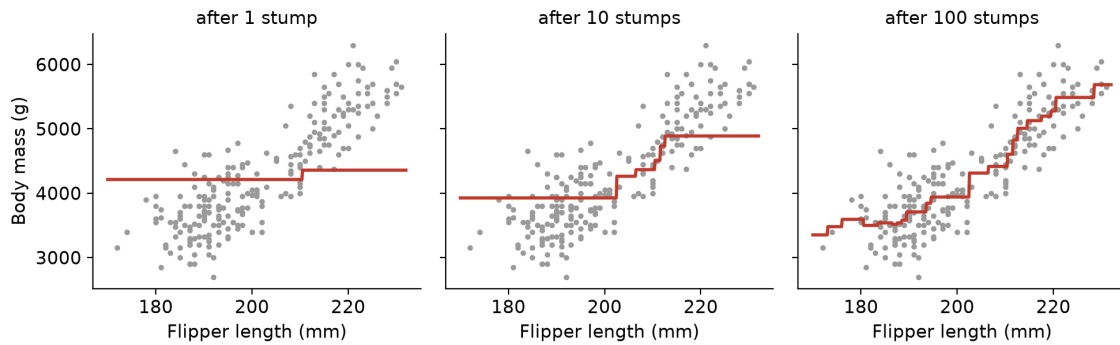


Figure 3.2: Gradient boosting with a learning rate of 0.1, predicting body mass from flipper length.

Here is `gradient_boost.py`, which predicts body mass from all the features we used for multiple regression:

```

1  import numpy as np
2  from penguins import load_penguins
3
4  data = load_penguins()
5  y = data["body_mass_g"]
6  data["male"] = (data["sex"] == "male").astype(float)
7  data["gentoo"] = (data["species"] == "Gentoo").astype(float)
8  data["chinstrap"] = (data["species"] == "Chinstrap").astype(float)
9  features = ["flipper_length_mm", "bill_length_mm", "bill_depth_mm",
10             "male", "gentoo", "chinstrap"]
11  X = np.column_stack([data[f] for f in features])
12  order = np.random.default_rng(1).permutation(len(y))
13  train, test = order[:233], order[233:]
14
15  def fit_stump(X, r):
16      """The one-question rule that best predicts the residuals r (least squares)."""
17      best = (np.inf, None, None, None, None)
18      for j in range(X.shape[1]):
19          values = np.unique(X[:, j])
20          for t in (values[:-1] + values[1:]) / 2:
21              left = X[:, j] <= t
22              mean_left, mean_right = r[left].mean(), r[~left].mean()
23              sse = np.sum((r[left] - mean_left)**2) + np.sum((r[~left] -
24                  ↪ mean_right)**2)
25              if sse < best[0]:
26                  best = (sse, j, t, mean_left, mean_right)
27      return best
28
29  def stump_predict(stump, X):
30      _, j, t, mean_left, mean_right = stump
31      return np.where(X[:, j] <= t, mean_left, mean_right)

```

```

32 def rmse(pred, rows):
33     return np.sqrt(np.mean((y[rows] - pred)**2))
34
35 rate = 0.1
36 pred_train = np.full(len(train), y[train].mean()) # start with the average
37 pred_test = np.full(len(test), y[train].mean())
38 for t in range(1, 501):
39     residuals = y[train] - pred_train
40     stump = fit_stump(X[train], residuals)
41     pred_train += rate * stump_predict(stump, X[train])
42     pred_test += rate * stump_predict(stump, X[test])
43     if t in (1, 10, 50, 100, 200, 500):
44         print(f"{t:3d} stumps: training RMSE {rmse(pred_train, train):.0f}, "
45               f"test RMSE {rmse(pred_test, test):.0f}")

```

```

1 stumps: training RMSE 762, test RMSE 727
10 stumps: training RMSE 494, test RMSE 482
50 stumps: training RMSE 294, test RMSE 294
100 stumps: training RMSE 278, test RMSE 289
200 stumps: training RMSE 265, test RMSE 290
500 stumps: training RMSE 249, test RMSE 303

```

A regression stump predicts the average residual on each side of its threshold, and it chooses the threshold that makes the squared error smallest, just as a classification stump minimizes impurity.

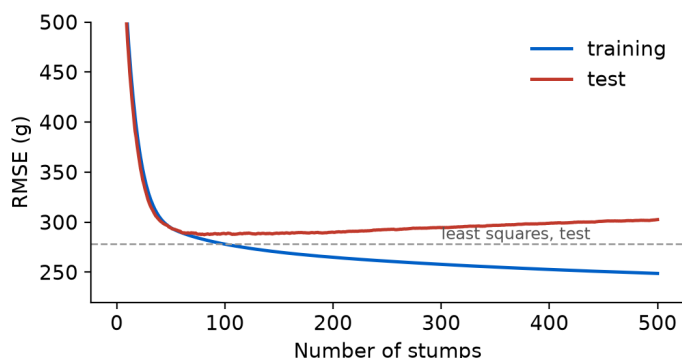


Figure 3.3: Gradient boosting RMSE as stumps are added. The dashed line is least squares on the same six features.

The pattern is the same as for AdaBoost: the training error keeps falling, while the test error bottoms out around 100 stumps and then creeps back up. At its best, gradient boosting gets a test RMSE of 289 g. That is honest but humbling: least squares on the same six features gets 278 g. Penguin mass is close to a linear function of these features, and a linear model is hard to beat at being linear. Boosting earns its reputation on problems with curved relationships and interactions between features, where a straight line cannot

follow the data. Boosted trees win a large share of machine-learning competitions on tables of data.

3.5 Cautions

- Boosting can overfit if the number of rounds is too large. Choose it with validation data.
- A small learning rate slows the fit and usually generalizes better, but needs more rounds.
- Because each model chases the hardest examples, mislabeled data (a male recorded as female) can pull boosting off course.

Exercise 13 Compute alpha

A weak learner has weighted error $\varepsilon = 0.1$. What is α ?

Working Space

Answer on Page 47

Exercise 14 Which points gain weight?

After a round of AdaBoost, which points have their weights increased?

Working Space

Answer on Page 47

Exercise 15 A coin-flip learner

A weak learner has weighted error $\varepsilon = 0.5$. What is α , and what does that

Working Space

mean?

Answer on Page 47

Exercise 16 **Half the weight**

Show that after any AdaBoost round, the points the learner got wrong carry exactly half of the total weight. (Hint: before normalizing, the wrong points have total weight ϵe^α and the right ones have $(1 - \epsilon)e^{-\alpha}$.)

Working Space

Answer on Page 47

Exercise 17 **A second round**

Continue the house example for one more round: the second tree learns the residuals -1.5 , -0.5 , and 2 exactly, and the learning rate is still 0.5 . What are

Working Space

the new predictions and residuals?

Answer on Page 47

Exercise 18 **Learning rate**

In `gradient_boost.py`, change the learning rate to 1.0 and then to 0.01, keeping 500 rounds. What happens to the training and test RMSE?

Working Space

Answer on Page 48

Generalized Linear Models

How many customers will walk into the smoothie shop next to school this afternoon? How many goals will your team score? Linear regression could make a prediction for each, but it might predict -0.3 customers on a cold day, and it assumes the scatter around the line is the same size everywhere, which is not how counts behave.

This chapter shows how to keep the simplicity of a linear model while respecting what the answer is allowed to be. The trick is a *link function*: a leash that keeps a linear predictor, which can wander anywhere from $-\infty$ to ∞ , inside the range the answer is allowed to take. You have already met one: logistic regression is a generalized linear model.

4.1 Beyond the Normal Distribution

Ordinary linear regression assumes the response is normal, with a mean that is a linear function of the predictors. Many responses are not normal: counts, yes-or-no outcomes, and positive amounts all break that assumption. A *generalized linear model* (GLM) keeps the linear predictor but allows other distributions for the response.

4.2 Three Components

A GLM has three parts.

1. **Random component**: the distribution of the response, such as normal, binomial, or Poisson.
2. **Systematic component**: the linear predictor $\eta = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p = X\beta$.
3. **Link function** g : a function that connects the mean of the response to the linear predictor,

$$g(E[Y]) = \eta$$

Model	Response	Distribution	Link
linear regression	any real number	normal	identity, μ
logistic regression	yes or no	binomial	logit, $\ln \frac{\mu}{1-\mu}$
Poisson regression	a count	Poisson	log, $\ln \mu$

4.3 Link Functions

Write $\mu = E[Y]$ for the mean of the response, so the link says $g(\mu) = \eta$. The link must map the allowed values of μ onto all real numbers, because η can be any real number. The inverse link $\mu = g^{-1}(\eta)$ turns a prediction on the linear scale back into the scale of the response.

- **Identity:** $g(\mu) = \mu$. The mean can be any real number. This is ordinary linear regression.
- **Log:** $g(\mu) = \ln \mu$. The mean must be positive, as for counts. The inverse is $\mu = e^\eta$.
- **Logit:** $g(\mu) = \ln \frac{\mu}{1-\mu}$. The mean is a probability between 0 and 1. The inverse is the sigmoid from the Logistic Regression chapter.
- **Probit:** $g(\mu) = \Phi^{-1}(\mu)$, where Φ is the standard normal cumulative distribution function. It is another link for probabilities.

For a mean of $\mu = 0.8$, the logit is $\ln \frac{0.8}{0.2} = \ln 4 \approx 1.386$, and the inverse logit of 1.386 returns $\frac{e^{1.386}}{1+e^{1.386}} = 0.8$. For a mean of $\mu = 10$, the log link is $\ln 10 \approx 2.303$, and the inverse gives $e^{2.303} = 10$.

Each distribution has a *canonical* link that is the natural default: identity for the normal, logit for the binomial, and log for the Poisson. The choice changes how coefficients are read. Under the identity link, a coefficient *adds* to the mean; under the log link, e^β *multiplies* the mean; under the logit link, e^β multiplies the odds.

4.4 Counts and the Poisson Distribution

The *Poisson distribution* describes counts of events that happen independently at some average rate: customers per hour, typos per page, goals per game. If the mean is μ ,

$$P(Y = k) = \frac{\mu^k e^{-\mu}}{k!}, \quad k = 0, 1, 2, \dots$$

Its variance equals its mean. So when the expected count is small, the counts are bunched near zero, and when it is large, they spread out:

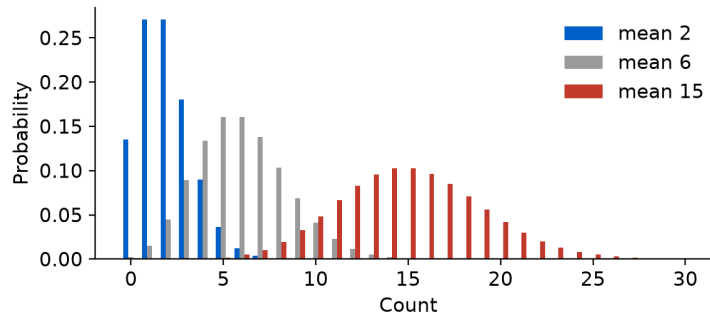


Figure 4.1: Poisson distributions with means 2, 6, and 15. The spread grows with the mean.

Poisson regression uses the log link: $\ln \mu = \beta_0 + \beta_1 x_1 + \dots$, so $\mu = e^{\beta_0 + \beta_1 x_1 + \dots}$, which is always positive.

Example: A Poisson model for the number of customers Y per hour at a smoothie shop has $\beta_0 = 0.5$ and $\beta_1 = 0.2$, where x is the number of social media posts the shop made that day. Find the expected number of customers when $x = 3$.

Solution: $\eta = 0.5 + 0.2(3) = 1.1$, so $E[Y] = e^{1.1} \approx 3.004$. Each extra post multiplies the expected number of customers by $e^{0.2} \approx 1.22$, a 22% increase.

4.5 Fitting a GLM

Like logistic regression, a GLM is fit by maximum likelihood. For Poisson regression, with $\mu_i = e^{\eta_i}$ and $\eta_i = \mathbf{x}_i^T \boldsymbol{\beta}$, the log of the likelihood is

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^n [y_i \eta_i - e^{\eta_i} - \ln(y_i!)]$$

Its gradient has a familiar shape:

$$\nabla \ell = \mathbf{X}^T (\mathbf{y} - \boldsymbol{\mu})$$

Predicted minus actual (or here, actual minus predicted), multiplied by $\mathbf{X}^T$, exactly as for linear and logistic regression. That is no coincidence: it is true for every GLM with its canonical link.

You could climb this hill with gradient ascent. But GLMs are usually fit with *Newton's method*, which also uses the second derivatives. The matrix of second derivatives (the *Hessian*) is $-\mathbf{X}^T \mathbf{W} \mathbf{X}$, where $\mathbf{W}$ is a diagonal matrix with the μ_i on its diagonal, and each Newton step is

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T (\mathbf{y} - \boldsymbol{\mu})$$

Each step solves a weighted least squares problem, with weights that change as μ changes, so the method is called *iteratively reweighted least squares* (IRLS). It usually converges in a handful of steps, far fewer than gradient descent.

4.6 Poisson Regression in Python

The smoothie shop has kept a tally of its customers between 3 and 4 p.m. for 120 days. (The data is simulated, from a Poisson model with known coefficients, so we can check our answer.) Does the crowd depend on the temperature and on whether it is a weekend? Here is `smoothies.py`:

```

1 import numpy as np
2
3 # Simulated data: customers at a smoothie shop between 3 and 4 p.m. on 120 days
4 rng = np.random.default_rng(7)
5 n = 120
6 temp = rng.uniform(0, 32, n)           # afternoon temperature, degrees
   ↪ C
7 weekend = (rng.random(n) < 2 / 7).astype(float)
8 true_beta = np.array([0.3, 0.07, 0.5]) # intercept, temperature,
   ↪ weekend
9 X = np.column_stack([np.ones(n), temp, weekend])
10 customers = rng.poisson(np.exp(X @ true_beta))
11
12 # Poisson regression by iteratively reweighted least squares (Newton's method)
13 beta = np.array([np.log(customers.mean()), 0, 0]) # start: every day average
14 for step in range(1, 7):
15     mu = np.exp(X @ beta)                # current predicted means
16     gradient = X.T @ (customers - mu)
17     hessian = X.T @ (mu[:, None] * X)    # X^T W X with W = diag(mu)
18     beta = beta + np.linalg.solve(hessian, gradient)
19     print(f"step {step}: beta = {np.round(beta, 4)}")
20
21 print("multipliers e^beta:", np.round(np.exp(beta), 3))
22
23 # Compare with an ordinary least squares line
24 b_ls, *_ = np.linalg.lstsq(X, customers)
25 cold_weekday = np.array([1, 0, 0])
26 hot_weekend = np.array([1, 32, 1])
27 for name, x in [("0 C weekday", cold_weekday), ("32 C weekend", hot_weekend)]:
28     print(f"{name}: Poisson {np.exp(x @ beta):5.1f}, least squares {x @
   ↪ b_ls:5.1f}")

```

```

step 1: beta = [0.7303 0.0582 0.4481]
step 2: beta = [0.5383 0.061 0.4238]
step 3: beta = [0.5179 0.0615 0.4197]
step 4: beta = [0.5177 0.0616 0.4196]
step 5: beta = [0.5177 0.0616 0.4196]

```

```

step 6: beta = [0.5177 0.0616 0.4196]
multipliers e^beta: [1.678 1.063 1.521]
0 C weekday: Poisson 1.7, least squares -0.3
32 C weekend: Poisson 18.3, least squares 13.4

```

`rng.poisson` draws random counts with the given means. `mu[:, None] * X` multiplies row `i` of `X` by μ_i , which is the same as `WX` without building the 120×120 diagonal matrix.

Newton's method settles in four steps. (Starting from all zeros, it overshoots wildly; starting from "every day is average," $\beta_0 = \ln \bar{y}$, is the standard trick.) The estimates, 0.52, 0.062, and 0.42, are close to the true 0.3, 0.07, and 0.5 but not equal, because 120 days of random counts carry only so much information. Reading the multipliers:

Term	e^β	Meaning
intercept	1.678	about 1.7 customers expected on a 0 °C weekday
temperature	1.063	each degree warmer: 6.3% more customers
weekend	1.521	weekends: 52% more customers, at any temperature

Over a 30-degree range, 6.3% per degree compounds: $1.063^{30} \approx 6.3$ times as many customers on a hot day. That is the curve in Figure 4.2.

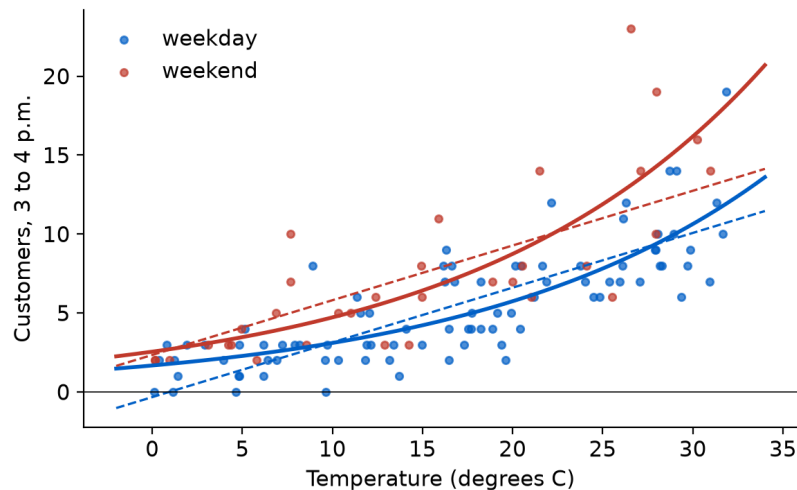


Figure 4.2: The simulated smoothie data. Solid curves: the Poisson regression. Dashed lines: least squares on the same features.

The least squares lines miss in both directions. They predict a negative number of customers on a freezing weekday, and they cannot bend upward, so they underestimate the hot weekends. They also treat a miss of 5 customers as equally surprising on a slow day and a busy one, when the Poisson distribution says busy days naturally vary more.

Exercise 19 Compute the mean

In the social-media example, what is the expected count when $x = 5$?

Working Space

Answer on Page 48

Exercise 20 Name the model

A survey records whether each person voted (yes or no). Which GLM is appropriate, and what is its link?

Working Space

Answer on Page 48

Exercise 21 Logit value

Find the logit of $\mu = 0.25$.

Working Space

Answer on Page 48

Exercise 22 Interpret the log link

Under the log link, $\ln \mu = \beta_0 + \beta_1 x$. If $\beta_1 = -0.1$, by what factor does the mean change when x increases by 1?

Working Space

Answer on Page 48

Exercise 23 Logit versus probit

For $\mu = 0.8$, the logit is 1.386. The probit is $\Phi^{-1}(0.8)$. Use a z-table (or `scipy.stats.norm.ppf`) to find it, and compare.

Working Space

Answer on Page 48

Exercise 24 Poisson probabilities

On a 20 °C weekday, the fitted model expects $e^{0.5177+0.0616(20)} \approx 5.8$ customers. Using the Poisson distribution, what is the probability of exactly 5 customers? Of none?

Working Space

Answer on Page 49

Exercise 25 Check with SciPy

Write a function that returns the negative Poisson log-likelihood $-\ell(\beta)$ for the smoothie data, minimize it with SciPy starting from zeros, and compare with the IRLS answer. (SciPy's `gamma` func-

Working Space

tion computes $\ln(y!)$ as $\ln \Gamma(y + 1)$.)

Answer on Page 49

Exercise 26 **How sure are we?**

For maximum likelihood estimates, the standard errors are the square roots of the diagonal of the inverse of $X^T W X$ at the solution. Compute them for the smoothie model. Is the true weekend coefficient, 0.5, within two standard errors of the estimate?

Working Space

Answer on Page 49

Clustering with k-Means

Imagine that a field team measured 333 penguins but forgot to write down the species. All they have are bill lengths and bill depths. Could a computer discover, on its own, that there are three kinds of penguin, and sort them? Streaming services face the same problem with listeners: nobody labels you “indie-folk fan,” but your listening history puts you near other people with similar taste.

5.1 Unsupervised Learning

Every method so far in this unit was *supervised*: we had labels and learned to predict them. In *unsupervised learning*, there are no labels. We look for structure in the features alone. *Clustering* groups the points so that points in the same group are close to each other and far from the other groups. The *k-means* algorithm does this for a chosen number of groups k .

5.2 The Algorithm

Given points $x_1, \dots, x_N$ and a number k , k -means looks for *centroids* $c_1, \dots, c_k$ that make the total squared distance from each point to its nearest centroid as small as possible. It alternates two steps:

1. **Assign:** each point joins the cluster of its nearest centroid.
2. **Update:** each centroid moves to the mean of the points assigned to it.

Repeat until the assignments stop changing. Each step can only lower the total squared distance (the assign step picks the closest centroid for each point, and the mean is the point that minimizes the squared distance to a group), so the algorithm always stops.

Example: Four points on a line are 1, 2, 6, 8, and $k = 2$. Start with centroids $c_1 = 1$ and $c_2 = 6$.

Solution:

- Assign: the points 1 and 2 are nearer c_1 , and the points 6 and 8 are nearer c_2 .
- Update: $c_1 = \frac{1+2}{2} = 1.5$ and $c_2 = \frac{6+8}{2} = 7$.

The next assignment is the same, so the algorithm stops with clusters $\{1, 2\}$ and $\{6, 8\}$. The total squared distance is $0.5^2 + 0.5^2 + 1^2 + 1^2 = 2.5$.

Figure 5.1 shows the same two steps on the penguins, starting from a deliberately bad choice: three Adelies, all in the same corner. Within a few steps, the centroids spread out to the three groups.

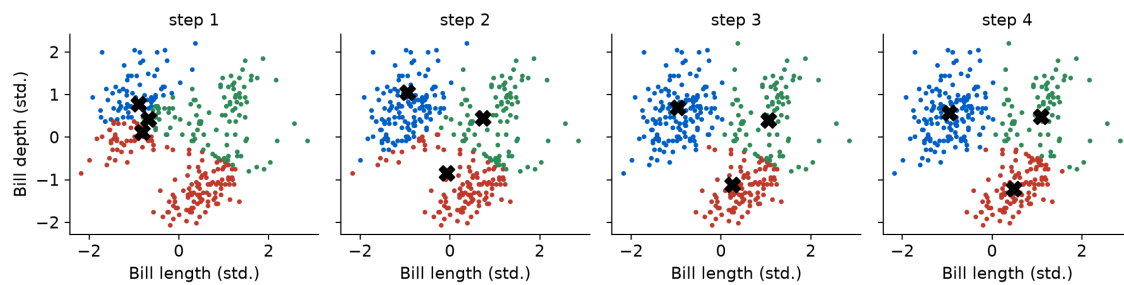


Figure 5.1: k-means on standardized bill measurements, $k = 3$. Each panel shows the centroids (crosses) and the assignment they produce; the next panel's centroids are the means of those groups.

5.3 Starting Well

The result depends on the starting centroids. A bad start can leave the algorithm stuck in a poor local minimum, for example with two centroids sharing one real group while a third covers two. Two standard defenses:

- **Run it several times** from different random starts and keep the run with the smallest total squared distance.
- **k-means++**: choose the first centroid at random, then choose each next one at random with probability proportional to its squared distance from the nearest centroid already chosen. Far-away points are likely picks, so the starting centroids spread out.

5.4 Clustering Penguins

Here is `kmeans.py`. It standardizes the two bill measurements (distances again, so units matter), then runs k-means++ ten times for each k and keeps the best run:

```
1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 X = np.column_stack([data["bill_length_mm"], data["bill_depth_mm"]])
```

```

6 Z = (X - X.mean(axis=0)) / X.std(axis=0)           # standardize first
7
8 def kmeans_plus_plus(Z, k, rng):
9     """Spread out the starting centroids: each new one is chosen with
10    probability proportional to its squared distance from the nearest so far."""
11    centers = [Z[rng.integers(len(Z))]]
12    for _ in range(k - 1):
13        d2 = np.min([np.sum((Z - c)**2, axis=1) for c in centers], axis=0)
14        centers.append(Z[rng.choice(len(Z), p=d2 / d2.sum())])
15    return np.array(centers)
16
17 def kmeans(Z, k, seed=0, max_steps=100):
18     rng = np.random.default_rng(seed)
19     centers = kmeans_plus_plus(Z, k, rng)
20     for step in range(max_steps):
21         # Assign: squared distance from every point to every center
22         d2 = np.sum((Z[:, None, :] - centers[None, :, :])**2, axis=2)
23         labels = d2.argmin(axis=1)
24         # Update: move each center to the mean of its points
25         new = np.array([Z[labels == j].mean(axis=0) for j in range(k)])
26         if np.allclose(new, centers):
27             break
28         centers = new
29         inertia = np.sum((Z - centers[labels])**2)           # total squared distance
30     return labels, centers, inertia, step + 1
31
32 for k in range(1, 7):
33     best = min((kmeans(Z, k, seed) for seed in range(10)), key=lambda r: r[2])
34     print(f"k = {k}: total squared distance {best[2]:6.1f}")
35
36 labels, centers, inertia, steps = min((kmeans(Z, 3, seed) for seed in range(10)),
37                                     key=lambda r: r[2])
38 print(f"k = 3 converged in {steps} steps")
39 print("Centers in mm:")
40 print(np.round(centers * X.std(axis=0) + X.mean(axis=0), 1))
41
42 species = data["species"]
43 print("cluster Adelie Chinstrap Gentoo")
44 for j in range(3):
45     counts = [np.sum((labels == j) & (species == s)) for s in ("Adelie",
46     ↪ "Chinstrap", "Gentoo")]
47     print(f"{j:7d} {counts[0]:7d} {counts[1]:10d} {counts[2]:7d}")

```

```

k = 1: total squared distance 666.0
k = 2: total squared distance 334.0
k = 3: total squared distance 183.1
k = 4: total squared distance 145.4
k = 5: total squared distance 112.1
k = 6: total squared distance 92.7
k = 3 converged in 5 steps
Centers in mm:
[[47.2 15. ]

```

```
[49.9 18.7]
[38.8 18.3]]
cluster Adelie Chinstrap Gentoo
0      0      9     112
1      4     54      6
2     142      5      1
```

The assign step uses the same broadcasting trick as k-nearest neighbors. The total squared distance is often called the *inertia*. (With $k = 1$, it is $666 = 2 \times 333$, because each standardized feature has variance 1.) The `key=lambda r: r[2]` tells `min` to compare runs by their inertia.

5.5 Did It Find the Species?

We hid the species from the algorithm, but we still have them, so we can check. The table of counts at the end of the output is called a *contingency table*:

	Adelie	Chinstrap	Gentoo
cluster 0	0	9	112
cluster 1	4	54	6
cluster 2	142	5	1

Each cluster is mostly one species, and $112 + 54 + 142 = 308$ of the 333 penguins, 92.5%, are in the cluster that matches their species. The algorithm was never told there were species at all. Notice that the cluster *numbers* mean nothing: cluster 2 is “the Adelie cluster” only because we looked. In a real unsupervised problem, deciding what each cluster means is the human’s job.

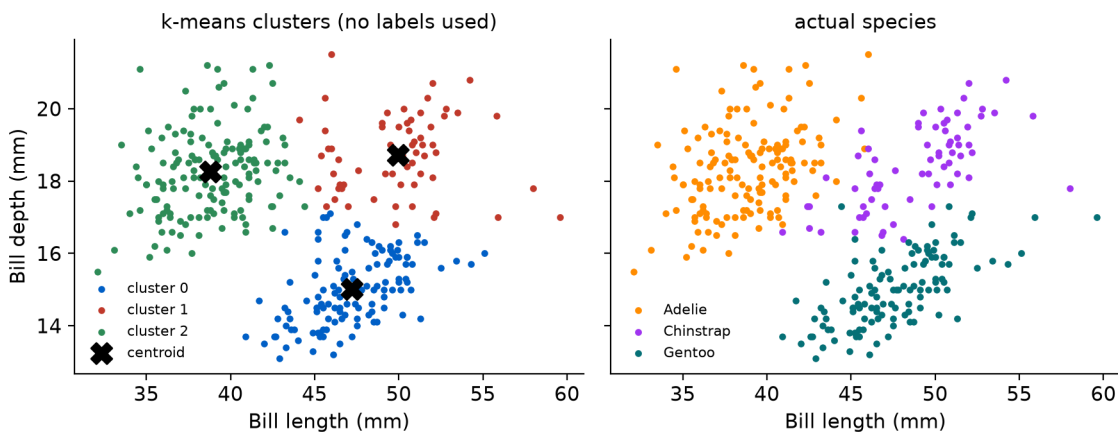


Figure 5.2: Left: the three clusters and their centroids. Right: the true species, for comparison.

The mistakes are where the species overlap, mainly Chinstraps with short or shallow bills. k-means also assumes clusters are round blobs of similar size, since it uses plain distance to the nearest centroid. The Gentoo and Chinstrap clouds are tilted ovals, so some of their points end up closer to the wrong centroid.

5.6 Choosing k

The number k is not learned by the algorithm. Inertia always falls as k grows (with $k = N$, every point is its own cluster and the inertia is 0), so you cannot just pick the k with the smallest inertia. The *elbow method* plots inertia against k and looks for the point where the decrease levels off.

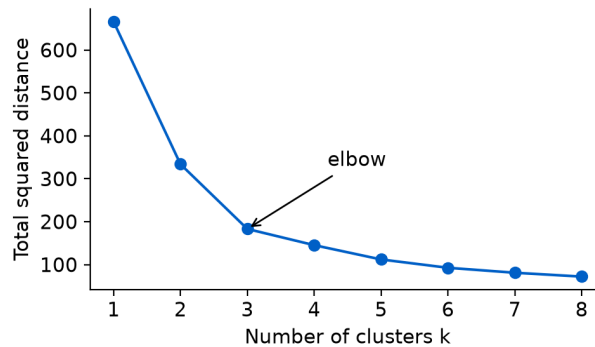


Figure 5.3: Inertia against k for the penguins.

Going from 2 to 3 clusters removes 151 units of inertia; going from 3 to 4 removes only 38. The elbow is at $k = 3$, matching the three species. Real data is rarely this cooperative. Other tools, such as *silhouette scores*, measure how much closer each point is to its own cluster than to the next nearest one. Often the best guide is whether the clusters are useful for whatever you plan to do with them.

Exercise 27 One iteration

Using the four points 1, 2, 6, 8, start with the poorly placed centroids $c_1 = 1$ and

Working Space

$c_2 = 2$. Run k-means until it stops.

Answer on Page 49

Exercise 28 Why run it again?

Why is k-means usually run several times with different starting centroids?

Working Space

Answer on Page 50

Exercise 29 Two-dimensional step

Points $(0, 0)$, $(0, 2)$, $(4, 0)$, and $(4, 2)$ are clustered with $k = 2$, starting from centroids $(0, 1)$ and $(1, 1)$. Do one assign step and one update step. Where do the centroids end up?

Working Space

Answer on Page 50

Exercise 30 Does scaling matter?

Run `kmeans.py` with $k = 3$ on all four measurements (bill length, bill depth, flipper length, and body mass), first standardized and then in raw units. Make the contingency table for each. What happens without standardizing, and why?

*Working Space**Answer on Page 50*

Answers to Exercises

Answer to Exercise 1 (on page 8)

$$G = 1 - (0.75^2 + 0.25^2) = 1 - 0.625 = 0.375.$$

Answer to Exercise 2 (on page 9)

The weighted impurity is $0.5(0.2) + 0.5(0.3) = 0.25$, which is lower than 0.48. The split is an improvement.

Answer to Exercise 3 (on page 9)

The left child is pure, so its impurity is 0. The right child has $1 - (0.2^2 + 0.8^2) = 0.32$. The weighted impurity is $\frac{5}{10}(0) + \frac{5}{10}(0.32) = 0.16$, much lower than the 0.40 from the worked example. The tree chooses this new split.

Answer to Exercise 4 (on page 9)

$H = -(0.6 \log_2 0.6 + 0.4 \log_2 0.4) \approx 0.971$ bits. The left child has $H = -(0.8 \log_2 0.8 + 0.2 \log_2 0.2) \approx 0.722$, and the right child has $H \approx 0.971$ (it is the same mix, reversed). The weighted entropy is $0.5(0.722) + 0.5(0.971) \approx 0.846$, so the information gain is $0.971 - 0.846 \approx 0.125$ bits.

Answer to Exercise 5 (on page 10)

(a) $190 \leq 206.5$, then $48 > 43.35$: Chinstrap. (b) $220 > 206.5$, then $14.5 \leq 17.65$: Gentoo. (c) $210 > 206.5$, then $19 > 17.65$: Chinstrap. That last leaf is small (2 Adelies and 4 Chinstraps), so the tree is not very sure about it.

Answer to Exercise 6 (on page 10)

The first question becomes “bill depth ≤ 16.45 mm?”, which separates the shallow-billed Gentoos, and the second level splits on bill length. The depth-2 tree scores 92.5% on the test penguins, and depths 4 and above score 94.0%, a little below the 95.5% with flipper length. The tree finds another route to nearly the same answer.

Answer to Exercise 7 (on page 15)

$$(1 - \frac{1}{5})^5 = (0.8)^5 \approx 0.328.$$

Answer to Exercise 8 (on page 16)

$$m = \sqrt{16} = 4.$$

Answer to Exercise 9 (on page 16)

The majority is right if all three are right or exactly two are: $0.8^3 + 3(0.8^2)(0.2) = 0.512 + 0.384 = 0.896$.

Answer to Exercise 10 (on page 16)

Our tree-growing algorithm has no randomness, so every tree would be identical. All 100 would make exactly the same predictions, the vote would always be unanimous, and the forest would be right exactly when the single tree is.

Answer to Exercise 11 (on page 17)

```
1 rng = np.random.default_rng(0)
2 n = 266
3 fractions = [len(np.unique(rng.integers(0, n, size=n))) / n for _ in range(1000)]
4 print(np.mean(fractions))
```

The average is about 0.635, matching $1 - 0.367 = 0.633$ up to random variation.

Answer to Exercise 12 (on page 17)

The single trees get worse on average (89.3%, ranging from 73% to 100%), because many splits are forced to use a weak feature. The forest still gets 97.0% of the test penguins right, with an out-of-bag accuracy of 97.4%, because the votes of many varied trees cancel out much of that weakness. There is a trade-off: smaller m makes the trees more independent but individually less accurate.

Answer to Exercise 13 (on page 25)

$$\alpha = \frac{1}{2} \ln \frac{0.9}{0.1} = \frac{1}{2} \ln 9 \approx 1.099.$$

Answer to Exercise 14 (on page 25)

The points that h_t misclassified. For those points, $y_i h_t(x_i) = -1$, so $e^{-\alpha_t y_i h_t(x_i)} = e^{\alpha_t} > 1$.

Answer to Exercise 15 (on page 25)

$\alpha = \frac{1}{2} \ln \frac{0.5}{0.5} = \frac{1}{2} \ln 1 = 0$. A learner that does no better than flipping a coin gets no vote in the final classifier.

Answer to Exercise 16 (on page 26)

With $\alpha = \frac{1}{2} \ln \frac{1-\varepsilon}{\varepsilon}$, we have $e^\alpha = \sqrt{\frac{1-\varepsilon}{\varepsilon}}$. So the wrong points have $\varepsilon e^\alpha = \sqrt{\varepsilon(1-\varepsilon)}$, and the right points have $(1-\varepsilon)e^{-\alpha} = \sqrt{\varepsilon(1-\varepsilon)}$, the same amount. After normalizing, each group has weight $\frac{1}{2}$. That is why the worked example ended with the missed point at exactly 0.5: the last learner now looks no better than a coin flip on the new weights, so the next learner must find something new.

Answer to Exercise 17 (on page 26)

The predictions become $4.5 - 0.75 = 3.75$, $5.5 - 0.25 = 5.25$, and $8 + 1 = 9$. The residuals are -0.75 , -0.25 , and 1 , half as big again. Each round halves the error, so the predictions approach the true prices.

Answer to Exercise 18 (on page 27)

With a rate of 1.0, the training RMSE drops very fast (to 180 g after 500 stumps), but the test RMSE is best, about 298 g, around 20 stumps, and then climbs to 355 g as the model overfits. With 0.01, progress is slow: after 500 stumps, both RMSEs are still falling (296 g training, 295 g test), and the model has not finished learning. A smaller rate needs more rounds, but it gives a gentler curve that is easier to stop at the right place.

Answer to Exercise 19 (on page 34)

$\eta = 0.5 + 0.2(5) = 1.5$, so $E[Y] = e^{1.5} \approx 4.48$.

Answer to Exercise 20 (on page 34)

Logistic regression: a binomial response with the logit link.

Answer to Exercise 21 (on page 34)

$\ln \frac{0.25}{0.75} = \ln \frac{1}{3} \approx -1.099$.

Answer to Exercise 22 (on page 34)

The mean is multiplied by $e^{-0.1} \approx 0.905$, a decrease of about 9.5%. A negative coefficient under the log link shrinks the mean, but never makes it negative.

Answer to Exercise 23 (on page 35)

$\Phi^{-1}(0.8) \approx 0.842$. The probit value is smaller because the normal curve has thinner tails than the logistic curve. The two links usually give similar fitted probabilities, but their coefficients are on different scales (logit coefficients are roughly 1.6 to 1.8 times probit coefficients).

Answer to Exercise 24 (on page 35)

$P(Y = 5) = \frac{5.8^5 e^{-5.8}}{5!} \approx \frac{6563.6 \times 0.003028}{120} \approx 0.166$, and $P(Y = 0) = e^{-5.8} \approx 0.0030$. An empty hour on a warm afternoon would be very unusual.

Answer to Exercise 25 (on page 35)

```

1 from scipy.optimize import minimize
2 from scipy.special import gammaln
3
4 def neg_log_like(b):
5     eta = X @ b
6     return -np.sum(customers * eta - np.exp(eta) - gammaln(customers + 1))
7
8 result = minimize(neg_log_like, np.zeros(3))
9 print(np.round(result.x, 4))

```

It returns about (0.5177, 0.0616, 0.4196), the same as IRLS. The general-purpose minimizer needs more steps (10 here, against 4), because it has to estimate the derivatives that Newton's method gets from formulas.

Answer to Exercise 26 (on page 36)

```

1 mu = np.exp(X @ beta)
2 cov = np.linalg.inv(X.T @ (mu[:, None] * X))
3 print(np.round(np.sqrt(np.diag(cov)), 4))

```

The standard errors are about 0.105 for the intercept, 0.0044 for temperature, and 0.079 for weekend. The weekend estimate, 0.42, is about 1.0 standard error from 0.5, so yes. The intercept and temperature estimates are each about two standard errors from their true values (0.3 and 0.07): a reminder that estimates from random data wander.

Answer to Exercise 27 (on page 41)

Iteration 1: point 1 joins c_1 , and points 2, 6, and 8 join c_2 . The centroids become $c_1 = 1$ and $c_2 = \frac{2+6+8}{3} \approx 5.33$. Iteration 2: point 2 is now nearer c_1 (distance 1 versus 3.33), so the clusters are $\{1, 2\}$ and $\{6, 8\}$, and the centroids become 1.5 and 7. Iteration 3 changes nothing, so the algorithm stops. Even from a bad start, it found the natural clusters here,

though that does not always happen.

Answer to Exercise 28 (on page 42)

The algorithm converges to a local minimum of the total squared distance, and different starting centroids can lead to different final clusters. Keeping the run with the smallest total squared distance protects against an unlucky start.

Answer to Exercise 29 (on page 42)

$(0,0)$ and $(0,2)$ are at distance 1 from $(0,1)$ and $\sqrt{2}$ from $(1,1)$, so they join the first centroid. $(4,0)$ and $(4,2)$ are closer to $(1,1)$. The update moves the centroids to $(0,1)$ and $(4,1)$, and the next assignment does not change, so the algorithm stops.

Answer to Exercise 30 (on page 43)

Standardized, the clusters are $(0,0,119)$, $(124,5,0)$, and $(22,63,0)$ (Adelie, Chinstrap, Gentoos): the Gentoos are perfectly separated, and the main confusion is between Adelies and Chinstraps. In raw units, the clusters are $(0,0,70)$, $(108,52,1)$, and $(38,16,48)$, much worse. Body mass, in thousands of grams, swamps the other features, so the algorithm effectively clusters by weight alone, and weight does not separate the species well.



INDEX

- k-means, 37
- AdaBoost, 19
- Bagging, 12
- bagging, 12
- Boosting, 19
- boosting, 19
- bootstrap sample, 12
- canonical, 30
- centroids, 37
- Clustering, 37
- clustering, 37
- contingency table, 40
- decision stump, 19
- decision tree, 3
 - splitting, 4
- elbow method, 41
- entropy, 4
- feature importance, 15
- generalized linear model, 29
- Gini impurity, 3
- gradient boosted trees, 22
- Gradient boosting, 22
- greedy, 5
- Hessian, 31
- inertia, 40
- information gain, 4
- internal nodes, 3
- iteratively reweighted least squares, 32
- k-means clustering, 37
- k-means++, 38
- learning rate, 22
- leaves, 3
- link function, 30
- maximum likelihood, 31
- Newton's method, 31
- out-of-bag, 13
- out-of-bag error, 13
- Poisson distribution, 30
- Poisson regression, 31
- prune, 8
- random forest, 13
- recursion, 5
- unsupervised learning, 37
- weak learner, 19
- weak learners, 19