



CONTENTS

1	Logistic Regression	3
1.1	Why Not a Line?	3
1.2	The Model	4
1.3	Interpreting the Coefficients	4
1.4	Several Features	5
1.5	Fitting: Maximum Likelihood and Log Loss	5
1.6	Penguins in NumPy	6
1.7	Making a Decision	8
2	Evaluating Classification Systems	13
2.1	The Confusion Matrix	13
2.2	Performance Metrics	13
2.3	A Worked Example	13
2.4	Choosing a Metric	14
2.5	Grading a Chinstrap Detector	14
2.6	Beyond a Single Threshold	15
2.6.1	The Precision–Recall Tradeoff	16
2.6.2	The ROC Curve and AUC	16
2.6.3	Curves for the Chinstrap Detector	17
2.7	Which Measure to Use?	18
3	The k-Nearest Neighbor Classifier	23
3.1	The Algorithm	23
3.2	A Worked Example	23

3.3	Penguins, Neighbor by Neighbor	24
3.4	Choosing k	26
3.5	Strengths and Weaknesses	27
4	Stemming and Lemmatization	31
4.1	From Text to Words	31
4.2	Stemming	32
4.3	Lemmatization	32
4.4	A Stemmer in Python	32
4.5	Bag of Words	34
4.6	Where These Ideas Are Used	35
5	Bayesian Classifiers	39
5.1	Bayes' Rule for Classification	39
5.2	The Naive Assumption	40
5.3	Estimating the Word Probabilities	40
5.4	A Worked Example	40
5.5	Working in Logarithms	41
5.6	A Spam Filter in Python	42
5.7	Numbers Instead of Words	43
5.8	When Features Move Together	44
6	Bayesian Networks	51
6.1	Variables and Dependencies	51
6.2	A Small Network	51
6.3	Inference	51
6.3.1	Explaining Away, with Numbers	52
6.4	Is There School Today?	52
6.5	Inference in NumPy	54
6.6	Learning a Network	56
A	Answers to Exercises	59
	Index	67

Logistic Regression

Will it rain tomorrow? Will this email get opened? Will you get into the college you want? The answer is yes or no, but the honest prediction is a probability: “70% chance of rain.” *Logistic regression* turns numbers like humidity, hours of study, or GPA into exactly that kind of probability.

Our running example is a real problem for field biologists. Male and female Adelie penguins look alike, and checking a penguin’s sex directly takes a blood test. Can you predict it from a scale and a ruler?

1.1 Why Not a Line?

Code the label as a number: $y = 1$ for male and $y = 0$ for female. The goal is to model $p = P(y = 1)$, the probability that a penguin is male, from features like body mass.

You could fit a least squares line to the 0s and 1s, but look at Figure 1.1. The line predicts a probability of about -0.2 for the lightest Adelie and 1.4 for the heaviest. Probabilities must stay between 0 and 1. What we want is the S-shaped curve: flat near 0 for light penguins, flat near 1 for heavy ones, and rising in between, where the sexes overlap.

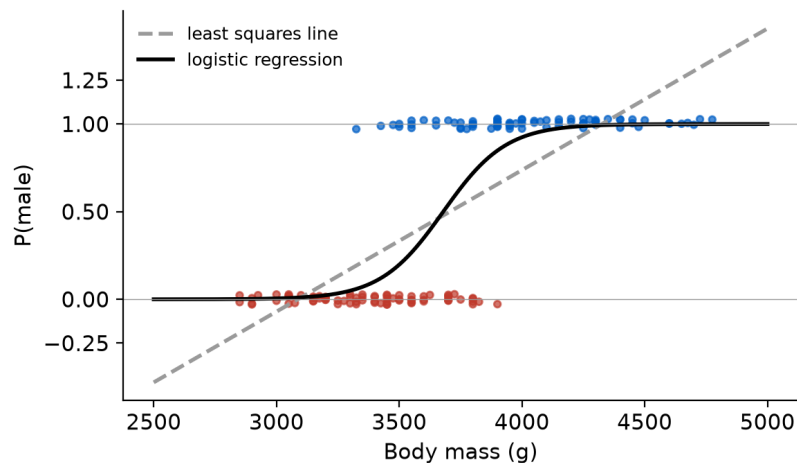


Figure 1.1: Sex against body mass for 146 Adelie penguins (males at 1, females at 0, shaken slightly so you can see them). The least squares line escapes the range from 0 to 1; the logistic curve does not.

1.2 The Model

The trick is to fit a line to something that *can* take any value: the log of the odds. The *odds* of an event are $\frac{p}{1-p}$; a probability of 0.75 is odds of 3 ("3 to 1"). Odds run from 0 to ∞ , and their logarithm, the *log odds* or *logit*, runs over all real numbers. So the model is

$$\ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x$$

Solving for p gives

$$p = \sigma(\beta_0 + \beta_1 x), \quad \text{where} \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

The function σ is the *sigmoid*, or logistic function. It squeezes any number into the range from 0 to 1, with $\sigma(0) = 0.5$.

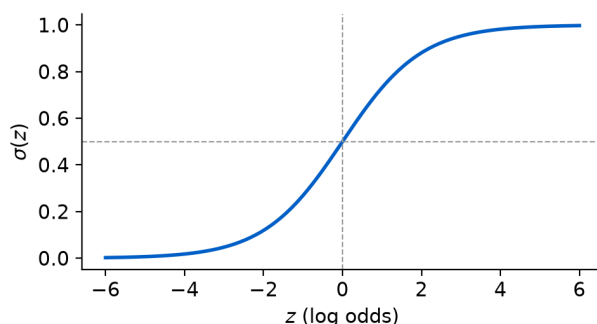


Figure 1.2: The sigmoid $\sigma(z) = 1/(1 + e^{-z})$.

Probability p	Odds $p/(1-p)$	Log odds
0.10	0.111	-2.20
0.25	0.333	-1.10
0.50	1	0
0.75	3	1.10
0.90	9	2.20

1.3 Interpreting the Coefficients

Take the model $\beta_0 = -4$ and $\beta_1 = 1$. Then $p(x) = \sigma(-4 + x)$. At $x = 3$, $p = \sigma(-1) \approx 0.269$. At $x = 4$, $p = \sigma(0) = 0.5$. At $x = 5$, $p = \sigma(1) \approx 0.731$.

Each one-unit increase in x adds β_1 to the log odds, which *multiplies* the odds by e^{β_1} . Here $e^1 \approx 2.718$: the odds at $x = 3$ are $e^{-1} \approx 0.368$, at $x = 4$ they are 1, and at $x = 5$ they are $e \approx 2.718$. The number e^{β_1} is called the *odds ratio*.

The divide-by-4 rule. Odds ratios are hard to feel. A quick estimate: the sigmoid is steepest at its midpoint, where its slope is $\frac{1}{4}$. So near $p = 0.5$, a one-unit increase in x changes the probability by about $\frac{\beta_1}{4}$. With $\beta_1 = 1$, that is about 25 percentage points per unit, and indeed p went from 0.5 to 0.731. The rule is only good near the middle of the curve.

1.4 Several Features

With features $x_1, \dots, x_p$, the log odds are a linear combination of all of them:

$$\ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p, \quad p = \sigma(\beta_0 + \beta_1 x_1 + \dots + \beta_p x_p)$$

In matrix form, with a design matrix X and parameters θ as in the last chapter, the predicted probabilities are $\mathbf{p} = \sigma(X\theta)$, applying σ to each entry. Each coefficient describes one feature with the others held constant.

Example: A student's chance of passing an exam is modeled with hours studied x_1 and whether they completed the prior course x_2 (1 if yes, 0 if no), with $\beta_0 = -3$, $\beta_1 = 0.5$, and $\beta_2 = 1$. Find the probability of passing for a student who studies 4 hours and completed the prior course, and for one who studies 2 hours and did not.

Solution: For the first student, the log odds are $-3 + 0.5(4) + 1 = 0$, so $P(\text{pass}) = \sigma(0) = 0.5$. For the second, the log odds are $-3 + 0.5(2) + 0 = -2$, so $P(\text{pass}) = \frac{1}{1+e^2} \approx 0.119$. Each extra hour multiplies the odds of passing by $e^{0.5} \approx 1.65$.

1.5 Fitting: Maximum Likelihood and Log Loss

Linear regression chose its parameters by least squares. For probabilities, there is a better measure of error. If the model says a penguin is male with probability p , then:

- if the penguin is male, the model should be penalized for a *small* p ;
- if it is female, the model should be penalized for a *large* p .

The *log loss* (also called *cross-entropy*) does exactly that:

$$\text{loss} = \begin{cases} -\ln p & \text{if } y = 1 \\ -\ln(1-p) & \text{if } y = 0 \end{cases} \quad \text{or, in one line,} \quad -[y \ln p + (1-y) \ln(1-p)]$$

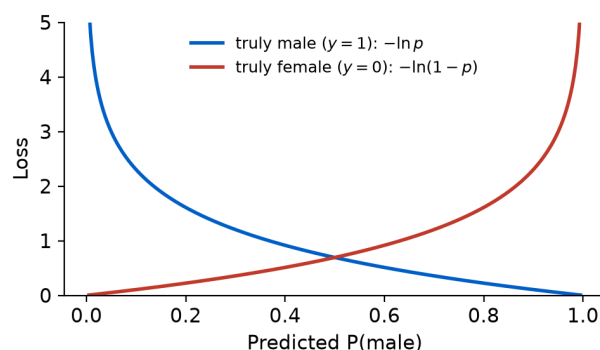


Figure 1.3: Log loss for a single penguin. Being confidently wrong costs a lot.

Example: A model gives four penguins these probabilities of being male. Find the average log loss.

Penguin	True sex	y	Predicted p	Loss
A	male	1	0.9	$-\ln 0.9 = 0.105$
B	male	1	0.6	$-\ln 0.6 = 0.511$
C	female	0	0.3	$-\ln 0.7 = 0.357$
D	female	0	0.8	$-\ln 0.2 = 1.609$

Solution: The average is $(0.105 + 0.511 + 0.357 + 1.609)/4 \approx 0.646$. Penguin D, a female the model was 80% sure was male, contributes more than the other three together.

Minimizing the average log loss is the same as choosing the parameters that make the observed labels most probable, which statisticians call *maximum likelihood*. Unlike least squares, there is no formula for the answer. But the loss is a smooth bowl, so gradient descent finds the bottom, and its gradient is remarkably simple:

$$\nabla J(\theta) = \frac{1}{n} X^T (\mathbf{p} - \mathbf{y})$$

Compare it with linear regression's $\frac{2}{n} X^T (X\theta - \mathbf{y})$: in both, you multiply X^T by the vector of errors, predicted minus actual.

1.6 Penguins in NumPy

Here is `logistic.py`. It predicts the sex of Adelie penguins from body mass and bill depth, training on 110 penguins and testing on 36:

```

1 import numpy as np
2 from scipy.optimize import minimize

```

```

3 from penguins import load_penguins
4
5 data = load_penguins()
6 adelic = data["species"] == "Adelie"
7 y = (data["sex"][adelic] == "male").astype(float) # 1 = male, 0 = female
8 features = np.column_stack([data["body_mass_g"][adelic],
9                             data["bill_depth_mm"][adelic]])
10 n = len(y)
11
12 order = np.random.default_rng(1).permutation(n)
13 train, test = order[:110], order[110:]
14
15 # Standardize with the training penguins' means and standard deviations
16 mean, sd = features[train].mean(axis=0), features[train].std(axis=0)
17 X = np.column_stack([np.ones(n), (features - mean) / sd])
18
19 def sigmoid(z):
20     return 1 / (1 + np.exp(-z))
21
22 def log_loss(theta, rows):
23     p = sigmoid(X[rows] @ theta)
24     return -np.mean(y[rows] * np.log(p) + (1 - y[rows]) * np.log(1 - p))
25
26 theta = np.zeros(3)
27 alpha = 0.5
28 for step in range(1, 1001):
29     p = sigmoid(X[train] @ theta)
30     gradient = X[train].T @ (p - y[train]) / len(train)
31     theta = theta - alpha * gradient
32     if step in (1, 10, 100, 1000):
33         print(f"step {step:4d}: loss {log_loss(theta, train):.4f}")
34
35 print("theta:", np.round(theta, 2))
36 for name, rows in [("train", train), ("test", test)]:
37     predicted = sigmoid(X[rows] @ theta) >= 0.5
38     print(f"{name} accuracy: {np.mean(predicted == y[rows]):.1%}")
39
40 # A new Adelie: 3600 g, bill depth 18.0 mm
41 x_new = np.concatenate([[1], (np.array([3600, 18.0]) - mean) / sd])
42 print(f"P(male) = {sigmoid(x_new @ theta):.2f}")
43
44 # Check against SciPy's general-purpose minimizer
45 result = minimize(log_loss, np.zeros(3), args=(train,))
46 print("SciPy theta:", np.round(result.x, 2))

```

```

step    1: loss 0.5855
step   10: loss 0.3401
step  100: loss 0.2443
step 1000: loss 0.2379
theta: [0.54 3.55 1.36]
train accuracy: 88.2%
test accuracy: 88.9%

```

```
P(male) = 0.32
SciPy theta: [0.54 3.55 1.36]
```

The model sexes 32 of the 36 test penguins correctly. The baseline, always guessing the more common sex, would get about half right. SciPy's `minimize` reaches the same θ , which is a good check that gradient descent found the bottom.

Reading the coefficients: the features are standardized, so 3.55 is the change in log odds for a penguin one standard deviation (474 g) heavier, with the same bill depth. Per 100 g, that is $3.55 \times 100/474 \approx 0.75$, an odds ratio of $e^{0.75} \approx 2.1$. Near a 50–50 penguin, the divide-by-4 rule says 100 g adds about $0.75/4 \approx 19$ percentage points to the chance of being male. Mass matters more than bill depth here, but both help.

1.7 Making a Decision

To turn a probability into a prediction, choose a *threshold*, usually 0.5: predict male when $p \geq 0.5$. Since $p = 0.5$ exactly when the log odds are 0, the boundary between the two predictions is where $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, a straight line. That is why logistic regression is called a *linear classifier*.

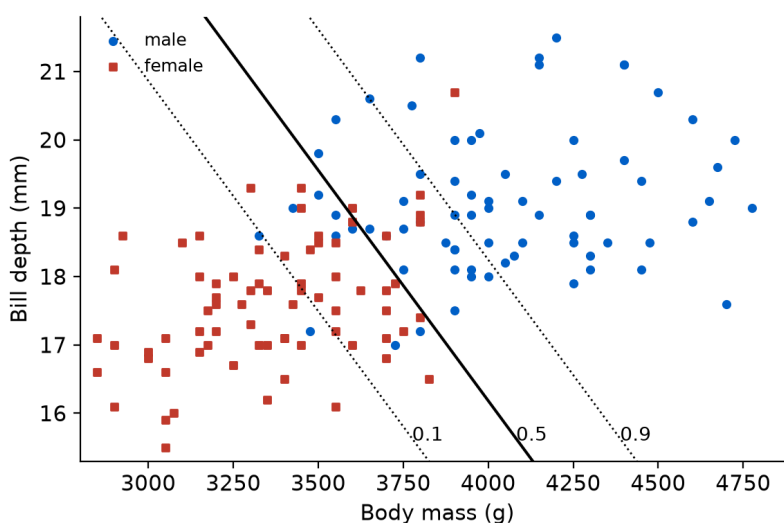


Figure 1.4: Lines of equal probability for a model fit to all 146 Adelies. The solid line is $P(\text{male}) = 0.5$; the dotted lines are 0.1 and 0.9.

The threshold is a choice, and changing it changes which mistakes the model makes. A spam filter might demand $p \geq 0.95$ before hiding a message, because hiding a real email is worse than letting one piece of spam through. The next chapter is all about measuring those trade-offs.

Exercise 1 **Compute a probability**

Using $\beta_0 = -4$ and $\beta_1 = 1$, find $p(2)$.

Working Space

Answer on Page 59

Exercise 2 **Find the threshold value**

With $\beta_0 = -4$ and $\beta_1 = 1$, for which x is $p(x) = 0.5$?

Working Space

Answer on Page 59

Exercise 3 **Interpret the odds ratio**

If $\beta_1 = 0.5$, by what factor do the odds of $y = 1$ change when x increases by 1?

Working Space

Answer on Page 59

Exercise 4 **From odds to probability**

A model gives log odds of 0.8 for a particular student passing. What are the odds, and what is the probability of passing?

Working Space

Answer on Page 59

Exercise 5 Compute a probability

In the exam example, find $P(\text{pass})$ for a student who studies 6 hours and did not complete the prior course.

Working Space

Answer on Page 59

Exercise 6 Divide by 4

In the exam example, the first student has $P(\text{pass}) = 0.5$. Use the divide-by-4 rule to estimate their probability of passing if they study one more hour. Then compute it exactly.

Working Space

Answer on Page 59

Exercise 7 Log loss by hand

A model predicts $P(\text{male}) = 0.95$ for a penguin that is female. What is its log loss? What would the loss be if the model had said 0.5 instead?

Working Space

Answer on Page 60

Exercise 8 **One feature at a time**

Change `logistic.py` to use only body mass, then only bill depth. Report the test accuracy of each. Was it worth using both?

*Working Space**Answer on Page 60***Exercise 9** **Where is the line?**

Using the printed θ from `logistic.py`, find the body mass at which an Adelie with an average bill depth has $P(\text{male}) = 0.5$. (The training means and standard deviations are 3720 g and 474 g for mass.)

*Working Space**Answer on Page 60*

Evaluating Classification Systems

A model that predicts “no one has the flu” is right most days in July. That does not make it a good model. To judge a classifier, you need to count not just how often it is right, but *how* it is wrong. This chapter shows you how, first for a single yes-or-no decision rule and then across every rule the classifier could use.

2.1 The Confusion Matrix

A *confusion matrix* compares a classifier’s predictions with the true classes. For a binary problem, one class is called positive and the other negative. The four cells are:

	Actual Positive	Actual Negative
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

A false positive is a false alarm. A false negative is a missed case.

2.2 Performance Metrics

Each metric answers a different question:

- **Accuracy** is the fraction of all predictions that are correct: $\frac{TP+TN}{TP+TN+FP+FN}$.
- **Precision** is the fraction of positive predictions that are correct: $\frac{TP}{TP+FP}$.
- **Recall** (also called sensitivity) is the fraction of actual positives that are found: $\frac{TP}{TP+FN}$.
- **Specificity** is the fraction of actual negatives that are correctly identified: $\frac{TN}{TN+FP}$.
- **F₁ score** is the harmonic mean of precision and recall: $F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$.

2.3 A Worked Example

A screening test is run on 200 people. The results are TP = 40, FP = 10, FN = 20, and TN = 130. Then:

- Accuracy = $\frac{40+130}{200} = 0.85$.
- Precision = $\frac{40}{50} = 0.80$. Of the 50 people flagged, 40 were truly positive.
- Recall = $\frac{40}{60} \approx 0.667$. The test found 40 of the 60 true positives.
- Specificity = $\frac{130}{140} \approx 0.929$.
- $F_1 = \frac{2(0.80)(0.667)}{0.80+0.667} \approx 0.727$.

The accuracy looks good, but the test misses one third of the people who have the condition. For this reason, accuracy alone can hide a poor classifier, especially when one class is rare.

2.4 Choosing a Metric

The right metric depends on the cost of each error. For a screening test, a missed case (a false negative) is often the more serious error, so recall matters most. For a spam filter, a false positive means a real email is lost, so precision matters most. The F_1 score balances the two, and it is useful when both matter. Think of a smoke detector. A false positive means burnt toast sets off the alarm. A false negative means a real fire goes unnoticed. The second error is far worse, so smoke detectors are built for high recall, and we put up with the occasional toast alarm.

2.5 Grading a Chinstrap Detector

Chinstraps are the rarest species in the penguin data: 68 of 333. Suppose a researcher wants a program that flags Chinstraps from two quick measurements, bill length and bill depth. The program `evaluate.py` trains logistic regression, as in the last chapter, on 233 penguins and grades it on the other 100:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 y = (data["species"] == "Chinstrap").astype(int)    # 1 = Chinstrap
6 features = np.column_stack([data["bill_length_mm"], data["bill_depth_mm"]])
7 n = len(y)
8
9 order = np.random.default_rng(1).permutation(n)
10 train, test = order[:233], order[233:]
11 mean, sd = features[train].mean(axis=0), features[train].std(axis=0)
12 X = np.column_stack([np.ones(n), (features - mean) / sd])
13
14 # Logistic regression, as in the last chapter

```

```

15 def sigmoid(z):
16     return 1 / (1 + np.exp(-z))
17
18 theta = np.zeros(3)
19 for _ in range(3000):
20     p = sigmoid(X[train] @ theta)
21     theta = theta - 0.5 * X[train].T @ (p - y[train]) / len(train)
22
23 scores = sigmoid(X[test] @ theta)      # P(Chinstrap) for each test penguin
24 actual = y[test]
25 print(f"Test penguins: {len(actual)}, Chinstraps: {actual.sum()}")
26
27 def confusion(actual, predicted):
28     tp = np.sum((predicted == 1) & (actual == 1))
29     fp = np.sum((predicted == 1) & (actual == 0))
30     fn = np.sum((predicted == 0) & (actual == 1))
31     tn = np.sum((predicted == 0) & (actual == 0))
32     return tp, fp, fn, tn
33
34 tp, fp, fn, tn = confusion(actual, (scores >= 0.5).astype(int))
35 print(f"TP={tp} FP={fp} FN={fn} TN={tn}")
36 print(f"accuracy {(tp + tn) / len(actual):.2f}")
37 print(f"precision {tp / (tp + fp):.2f}")
38 print(f"recall {tp / (tp + fn):.2f}")

```

```

Test penguins: 100, Chinstraps: 26
TP=14 FP=1 FN=12 TN=73
accuracy 0.87
precision 0.93
recall 0.54

```

	Actually Chinstrap	Actually not
Predicted Chinstrap	TP = 14	FP = 1
Predicted not	FN = 12	TN = 73

An accuracy of 87% sounds good until you remember that a program that never says “Chinstrap” would score 74%. The confusion matrix tells the real story. When the detector says “Chinstrap,” it is almost always right (precision 0.93), but it finds only 14 of the 26 Chinstraps (recall 0.54). It is a very cautious detector.

2.6 Beyond a Single Threshold

A weather app that says “70% chance of rain” has not decided anything yet. You decide: maybe you bring an umbrella above 50%, while your friend waits for 80%. Same forecast, different thresholds, different mistakes.

Many classifiers work the same way. They output a score, such as a probability, and predict

the positive class when the score is above a threshold. Every number in the previous sections, from accuracy to F_1 , comes from one confusion matrix, and that matrix depends on the threshold. Change the threshold and you get a new matrix. The measures below look across *all* thresholds at once, so they grade the forecast itself.

2.6.1 The Precision–Recall Tradeoff

Lowering the threshold catches more true positives, so recall rises. It also flags more false positives, so precision usually falls. Raising the threshold does the opposite. A *precision–recall curve* plots precision against recall as the threshold changes. It is most informative when positives are rare, because accuracy then favors predicting negative.

2.6.2 The ROC Curve and AUC

The *receiver operating characteristic* (ROC) curve plots the true positive rate (recall) against the false positive rate $\frac{FP}{FP+TN}$ as the threshold changes. A perfect classifier passes through the top-left corner. A random guess lies on the diagonal. Back to the smoke detector: a sensitive setting rings at every candle (high true positive rate, but also a high false positive rate), and a dull setting misses small fires. Each setting of the threshold gives one point on the ROC curve.

The *area under the ROC curve* (AUC) summarizes the curve in one number. There is an equivalent meaning: the AUC is the probability that a randomly chosen positive example gets a higher score than a randomly chosen negative example. Ties count as half.

A Worked Example

Two positive examples have scores 0.9 and 0.6. Three negative examples have scores 0.7, 0.2, and 0.1. There are $2 \times 3 = 6$ pairs. The positive 0.9 beats all three negatives, which is 3 pairs. The positive 0.6 beats 0.2 and 0.1, which is 2 pairs, and loses to 0.7. So 5 of the 6 pairs are correctly ordered, and

$$AUC = \frac{5}{6} \approx 0.833$$

You can also see this as an area. Sweep the threshold down through the sorted scores 0.9 (P), 0.7 (N), 0.6 (P), 0.2 (N), 0.1 (N). Each positive moves the curve up by $\frac{1}{2}$ and each negative moves it right by $\frac{1}{3}$:

$$(0, 0) \rightarrow (0, \frac{1}{2}) \rightarrow (\frac{1}{3}, \frac{1}{2}) \rightarrow (\frac{1}{3}, 1) \rightarrow (\frac{2}{3}, 1) \rightarrow (1, 1)$$

The area under this staircase is $\frac{1}{3} \cdot \frac{1}{2} + \frac{2}{3} \cdot 1 = \frac{5}{6}$, the same answer.

2.6.3 Curves for the Chinstrap Detector

Figure 2.1 shows why the detector is so cautious: 12 of the 26 Chinstraps get scores below the default threshold of 0.5, a few of them below 0.1.

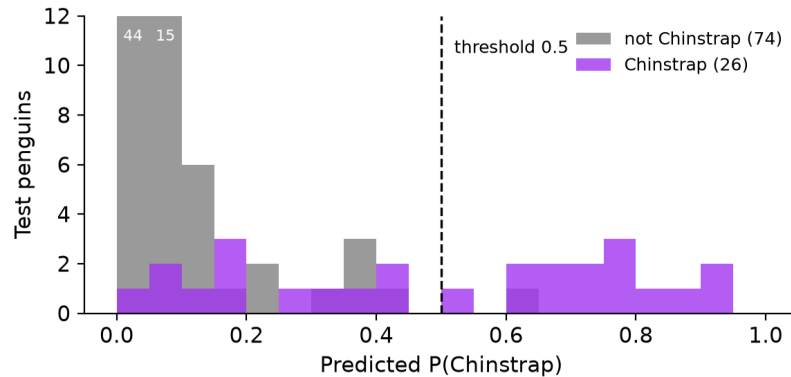


Figure 2.1: Predicted probabilities for the 100 test penguins. The two tallest gray bars are cut off; their heights are printed on them.

The rest of `evaluate.py` sweeps the threshold and computes the ROC curve and its area. To trace the ROC curve, sort the penguins from highest score to lowest and lower the threshold past them one at a time, exactly like the staircase above: each Chinstrap steps up, and each other penguin steps right. `np.cumsum` counts the steps, and `np.trapezoid` finds the area under the result.

```

1 # Sweep the threshold
2 print("threshold TP FP FN TN precision recall")
3 for threshold in [0.1, 0.2, 0.3, 0.5, 0.7, 0.9]:
4     tp, fp, fn, tn = confusion(actual, (scores >= threshold).astype(int))
5     print(f"threshold:{threshold:9.1f} {tp:3d} {fp:3d} {fn:3d} {tn:3d}"
6           f" {tp / (tp + fp):9.2f} {tp / (tp + fn):6.2f}")
7
8 # ROC curve: lower the threshold past each score in turn
9 ranked = actual[np.argsort(-scores)] # labels, highest score first
10 tpr = np.concatenate([[0], np.cumsum(ranked) / ranked.sum()])
11 fpr = np.concatenate([[0], np.cumsum(1 - ranked) / (1 - ranked).sum()])
12 print(f"AUC (area) = {np.trapezoid(tpr, fpr):.3f}")
13
14 # AUC as the fraction of (positive, negative) pairs ranked correctly
15 pos, neg = scores[actual == 1], scores[actual == 0]
16 print(f"AUC (pairs) = {np.mean(pos[:, None] > neg[None, :]):.3f}")

```

threshold	TP	FP	FN	TN	precision	recall
0.1	23	15	3	59	0.61	0.88
0.2	19	8	7	66	0.70	0.73

0.3	18	6	8	68	0.75	0.69
0.5	14	1	12	73	0.93	0.54
0.7	9	0	17	74	1.00	0.35
0.9	2	0	24	74	1.00	0.08

AUC (area) = 0.921
AUC (pairs) = 0.921

Read down the table: as the threshold rises, precision climbs to 1.00 and recall falls to 0.08. If missing a Chinstrap is expensive (say the researcher is counting a declining population), a threshold of 0.2 or 0.3 finds about 70% of them, at the cost of a few false alarms. Both ways of computing the AUC agree, as promised.

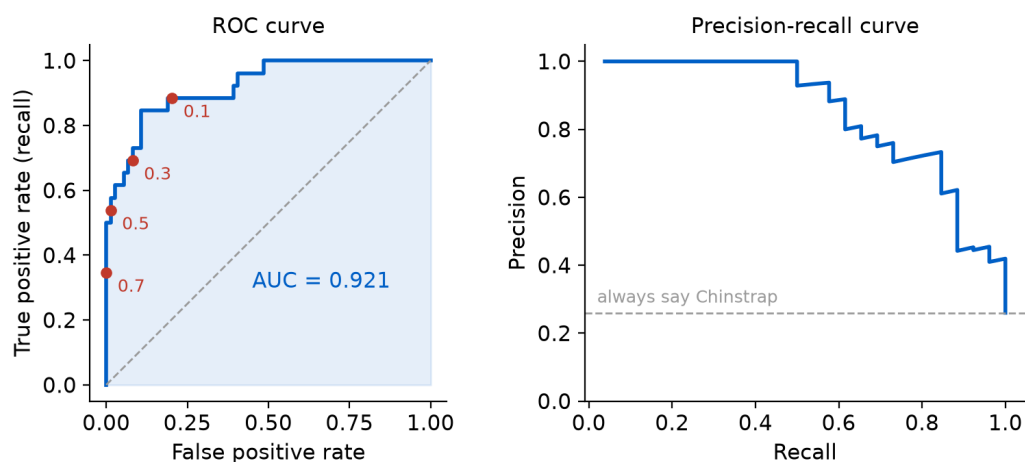


Figure 2.2: Left: the ROC curve, with the points for four thresholds. Right: the precision–recall curve. The dashed line is a detector that always says “Chinstrap,” whose precision is the fraction of Chinstraps, 0.26.

2.7 Which Measure to Use?

When classes are balanced, ROC and AUC are reasonable summaries. When positives are rare, the precision–recall curve is more informative, because the false positive rate stays small even when there are many false positives. Always report the threshold used for a final decision, since no single number describes every threshold.

Exercise 10 Compute the metrics

A classifier has $TP = 30$, $FP = 5$, $FN = 15$, and $TN = 50$. Compute the accuracy

Working Space

and the precision.

Answer on Page 60

Exercise 11 Which error matters?

A test detects a dangerous disease. Which metric should be the main goal, and why?

Working Space

Answer on Page 60

Exercise 12 Accuracy can mislead

In a dataset where 95% of cases are negative, a classifier always predicts negative. What is its accuracy, and what is its recall?

Working Space

Answer on Page 60

Exercise 13 Compute the AUC

Positive scores are 0.8, 0.4. Negative scores are 0.5, 0.3. Find the AUC.

Working Space

Answer on Page 61

Exercise 14 **Moving the threshold**

Use the scores from the ROC worked example: positives 0.9 and 0.6, negatives 0.7, 0.2, and 0.1. Predict positive when the score is at least the threshold. For a threshold of 0.5 and then 0.8, find TP, FP, FN, and TN, along with the precision, the recall, and the false positive rate.

Working Space

Answer on Page 61

Exercise 15 **The best F1**

F_1 can also be written $\frac{2TP}{2TP+FP+FN}$. Use the threshold table for the Chinstrap detector to find F_1 at thresholds 0.1, 0.3, 0.5, and 0.7. Which threshold has the highest F_1 ?

Working Space

Answer on Page 61

Exercise 16 **Three classes**

A species classifier is tested on 100 penguins. Rows are the predicted species and columns are the actual species:

Working Space

	Adelie	Chinstrap	Gentoo
Adelie	40	6	0
Chinstrap	3	14	0
Gentoo	1	0	36

Find the accuracy, and the precision and recall for Chinstrap.

Answer on Page 61

Exercise 17 A better detector

Change `evaluate.py` to use bill length and body mass instead of bill length and bill depth. What happens to the confusion matrix at threshold 0.5, and to the AUC? Why might body mass help?

Working Space

Answer on Page 61

The k-Nearest Neighbor Classifier

New to a school and not sure which lunch table to sit at? You might look at who you resemble: same classes, same sports, same music. That is the whole idea of the *k-nearest neighbor* (k-NN) classifier. To label something new, find the k labeled examples most like it and go with the majority.

It is the simplest classifier in this unit. There are no coefficients and no gradient descent. Training consists of storing the data, and all the work happens when you ask for a prediction. This is called *instance-based* learning.

3.1 The Algorithm

To classify a new point $\mathbf{x}$:

1. Compute the distance from $\mathbf{x}$ to every training point.
2. Find the k training points with the smallest distances.
3. Predict the most common class among those k points.

For regression, step 3 becomes “predict the average label of those k points.”

The usual distance is the Euclidean distance. For two points with p features,

$$d(\mathbf{x}, \mathbf{z}) = \sqrt{\sum_{j=1}^p (x_j - z_j)^2}$$

3.2 A Worked Example

Six labeled points are given in the plane:

$$A : (1, 1), \quad A : (2, 1), \quad A : (4, 1), \quad B : (2, 3), \quad B : (5, 4), \quad B : (6, 5)$$

Classify the new point $\mathbf{x} = (1, 2.5)$.

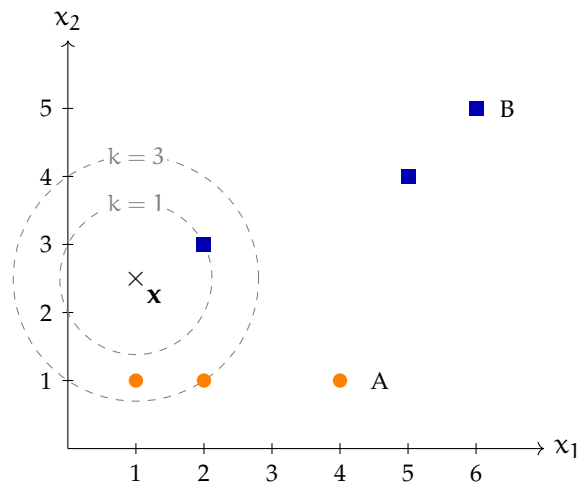


Figure 3.1: The new point x (cross), class A (circles), and class B (squares). The inner circle reaches the single nearest neighbor; the outer one reaches the three nearest.

The three nearest points are:

Point	Class	Distance from (1, 2.5)
(2, 3)	B	$\sqrt{1^2 + 0.5^2} \approx 1.118$
(1, 1)	A	$\sqrt{0^2 + 1.5^2} = 1.5$
(2, 1)	A	$\sqrt{1^2 + 1.5^2} \approx 1.803$

With $k = 1$, the nearest point is labeled B, so the prediction is B. With $k = 3$, the vote is B, A, A, so the prediction is A. The choice of k changed the answer.

3.3 Penguins, Neighbor by Neighbor

Back to the penguins, and to the question from the Classification and Regression chapter: which species is this, judging by its bill? We use two features, bill length and bill depth, so we can draw the results.

Two lessons from earlier chapters apply directly. First, k -NN is all about distance, so we **standardize**, as in the Preparing Data for Models chapter, using the training penguins' means and standard deviations. Second, k is a choice we make, not something the algorithm learns, so we pick it by **cross-validation**, as in the Training, Validation, and Testing chapter, and save a test set for the very end. Here is `knn.py`:

```

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) # y is 0, 1, or 2
6 features = np.column_stack([data["bill_length_mm"], data["bill_depth_mm"]])
7 n = len(y)
8
9 order = np.random.default_rng(1).permutation(n)
10 dev, test = order[:266], order[266:] # 266 to develop with, 67 to test
11
12 def knn_predict(X_train, y_train, X_query, k):
13     # Standardize using the training points only
14     mean, sd = X_train.mean(axis=0), X_train.std(axis=0)
15     A = (X_train - mean) / sd
16     Q = (X_query - mean) / sd
17     # Squared distance from every query point to every training point
18     dist = np.sum((Q[:, None, :] - A[None, :, :])**2, axis=2)
19     nearest = np.argsort(dist, axis=1)[:k] # indices of the k closest
20     votes = y_train[nearest] # their labels
21     return np.array([np.bincount(row, minlength=3).argmax() for row in votes])
22
23 # 5-fold cross-validation on the development penguins
24 folds = np.array_split(dev, 5)
25 for k in [1, 3, 5, 15, 51, 101]:
26     scores = []
27     for i in range(5):
28         val = folds[i]
29         fit = np.concatenate([folds[j] for j in range(5) if j != i])
30         predicted = knn_predict(features[fit], y[fit], features[val], k)
31         scores.append(np.mean(predicted == y[val]))
32     print(f"k = {k:3d}: cross-validation accuracy {np.mean(scores):.3f}")
33
34 # Final check with the chosen k
35 predicted = knn_predict(features[dev], y[dev], features[test], 3)
36 print(f"Test accuracy with k = 3: {np.mean(predicted == y[test]):.3f}")
37
38 matrix = np.zeros((3, 3), dtype=int)
39 for p, a in zip(predicted, y[test]):
40     matrix[p, a] += 1
41 print("Rows: predicted, columns: actual", names)
42 print(matrix)

```

k = 1: cross-validation accuracy 0.959

k = 3: cross-validation accuracy 0.981

k = 5: cross-validation accuracy 0.977

k = 15: cross-validation accuracy 0.974

k = 51: cross-validation accuracy 0.944

k = 101: cross-validation accuracy 0.857

Test accuracy with k = 3: 0.925

Rows: predicted, columns: actual ['Adelie' 'Chinstrap' 'Gentoo']

```
[[28  3  0]
```

```
 [ 0 15  0]
```

```
 [ 0  2 19]]
```

A few NumPy details are worth slowing down for:

- `np.unique(..., return_inverse=True)` turns the species names into the numbers 0, 1, and 2.
- `Q[:, None, :] - A[None, :, :]` uses broadcasting to subtract every training point from every query point at once. The result has shape (queries, training points, features), so summing over the last axis gives a whole table of squared distances. (We skip the square root, since it does not change which points are nearest.)
- `np.argsort` along each row lists the training points from nearest to farthest; we keep the first k .
- `np.bincount` counts the votes for each class, and `argmax` picks the winner. (Ties go to the lower class number. Odd values of k avoid ties between two classes, but with three classes, ties can still happen.)

Cross-validation picks $k = 3$, and on the 67 test penguins it is right 92.5% of the time, a bit lower than the cross-validation estimate. With only 67 test penguins, each mistake costs 1.5 percentage points, so 5 mistakes instead of 1 or 2 is within the range of luck. The confusion matrix shows where the mistakes are: all five are Chinstraps, called Adelie or Gentoo.

3.4 Choosing k

Figure 3.2 colors every point in the plane by the species k -NN would predict there.

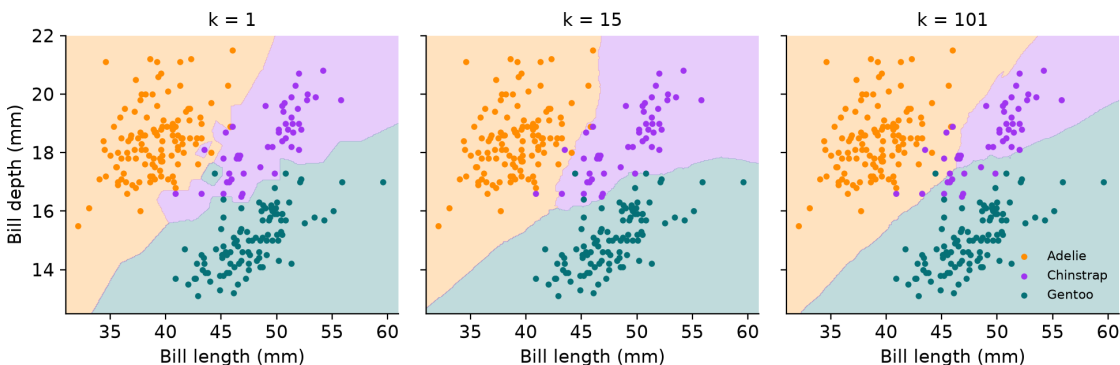


Figure 3.2: Predicted species across the plane for three values of k , with the 266 development penguins.

With $k = 1$, the boundary twists around individual penguins; one odd Gentoo carves out its own island. That is overfitting: on the training data itself, $k = 1$ is always 100% accurate, because each penguin is its own nearest neighbor. With $k = 101$, the boundaries

are smooth, but the Chinstrap region has shrunk. Chinstraps are only about a fifth of the penguins, so in a vote of 101 neighbors, the more common species win even near the Chinstrap cluster. That is underfitting. In between, $k = 15$ looks about right.

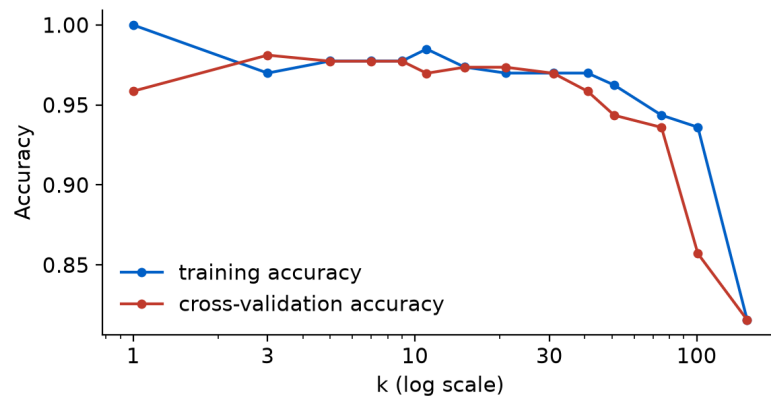


Figure 3.3: Training and cross-validation accuracy for different values of k .

	Small k	Large k
Boundary	jagged, follows single points	smooth
Training accuracy	very high (100% at $k = 1$)	lower
Risk	overfitting: noise counts	underfitting: rare classes get outvoted

3.5 Strengths and Weaknesses

- **Simple and flexible.** k -NN can draw boundaries of any shape, which logistic regression, with its straight line, cannot.
- **Slow to predict.** Each prediction measures the distance to every training point. The test above computed $67 \times 266 = 17,822$ distances; with a million training points, every single prediction needs a million. Clever data structures help, but this is the main cost.
- **Needs scaling.** Distances mean nothing if one feature is measured in grams and another in millimeters.
- **Struggles with many features.** In high dimensions, every point is far from every other point, and “nearest” stops meaning much. This is called the *curse of dimensionality*.

Exercise 18 One neighbor

Using the six points in the worked ex-

Working Space

ample, classify $\mathbf{x} = (3, 1.5)$ with $k = 1$.

Answer on Page 62

Exercise 19 **Three neighbors**

Using the same point, $(3, 1.5)$, and $k = 3$, what is the prediction?

Working Space

Answer on Page 62

Exercise 20 **Why scale?**

A data set has height in centimeters and income in dollars. Explain why the distances are dominated by income.

Working Space

Answer on Page 62

Exercise 21 **Six neighbors**

In the worked example, what does $k = 6$ predict for any new point? What does that say about very large k ?

Working Space

Answer on Page 62

Exercise 22 **Nearest-neighbor regression**

Write a version of `knn_predict` that returns the *average* body mass of the k nearest neighbors instead of a vote. Using flipper length, bill length, and bill depth, train on the 266 development penguins and find the test RMSE for $k = 1, 5, 20, 50$. How does it compare with a least squares fit on the same three features, which has a test RMSE of 359 g?

Working Space

Answer on Page 62

Stemming and Lemmatization

Search your phone for “running” and you probably want the songs that say “run” and “runs” too. Your phone does not understand English, so how does it know those words go together? Before a program can do anything with text, whether searching it, sorting it, or deciding that a message is spam, it has to turn the text into something it can count. This chapter shows how, and it ends with every message in a small data set turned into a row of numbers, ready for the spam filter in the next chapter.

4.1 From Text to Words

Every text-processing program starts with a few cleanup steps. Here is what happens to one message:

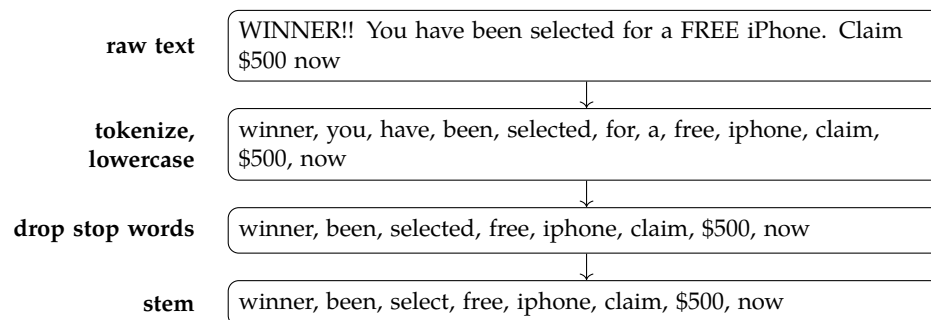


Figure 4.1: Preparing a message for a computer.

- **Tokenizing** splits text into words, called *tokens*. We also lowercase everything, so “FREE” and “free” count as the same word, and drop punctuation.
- **Stop words** like “the,” “a,” and “you” appear in almost every message, so they tell you little about any particular one. Many programs remove them.
- **Stemming or lemmatization** merges different forms of the same word, which is the subject of the rest of the chapter.

These steps are called *text normalization*.

4.2 Stemming

Stemming reduces a word to its *stem* by chopping off endings. A stemmer would reduce “jumping,” “jumped,” and “jumps” all to “jump.”

The stem does not have to be a real word. The most famous stemmer, written by Martin Porter in 1980, turns both “studies” and “studying” into “studi.” That is fine: the point of a stem is to group related words, not to be pretty. Stemmers are fast and simple, because they follow spelling rules and never look anything up.

4.3 Lemmatization

Lemmatization reduces a word to its *lemma*, the form you would look up in a dictionary: “running” and “ran” both become “run,” and “better” becomes “good.” A lemmatizer needs a large dictionary of word forms, and often needs to know how the word is used in the sentence. (Is “saw” the past tense of “see,” or a tool?) That makes it slower and more complicated, but more accurate.

Word	Typical stem	Lemma
jumping	jump	jump
studies	studi	study
running	run	run
ran	ran	run
better	better	good
mice	mice	mouse

Example: Would a stemmer or a lemmatizer group “ran” with “running”?

Solution: Only the lemmatizer. “Ran” has no ending to chop, so a stemmer leaves it alone, and “ran” and “run” are spelled too differently for any spelling rule to connect them. A lemmatizer has “ran” in its dictionary as a form of “run.”

4.4 A Stemmer in Python

Real stemmers have dozens of rules. Ours has six, but it shows the idea. Here is `text.py`, which you will use again in the next chapter:

```

1  """Tools for turning text into lists of word stems."""
2  import re
3
4  STOP_WORDS = {"a", "an", "and", "are", "at", "be", "by", "can", "do", "for",
5               "from", "have", "i", "in", "is", "it", "me", "my", "of", "on",
```

```

6         "or", "so", "the", "this", "to", "we", "you", "your"}
7
8 SUFFIXES = [("sses", "ss"), ("ies", "i"), ("ing", ""), ("ed", ""),
9             ("ly", ""), ("s", "")]
10
11 def tokenize(text):
12     """Split text into lowercase words, dropping punctuation."""
13     return re.findall(r"[a-z0-9$]+", text.lower())
14
15 def stem(word):
16     """Chop one common suffix off a word, as long as 3 letters remain."""
17     for suffix, replacement in SUFFIXES:
18         if word.endswith(suffix) and len(word) - len(suffix) >= 3:
19             word = word[:len(word) - len(suffix)] + replacement
20             # running -> runn -> run
21             if suffix in ("ing", "ed") and word[-1] == word[-2] and word[-1] not in
22                 ↪ "lsz":
23                 word = word[:-1]
24             return word
25     return word
26
27 def prepare(text):
28     """Tokenize, drop stop words, and stem."""
29     return [stem(w) for w in tokenize(text) if w not in STOP_WORDS]

```

`re.findall` uses a *regular expression*, a tiny pattern language for text. The pattern `[a-z0-9$]+` means “one or more letters, digits, or dollar signs in a row,” so everything else, including spaces and punctuation, acts as a separator. (We keep `$` because it turns out to be a useful clue for spotting spam.)

The order of the suffix list matters: the stemmer uses the first rule that fits, so “classes” becomes “class” (by the “sses” rule) instead of “classe” (by the “s” rule). The “at least 3 letters” check keeps it from turning “is” into “i” or “sing” into “s.” Try it:

```

1 from text import stem
2 words = ["jumping", "jumped", "jumps", "studies", "running", "ran",
3         "classes", "happily", "connected", "connection"]
4 print([stem(w) for w in words])

```

```
['jump', 'jump', 'jump', 'studi', 'run', 'ran', 'class', 'happi', 'connect',
'connection']
```

Just like Porter’s stemmer, ours turns “studies” into “studi” and cannot connect “ran” with “run.” It also misses “connection,” because it has no rule for “-ion.”

4.5 Bag of Words

Now we can turn a whole collection of messages into numbers. The file `messages.csv` contains 60 short text messages of the kind a high school student might get, each labeled spam (22 messages) or ham (38 messages; “ham” is the traditional name for messages that are not spam).

The simplest numerical description of a message is a *bag of words*: count how many times each stem appears, and forget the order. “Dog bites man” and “man bites dog” have the same bag. That sounds like a big loss, but for many tasks, including spam filtering, which words appear matters much more than their order.

Here is `bag_of_words.py`:

```

1 import csv
2 import numpy as np
3 from text import tokenize, stem, prepare, STOP_WORDS
4
5 with open("messages.csv", newline="") as f:
6     rows = list(csv.DictReader(f))
7     texts = [row["text"] for row in rows]
8     print("Messages:", len(texts))
9
10 # How big is the vocabulary after each step?
11 raw = {w for t in texts for w in t.split()}
12 lower = {w for t in texts for w in tokenize(t)}
13 no_stop = {w for w in lower if w not in STOP_WORDS}
14 stems = {stem(w) for w in no_stop}
15 for name, words in [("split on spaces", raw), ("lowercase, no punctuation", lower),
16                    ("without stop words", no_stop), ("stemmed", stems)]:
17     print(f"{name:26s} {len(words):4d} distinct words")
18
19 # Bag of words: one row per message, one column per stem
20 vocab = sorted(stems)
21 column = {word: j for j, word in enumerate(vocab)}
22 counts = np.zeros((len(texts), len(vocab)), dtype=int)
23 for i, t in enumerate(texts):
24     for word in prepare(t):
25         counts[i, column[word]] += 1
26 print("Count matrix:", counts.shape)
27
28 # The first message as counts
29 print({vocab[j]: int(counts[0, j]) for j in np.nonzero(counts[0])[0]})
30
31 # The most common stems overall
32 totals = counts.sum(axis=0)
33 top = np.argsort(-totals)[:8]
34 print("Most common:", [(vocab[j], int(totals[j])) for j in top])

```

```

Messages: 60
split on spaces          368 distinct words
lowercase, no punctuation 311 distinct words
without stop words       285 distinct words
stemmed                  271 distinct words
Count matrix: (60, 271)
{'been': 1, 'claim': 1, 'click': 1, 'free': 1, 'iphone': 1, 'link': 1,
'now': 1, 'prize': 1, 'select': 1, 'winner': 1}
Most common: [('free', 15), ('now', 11), ('claim', 9), ('click', 8), ('today', 6),
('earn', 5), ('cash', 5), ('before', 4)]

```

(Two long output lines are wrapped here to fit the page.)

Step	Distinct words
split on spaces	368
lowercase, no punctuation	311
without stop words	285
stemmed	271

The first message, “WINNER!! You have been selected for a FREE iPhone. Click the link to claim your prize now,” has become ten counts of 1, one for each stem it contains. Each step shrinks the vocabulary by merging things that should be the same. “FREE,” “free,” and “free!!” are one word now. The result is a 60×271 matrix of counts, with one row per message. Most entries are zero, since each message uses only a handful of the 271 stems; a matrix like that is called *sparse*.

Look at the most common stems: free, now, claim, click, cash. Every one of them smells like spam. That is exactly the pattern the naive Bayes classifier in the next chapter will learn.

4.6 Where These Ideas Are Used

Search engines stem both your query and their documents, so “connected” finds “connecting.” Spam filters and other text classifiers use bags of words as features. Modern language models, like the ones behind chatbots, skip stemming and split text into pieces of words called subword tokens, but the first step is the same: turn text into numbers a program can work with.

Exercise 23 Stem or lemma?

For each pair of words, say whether a stemmer would likely give them the same stem, and whether a lemmatizer

Working Space

would give them the same lemma: (a) “happy” and “happiness”; (b) “better” and “good”.

Answer on Page 63

Exercise 24 Search Results

A search engine stems every word in its documents and every word you type, using Porter’s stemmer. You search for “connected.” Which of these documents will match: one containing “connection,” one containing “connecting,” and one containing “linked”?

Working Space

Answer on Page 63

Exercise 25 Run the stemmer by hand

Use the stemming rules from this chapter on these words: dresses, hopping, filled, sing, quickly.

Working Space

Answer on Page 63

Exercise 26 A rule for -ion

Add a rule to SUFFIXES so that “connection” and “selection” stem to “connect” and “select.” Where in the list does it need to go, and does it break any word

Working Space

in the example list?

Answer on Page 63

Exercise 27 **A tiny lemmatizer**

Write a function `lemmatize(word)` that looks the word up in a small dictionary of irregular forms (ran, better, mice, was) and returns the word unchanged if it is not there. Why would a real lemmatizer need a dictionary with hundreds of thousands of entries?

Working Space

Answer on Page 63

Bayesian Classifiers

A text arrives: “Congratulations! You won a \$1000 gift card. Claim your reward before it expires tonight.” You know it is spam before you finish reading. How? Certain words, like “won,” “claim,” and “gift card,” are far more common in spam than in texts from your friends. Each one nudges your belief a little further toward “spam.”

That nudging has a precise mathematical form: Bayes’ rule. This chapter builds two classifiers on it. The first is a spam filter for the messages from the last chapter. The second identifies penguin species from measurements, and it shows what to do when the features are numbers instead of words.

5.1 Bayes’ Rule for Classification

We want the probability of each class C given the features $\mathbf{x}$ we observed. Bayes’ rule says

$$P(C | \mathbf{x}) = \frac{P(\mathbf{x} | C) P(C)}{P(\mathbf{x})}$$

Each piece has a name:

Term	Name	Meaning for spam
$P(C)$	prior	how common spam is before reading the message
$P(\mathbf{x} C)$	likelihood	how likely these words are in a spam message
$P(C \mathbf{x})$	posterior	how likely this message is spam, after reading it
$P(\mathbf{x})$	evidence	the same for every class; it just makes the posteriors add to 1

Since $P(\mathbf{x})$ is the same for every class, a classifier can compute $P(\mathbf{x} | C) P(C)$ for each class, pick the biggest, and divide by the total only if it wants actual probabilities.

5.2 The Naive Assumption

The hard part is the likelihood. A message is a list of words $w_1, w_2, \dots, w_m$, and there are far too many possible messages to estimate $P(w_1, \dots, w_m \mid \text{spam})$ directly. So we make a bold simplification: assume that, once you know the class, the words are independent of each other. Then the likelihood is a product of one-word probabilities:

$$P(w_1, \dots, w_m \mid C) = P(w_1 \mid C) P(w_2 \mid C) \cdots P(w_m \mid C)$$

This is the *naive Bayes* assumption. It is “naive” because it is false: in real spam, “gift” and “card” tend to appear together. But the classifier only needs to rank the classes correctly, not to get the probabilities exactly right, and naive Bayes is famous for working well anyway.

5.3 Estimating the Word Probabilities

From the training messages, $P(\text{spam})$ is just the fraction of messages that are spam: 22/60 in our data. For the word probabilities, count. Pour all the spam messages into one big bag of words; then $P(\text{free} \mid \text{spam})$ is the fraction of the words in that bag that are “free.”

After stemming and dropping stop words, the 22 spam messages contain 215 words, and the word “free” is 15 of them. The 38 ham messages contain 255 words, and none is “free.”

That zero is a problem. It would make $P(\text{free} \mid \text{ham}) = 0$, so any message containing “free” would have zero chance of being ham, no matter what else it said. One word would veto everything. The fix is *Laplace smoothing*: pretend you saw every word in the vocabulary one extra time in each class. With a vocabulary of $V = 271$ stems,

$$P(w \mid C) = \frac{\text{count of } w \text{ in class } C + 1}{\text{total words in class } C + V}$$

So $P(\text{free} \mid \text{spam}) = \frac{16}{486}$ and $P(\text{free} \mid \text{ham}) = \frac{1}{526}$: small, but not zero.

5.4 A Worked Example

The student council texts everyone: “Free pizza at the game tonight.” Is it spam?

After preparing the text, the stems are free, pizza, game, and tonight. “Pizza” never appears in the training messages, so the model knows nothing about it, and we skip it. The counts for the others:

Stem	In spam	In ham	$P(w \mid \text{spam})$	$P(w \mid \text{ham})$
free	15	0	$16/486$	$1/526$
game	1	3	$2/486$	$4/526$
tonight	1	2	$2/486$	$3/526$

Multiply the prior by the likelihoods:

$$\text{spam: } \frac{22}{60} \cdot \frac{16}{486} \cdot \frac{2}{486} \cdot \frac{2}{486} \approx 2.04 \times 10^{-7} \quad \text{ham: } \frac{38}{60} \cdot \frac{1}{526} \cdot \frac{4}{526} \cdot \frac{3}{526} \approx 5.22 \times 10^{-8}$$

Dividing by the total,

$$P(\text{spam} \mid \text{message}) = \frac{2.04 \times 10^{-7}}{2.04 \times 10^{-7} + 5.22 \times 10^{-8}} \approx 0.797$$

The filter calls it spam, almost entirely because of the word “free”: $P(\text{free} \mid \text{spam})$ is about 17 times $P(\text{free} \mid \text{ham})$, while “game” and “tonight” each lean toward ham by less than 2 to 1. A real filter would learn from more data, including legitimate messages about free pizza, and the same rule could balance that out.

5.5 Working in Logarithms

Multiplying many small probabilities quickly produces numbers too small for a computer to store. Each word probability here is around $\frac{1}{500}$, so a message of a few hundred words gives a product below 10^{-308} , the smallest number ordinary floating-point arithmetic can hold, and the computer rounds it to zero. The cure is to add logarithms instead of multiplying probabilities:

$$\ln [P(C) P(w_1 \mid C) \cdots P(w_m \mid C)] = \ln P(C) + \sum_{j=1}^m \ln P(w_j \mid C)$$

The class with the biggest log score is the class with the biggest product. To recover probabilities at the end, subtract the biggest log score from each score before calling `np.exp`, so the largest becomes $e^0 = 1$ and nothing underflows.

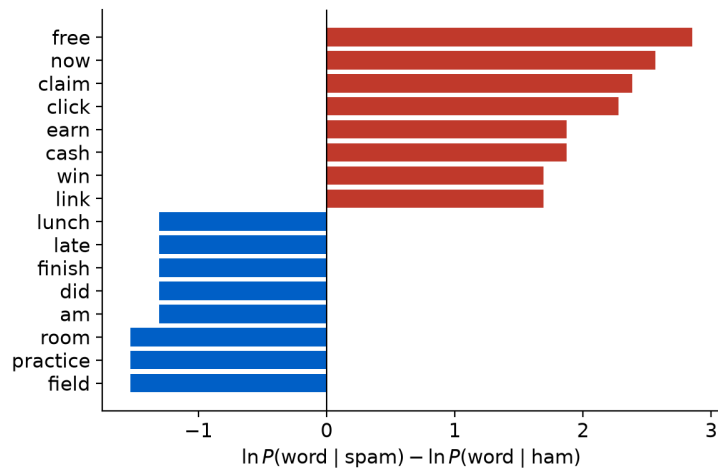


Figure 5.1: The eight stems that push hardest toward spam (red) and toward ham (blue). Each bar is how much one appearance of the word adds to the log score for spam, compared with ham.

5.6 A Spam Filter in Python

Put `messages.csv` and `text.py` from the last chapter next to `spam_filter.py`:

```

1  import csv
2  import numpy as np
3  from collections import Counter
4  from text import prepare
5
6  with open("messages.csv", newline="") as f:
7      rows = list(csv.DictReader(f))
8      texts = [row["text"] for row in rows]
9      labels = [row["label"] for row in rows]
10
11 def train(texts, labels):
12     counts = {"spam": Counter(), "ham": Counter()}
13     for text, label in zip(texts, labels):
14         counts[label].update(prepare(text))
15     vocab = set(counts["spam"]) | set(counts["ham"])
16     model = {}
17     for label in ("spam", "ham"):
18         total = sum(counts[label].values())
19         log_prior = np.log(labels.count(label) / len(labels))
20         # Laplace smoothing: add 1 to every count
21         log_like = {w: np.log((counts[label][w] + 1) / (total + len(vocab)))
22                     for w in vocab}
23         model[label] = (log_prior, log_like)
24     return model
25

```

```

26 def spam_probability(model, text):
27     scores = {}
28     for label, (log_prior, log_like) in model.items():
29         # Words the model has never seen are skipped
30         words = [w for w in prepare(text) if w in log_like]
31         scores[label] = log_prior + sum(log_like[w] for w in words)
32     # Turn the two log scores back into probabilities that add to 1
33     biggest = max(scores.values())
34     spam = np.exp(scores["spam"] - biggest)
35     ham = np.exp(scores["ham"] - biggest)
36     return spam / (spam + ham)
37
38 # Leave-one-out: train on 59 messages, test on the one left out, 60 times
39 correct = 0
40 for i in range(len(texts)):
41     model = train(texts[:i] + texts[i + 1:], labels[:i] + labels[i + 1:])
42     guess = "spam" if spam_probability(model, texts[i]) >= 0.5 else "ham"
43     correct += guess == labels[i]
44     if guess != labels[i]:
45         print("Missed:", labels[i], "|", texts[i])
46 print(f"Leave-one-out accuracy: {correct}/{len(texts)}")
47
48 model = train(texts, labels)
49 for text in ["Free pizza at the game tonight",
50             "Click here to claim your prize",
51             "Can you send me the notes from history class?"]:
52     print(f"{spam_probability(model, text):.3f} {text}")

```

```

Missed: ham | Is the essay supposed to be double spaced?
Missed: ham | My phone is about to die. Call me at home later
Leave-one-out accuracy: 58/60
0.797 Free pizza at the game tonight
0.999 Click here to claim your prize
0.010 Can you send me the notes from history class?

```

Counter is a dictionary that counts things; update adds one for each word in a list, and a missing word counts as 0. With only 60 messages, we cannot afford to hold out a big test set, so we use *leave-one-out* cross-validation: 60 rounds, each training on 59 messages and testing on the last one. (That is k-fold cross-validation with k equal to the number of examples.) The filter gets 58 of 60 right. Both misses are short messages made of words the training data barely contains, so the prior and a few weak clues decide. “Free pizza” gets 0.797, as we computed by hand.

5.7 Numbers Instead of Words

Bayes’ rule works for any features, as long as you can estimate $P(\mathbf{x} \mid C)$. For a number, like a penguin’s bill length, the standard choice is a normal distribution. For each species,

estimate the mean μ and standard deviation σ of each feature from the training penguins, and use the normal density

$$p(x | C) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/(2\sigma^2)}$$

as the likelihood. With the naive assumption, multiply the densities of the features. This is *Gaussian naive Bayes*.

From the same 266 development penguins as in the k-Nearest Neighbor chapter:

Species	Prior	Bill length (mm)		Bill depth (mm)	
		μ	σ	μ	σ
Adelie	0.444	39.05	2.68	18.43	1.18
Chinstrap	0.180	49.13	3.19	18.48	1.15
Gentoo	0.376	47.65	3.21	15.01	1.00

Example: A penguin has a 45 mm bill that is 17 mm deep. Which species is it?

Solution: Evaluate the normal densities (a calculator or `scipy.stats.norm.pdf` helps), multiply by the prior, and normalize:

Species	$p(45 C)$	$p(17 C)$	Prior $\times$ product	Posterior
Adelie	0.0126	0.1617	0.000902	0.213
Chinstrap	0.0541	0.1520	0.001482	0.350
Gentoo	0.0884	0.0557	0.001851	0.437
			sum 0.004235	

Gentoo wins, but not by much. The 45 mm bill is long for an Adelie (2.2 standard deviations above average), and the 17 mm depth is deep for a Gentoo. This penguin is hard to call.

5.8 When Features Move Together

The naive assumption says that, within a species, bill length tells you nothing about bill depth. That is not true: within Chinstraps, the correlation between the two is 0.78. A Chinstrap with a short bill usually has a shallow one, too.

The *multivariate normal distribution* handles this. Instead of a separate μ and σ for each feature, it has a mean vector μ and a *covariance matrix* Σ , whose diagonal holds the variances and whose off-diagonal entries hold the covariances. Its density for a vector x with d features is

$$p(x) = (2\pi)^{-d/2} \det(\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right)$$

For the Chinstraps, for example,

$$\mu = \begin{bmatrix} 49.13 \\ 18.48 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 10.16 & 2.87 \\ 2.87 & 1.32 \end{bmatrix}$$

Naive Bayes is the special case where all the off-diagonal entries are zero.

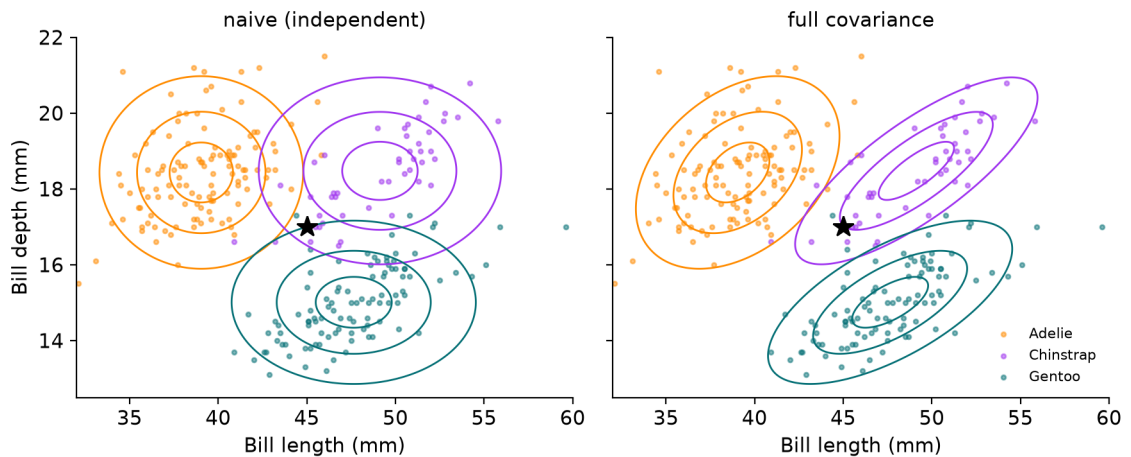


Figure 5.2: Contours of the estimated densities for each species. Left: naive Bayes, whose ovals can only point along the axes. Right: full covariance, whose ovals tilt with the data. The star is the mystery penguin.

In Figure 5.2, the tilted Chinstrap oval reaches down toward the mystery penguin. The program `gaussian_bayes.py` fits both versions and compares them with 5-fold cross-validation:

```

1 import numpy as np
2 from scipy import stats
3 from penguins import load_penguins
4
5 data = load_penguins()
6 names, y = np.unique(data["species"], return_inverse=True)
7 features = np.column_stack([data["bill_length_mm"], data["bill_depth_mm"]])
8 order = np.random.default_rng(1).permutation(len(y))
9 dev, test = order[:266], order[266:]
10
11 def fit(rows):
12     """For each species: prior, mean vector, and covariance matrix."""
13     model = []
14     for c in range(3):
15         group = features[rows][y[rows] == c]
16         model.append((len(group) / len(rows), group.mean(axis=0),
17                     np.cov(group.T, bias=True)))
18     return model
19 
```

```

20 def log_posteriors(model, X, naive):
21     scores = []
22     for prior, mean, cov in model:
23         if naive: # independent normal distributions, one per feature
24             sd = np.sqrt(np.diag(cov))
25             log_like = stats.norm.logpdf(X, mean, sd).sum(axis=1)
26         else: # one multivariate normal, with covariance
27             log_like = stats.multivariate_normal.logpdf(X, mean, cov)
28         scores.append(np.log(prior) + log_like)
29     return np.column_stack(scores)
30
31 def probabilities(scores):
32     p = np.exp(scores - scores.max(axis=1, keepdims=True))
33     return p / p.sum(axis=1, keepdims=True)
34
35 folds = np.array_split(dev, 5)
36 model = fit(dev)
37 mystery = np.array([[45.0, 17.0]]) # bill length 45 mm, depth 17 mm
38 for naive in (True, False):
39     accuracy = []
40     for i in range(5):
41         rest = np.concatenate([folds[j] for j in range(5) if j != i])
42         scores = log_posteriors(fit(rest), features[folds[i]], naive)
43         accuracy.append(np.mean(scores.argmax(axis=1) == y[folds[i]]))
44     guess = log_posteriors(model, features[test], naive).argmax(axis=1)
45     test_acc = np.mean(guess == y[test])
46     p = probabilities(log_posteriors(model, mystery, naive))
47     print("naive" if naive else "full ", f"CV {np.mean(accuracy):.3f}",
48           f"test {test_acc:.3f} mystery:", np.round(p, 3))

```

```

naive CV 0.936 test 0.925 mystery: [[0.213 0.35 0.437]]
full CV 0.977 test 0.925 mystery: [[0.033 0.954 0.013]]

```

`np.cov` expects one variable per row, hence the transpose; `bias=True` divides by n , matching the maximum likelihood estimate (NumPy's default divides by $n - 1$). The naive version uses only the diagonal of the covariance matrix.

The naive model reproduces our hand calculation. The full model is much more confident, and it changes its mind: Chinstrap, at 95%. Because Chinstrap bill length and depth rise and fall together, a Chinstrap with a shorter-than-average bill is expected to have a shallower bill, which is just what this penguin has. Taking the covariance into account raises the cross-validation accuracy from 93.6% to 97.7%. (On the 67 test penguins, the two happen to tie.)

	Naive Bayes	Full covariance
Numbers to estimate per class	2 per feature	grows with the square of the number of features
Needs much data?	no	yes, with many features
Handles correlated features?	no	yes
Typical use	text, with thousands of word features	a few numeric features

With 271 word features, a full covariance matrix would have $271^2 = 73,441$ entries, far more than 60 messages could estimate. That is why spam filters stay naive.

Exercise 28 Posterior by hand

A class A has prior 0.3 and class B has prior 0.7. For an observation $\mathbf{x}$, $P(\mathbf{x} | A) = 0.08$ and $P(\mathbf{x} | B) = 0.02$. Find $P(A | \mathbf{x})$.

Working Space

Answer on Page 64

Exercise 29 Without smoothing

If the filter did not use Laplace smoothing, what would it say about “Free pizza at the game tonight”? Why is that a bad idea?

Working Space

Answer on Page 64

Exercise 30 **Call me tonight**

Use the counts in this chapter to find the spam probability of “Call me tonight.” (“Me” is a stop word. “Call” appears once in spam and once in ham.)

Working Space

Answer on Page 64

Exercise 31 **How much smoothing?**

Change the 1 in the Laplace smoothing to a variable α (it must be added to every count, so the denominator becomes $\text{total} + \alpha * \text{len}(\text{vocab})$). Find the leave-one-out accuracy for $\alpha = 0.1, 0.5, 1, 2$, and 5.

Working Space

Answer on Page 64

Exercise 32 **A third feature**

Add flipper length as a third feature in `gaussian_bayes.py` (the mystery penguin’s flipper is 195 mm). What happens to the accuracies and to the

Working Space

mystery penguin?

Answer on Page 64

Bayesian Networks

You wake up and the lawn is wet. Did it rain, or did the sprinkler run? Your brain weighs the possibilities without effort. A Bayesian network does the same reasoning on paper, and it scales to problems with hundreds of variables, from medical diagnosis to deciding whether there is school today.

6.1 Variables and Dependencies

A *Bayesian network* is a probability model for several variables and the dependencies among them. Its structure is a *directed acyclic graph* (DAG): each node is a variable, and an arrow from A to B means A directly influences B. “Acyclic” means you can never follow the arrows in a loop back to where you started. Each node has a *conditional probability table* (CPT) giving its probabilities for each combination of its parents’ values.

The naive Bayes classifier from the last chapter is a small Bayesian network: one arrow from the class to each feature.

6.2 A Small Network

Let R be “it rained” and W be “the grass is wet,” with the arrow $R \rightarrow W$. The network needs one number for R (since $P(\neg R) = 1 - P(R)$) and two conditional probabilities for W:

$$P(R) = 0.2, \quad P(W | R) = 0.9, \quad P(W | \neg R) = 0.1$$

6.3 Inference

A Bayesian network answers questions about unknown variables given observed ones. Three kinds of reasoning are common.

- **Prediction** (causal): given a cause, how likely is the effect? Here, $P(W | R) = 0.9$.
- **Diagnosis** (evidential): given an effect, how likely is the cause? This needs Bayes’ rule.
- **Explaining away**: if two causes can produce the same effect, observing one cause lowers the probability of the other.

For the diagnosis question, first find the total probability of wet grass:

$$P(W) = P(W | R)P(R) + P(W | \neg R)P(\neg R) = 0.9(0.2) + 0.1(0.8) = 0.26$$

Then Bayes' rule gives

$$P(R | W) = \frac{P(W | R)P(R)}{P(W)} = \frac{0.18}{0.26} \approx 0.692$$

Seeing wet grass raises the probability of rain from 0.2 to about 0.69.

6.3.1 Explaining Away, with Numbers

Now add a sprinkler S , so the network is $R \rightarrow W \leftarrow S$. Let $P(R) = 0.2$ and $P(S) = 0.3$, independently, and

$$P(W | R, S) = 0.99, \quad P(W | R, \neg S) = 0.9, \quad P(W | \neg R, S) = 0.8, \quad P(W | \neg R, \neg S) = 0.05$$

Multiplying out the four cases where the grass is wet:

R	S	P(R, S, W)
yes	yes	$0.2 \cdot 0.3 \cdot 0.99 = 0.0594$
yes	no	$0.2 \cdot 0.7 \cdot 0.9 = 0.126$
no	yes	$0.8 \cdot 0.3 \cdot 0.8 = 0.192$
no	no	$0.8 \cdot 0.7 \cdot 0.05 = 0.028$

The total is $P(W) = 0.4054$. Knowing only that the grass is wet,

$$P(R | W) = \frac{0.0594 + 0.126}{0.4054} \approx 0.457$$

Now you also learn that the sprinkler was on. Keep only the rows where S is yes:

$$P(R | W, S) = \frac{0.0594}{0.0594 + 0.192} \approx 0.236$$

The sprinkler already explains the wet grass, so the probability of rain drops from 0.457 to 0.236, close to its starting value of 0.2.

6.4 Is There School Today?

It is 6 a.m. in January. Here is a network for the question every student cares about:

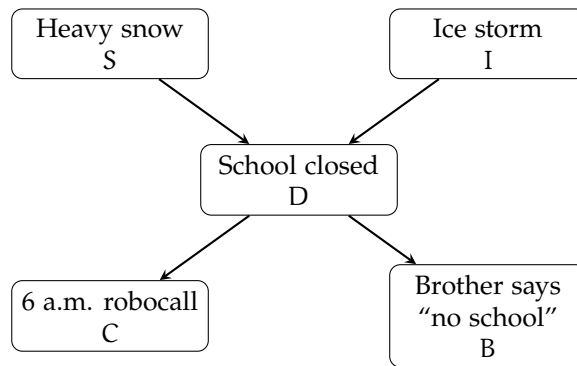


Figure 6.1: A Bayesian network for snow days.

Snow and ice can each close school. Whether school is closed affects whether the district’s robocall comes, and whether your little brother announces that there is no school (which he does fairly often anyway, hoping to make it true). The tables:

		S	I	P(D S, I)
Variable	P(true)	yes	yes	0.95
S	0.10	yes	no	0.70
I	0.08	no	yes	0.50
		no	no	0.01
		D		P(B D)
D	P(C D)	yes	yes	0.90
yes	0.98	no	no	0.30
no	0.02			

The joint distribution. The arrows say exactly how to build the probability of any complete combination of values: multiply each variable’s probability given its parents.

$$P(s, i, d, c, b) = P(s) P(i) P(d | s, i) P(c | d) P(b | d)$$

Five yes-or-no variables have $2^5 = 32$ combinations. Writing out all 32 probabilities would take 31 numbers (the last is 1 minus the rest), but the network needs only 10, the entries in the tables above. With hundreds of variables, that saving is the difference between impossible and easy.

Example: Before you hear anything, what is the probability that school is closed?

Solution: Add up over the four combinations of the parents:

S	I	P(S) P(I) P(D S, I)
yes	yes	$0.1 \cdot 0.08 \cdot 0.95 = 0.0076$
yes	no	$0.1 \cdot 0.92 \cdot 0.70 = 0.0644$
no	yes	$0.9 \cdot 0.08 \cdot 0.50 = 0.0360$
no	no	$0.9 \cdot 0.92 \cdot 0.01 = 0.00828$
		$P(D) = 0.11628$

About a 12% chance. Now your brother bursts in: “No school!” Bayes’ rule, as in the rain example:

$$P(D | B) = \frac{0.9 \times 0.11628}{0.9 \times 0.11628 + 0.3 \times 0.88372} = \frac{0.1047}{0.3698} \approx 0.283$$

His word raises the chance from 12% to 28%, which is not enough to go back to bed. The robocall, which almost never comes on a school day, is far more convincing:

$$P(D | C) = \frac{0.98 \times 0.11628}{0.98 \times 0.11628 + 0.02 \times 0.88372} \approx 0.866$$

Explaining away again. Suppose school is closed. Was there heavy snow? Use the table above: the rows where S is yes account for $0.0076 + 0.0644 = 0.072$ of the 0.11628, so

$$P(S | D) = \frac{0.072}{0.11628} \approx 0.619$$

Now you look outside and see that everything is coated in ice. Keep only the rows where I is yes:

$$P(S | D, I) = \frac{0.0076}{0.0076 + 0.0360} \approx 0.174$$

The ice storm explains the closing, so snow drops from 62% to 17%, which is not much above its starting value of 10%. Snow and ice are independent, but once you know school is closed, learning about one changes what you believe about the other.

6.5 Inference in NumPy

For a network this small, the simplest method is the one we used by hand, called *inference by enumeration*: build the whole joint table, keep the rows that match the evidence, and add. Here is `snow_day.py`:

```

1 import numpy as np
2
3 # Each table gives P(variable = True) for every combination of its parents.
4 # Axis order everywhere: S (snow), I (ice), D (school closed), C (robocall), B
5   ↪ (brother)
6 p_snow = 0.1
7 p_ice = 0.08
8 p_closed = np.array([[0.01, 0.5],                # index: S, I

```

```

8         [0.7, 0.95]])
9 p_call = np.array([0.02, 0.98])          # index: D
10 p_brother = np.array([0.3, 0.9])         # index: D
11
12 def table(p_true):
13     """Stack P(False) and P(True) along a new last axis."""
14     return np.stack([1 - p_true, p_true], axis=-1)
15
16 S = table(np.array(p_snow))              # shape (2,)
17 I = table(np.array(p_ice))               # shape (2,)
18 D = table(p_closed)                     # shape (2, 2, 2):   S, I, D
19 C = table(p_call)                       # shape (2, 2):     D, C
20 B = table(p_brother)                    # shape (2, 2):     D, B
21
22 # The joint distribution: P(s, i, d, c, b) = P(s) P(i) P(d|s,i) P(c|d) P(b|d)
23 joint = np.einsum("s,i,sid,dc,db->sidcb", S, I, D, C, B)
24 print("Joint table shape:", joint.shape, " total:", round(joint.sum(), 10))
25
26 AXES = "SIDCB"
27
28 def probability(query, given={}):
29     """P(query variable is True | given), by adding up rows of the joint table."""
30     keep = joint.copy()
31     for name, value in given.items():
32         index = [slice(None)] * 5
33         index[AXES.index(name)] = 1 - int(value)    # zero out rows that disagree
34         keep[tuple(index)] = 0
35     other = tuple(k for k in range(5) if k != AXES.index(query))
36     marginal = keep.sum(axis=other)
37     return marginal[1] / marginal.sum()
38
39 print(f"P(closed) = {probability('D'):.3f}")
40 print(f"P(closed | brother says so) = {probability('D', {'B': True}):.3f}")
41 print(f"P(closed | robocall) = {probability('D', {'C': True}):.3f}")
42 print(f"P(snow | closed) = {probability('S', {'D': True}):.3f}")
43 print(f"P(snow | closed, icy) = {probability('S', {'D': True, 'I':
44     ↪ True}):.3f}")
45 print(f"P(snow | closed, not icy) = {probability('S', {'D': True, 'I':
46     ↪ False}):.3f}")

```

```

Joint table shape: (2, 2, 2, 2, 2) total: 1.0
P(closed) = 0.116
P(closed | brother says so) = 0.283
P(closed | robocall) = 0.866
P(snow | closed) = 0.619
P(snow | closed, icy) = 0.174
P(snow | closed, not icy) = 0.886

```

Each variable gets its own axis, with index 0 for false and 1 for true. For example, `joint[1, 0, 1, 1, 0]` is $P(S, \neg I, D, C, \neg B)$. The star of the program is `np.einsum`, short for “Einstein summation.” Its first argument names the axes of each input and of the out-

put: "s,i,sid,dc,db->sidcb" says "for every combination of the five variables, multiply $S[s]$, $I[i]$, $D[s,i,d]$, $C[d,c]$, and $B[d,b]$." That is the factorization formula, typed almost directly. The total is 1, as a probability distribution's must be.

To condition on evidence, probability zeros out every entry that disagrees with it, adds over all the other variables, and divides by the total that remains: exactly the "keep the matching rows" step from the hand calculations.

What you know	P(heavy snow)
nothing	0.100
school is closed	0.619
school is closed, and there is no ice	0.886
school is closed, and it is icy	0.174

Enumeration doubles in cost with every variable you add: 50 yes-or-no variables would need a table with $2^{50} \approx 10^{15}$ entries. Real systems use smarter exact methods, which add up one variable at a time, or approximate methods based on random sampling.

6.6 Learning a Network

Learning has two parts. *Parameter learning* estimates the CPT entries from data, often by counting: if school was closed on 14 of the 20 heavy-snow, no-ice days in the district's records, estimate $P(D \mid S, \neg I) = 0.7$. *Structure learning* chooses the arrows, and it is harder, because many structures fit the same data. Expert knowledge is often used to choose the structure.

Exercise 33 Compute a total probability

In the rain network, find $P(W)$ if $P(R)$ is changed to 0.5, with the same conditional probabilities.

Working Space

Answer on Page 65

Exercise 34 Read a table

A node C has two parents, A and B , each either true or false. How many

Working Space

rows does its CPT have?

Answer on Page 65

Exercise 35 Both messages

In the snow-day network, the robocall comes *and* your brother says there is no school. Find $P(D \mid C, B)$ by hand. (Given D , the call and your brother are independent.)

Working Space

Answer on Page 65

Exercise 36 Count the numbers

A network has 6 yes-or-no variables. One has no parents, two have one parent each, and three have two parents each. How many numbers do its tables need? How many would the full joint table need?

Working Space

Answer on Page 65

Exercise 37 Check the code

Use probability in `snow_day.py` to check your answer to the “Both messages” exercise. Then find $P(\text{ice} \mid \text{robocall})$ and $P(\text{ice} \mid \text{robocall}, \text{no heavy snow})$. Explain the difference.

Working Space

Answer on Page 65

Exercise 38 Learn by simulating

Generate 10,000 random mornings from the network. Draw snow and ice by comparing random numbers with their probabilities, then draw school closings using the probabilities in `p_closed`. Estimate $P(S \mid D)$ by counting. How close is it to 0.619?

Working Space

Answer on Page 65

Answers to Exercises

Answer to Exercise 1 (on page 9)

$$p(2) = \sigma(-2) = \frac{1}{1+e^2} \approx \frac{1}{8.389} \approx 0.119.$$

Answer to Exercise 2 (on page 9)

$p(x) = 0.5$ when the log odds are 0, so $-4 + x = 0$. Thus $x = 4$.

Answer to Exercise 3 (on page 9)

The odds are multiplied by $e^{0.5} \approx 1.649$, an increase of about 65%.

Answer to Exercise 4 (on page 9)

The odds are $e^{0.8} \approx 2.23$, meaning passing is about 2.23 times as likely as failing. The probability is $\frac{2.23}{1+2.23} \approx 0.69$.

Answer to Exercise 5 (on page 10)

The log odds are $-3 + 0.5(6) + 0 = 0$, so $P = 0.5$.

Answer to Exercise 6 (on page 10)

The rule says the probability rises by about $\frac{0.5}{4} = 0.125$, to about 0.625. Exactly, the log odds are 0.5 and $P = \frac{1}{1+e^{-0.5}} \approx 0.622$. The rule is very close because the student starts at $P = 0.5$.

Answer to Exercise 7 (on page 10)

$-\ln(1 - 0.95) = -\ln 0.05 \approx 3.00$. With 0.5, the loss is $-\ln 0.5 \approx 0.69$. Being confidently wrong costs more than four times as much as shrugging.

Answer to Exercise 8 (on page 11)

Keep only one column of features, for example

```
1 features = data["body_mass_g"][adelie][:, None]
```

and start from `np.zeros(2)` (in both places) and drop the bill depth from `x_new`. Mass alone gets 77.8% of the test penguins right, and bill depth alone gets 75.0%. Together they get 88.9%, so yes: each feature carries information the other does not.

Answer to Exercise 9 (on page 11)

An average bill depth has a standardized value of 0, so the boundary is where $0.54 + 3.55z = 0$, at $z \approx -0.152$. In grams, that is $3720 - 0.152 \times 474 \approx 3648$ g. Average-billed Adelies heavier than about 3650 g are predicted male.

Answer to Exercise 10 (on page 18)

Accuracy = $\frac{30+50}{100} = 0.80$. Precision = $\frac{30}{35} \approx 0.857$.

Answer to Exercise 11 (on page 19)

Recall (sensitivity). A false negative means a sick person is told they are healthy, which is the more serious error in screening.

Answer to Exercise 12 (on page 19)

Accuracy is 0.95, which sounds high. Recall is $\frac{0}{TP+FN} = 0$, because it finds none of the positive cases.

Answer to Exercise 13 (on page 19)

There are 4 pairs. 0.8 beats both negatives (2 pairs). 0.4 beats 0.3 but loses to 0.5 (1 pair). So 3 of 4 pairs are correct, and the AUC is $\frac{3}{4} = 0.75$.

Answer to Exercise 14 (on page 20)

Threshold 0.5: the scores 0.9, 0.7, and 0.6 are predicted positive, so $TP = 2$, $FP = 1$, $FN = 0$, $TN = 2$. Precision = $\frac{2}{3}$, recall = 1, and the false positive rate = $\frac{1}{3}$.

Threshold 0.8: only 0.9 is predicted positive, so $TP = 1$, $FP = 0$, $FN = 1$, $TN = 3$. Precision = 1, recall = $\frac{1}{2}$, and the false positive rate = 0.

Raising the threshold made the classifier more careful: precision went up and recall went down. The two thresholds give the ROC points $(\frac{1}{3}, 1)$ and $(0, \frac{1}{2})$, both on the staircase in the worked example.

Answer to Exercise 15 (on page 20)

Threshold 0.1: $\frac{46}{46+15+3} \approx 0.719$. Threshold 0.3: $\frac{36}{36+6+8} = 0.720$. Threshold 0.5: $\frac{28}{28+1+12} \approx 0.683$. Threshold 0.7: $\frac{18}{18+0+17} \approx 0.514$. Threshold 0.3 is best, just barely. The default of 0.5 is not the best choice for this detector.

Answer to Exercise 16 (on page 20)

The correct predictions are on the diagonal: $\frac{40+14+36}{100} = 0.90$. Treat Chinstrap as positive and the rest as negative. Precision: of the 17 predicted Chinstraps, 14 are right, so $\frac{14}{17} \approx 0.82$. Recall: of the 20 actual Chinstraps, 14 were found, so $\frac{14}{20} = 0.70$. Most mistakes confuse Chinstraps with Adelies.

Answer to Exercise 17 (on page 21)

Replace `data["bill_depth_mm"]` with `data["body_mass_g"]`. At threshold 0.5, $TP = 25$, $FP = 0$, $FN = 1$, and $TN = 74$, and the AUC rises to 0.999. Chinstraps have long bills like Gentoos but are much lighter, so long-billed, light penguins are almost always Chinstraps. With both features, a straight boundary separates them well.

Answer to Exercise 18 (on page 27)

The points (2, 1) and (4, 1) are tied for nearest, both at distance $\sqrt{1^2 + 0.5^2} \approx 1.118$. Both are labeled A, so the prediction is A either way.

Answer to Exercise 19 (on page 28)

The two A points at distance 1.118 come first. The third nearest is (2, 3), labeled B, at $\sqrt{1^2 + 1.5^2} \approx 1.803$. The vote is A, A, B, so the prediction is A.

Answer to Exercise 20 (on page 28)

Income values are in the tens of thousands, while heights are around 170. Differences in income are far larger, so they control the distance. Standardizing both features gives each the same influence.

Answer to Exercise 21 (on page 28)

With $k = 6$, every point votes, and the vote is always 3 to 3, a tie, no matter where the new point is. With k equal to the size of the training set, k -NN ignores the features entirely and predicts the most common class (or has to break a tie), which is the baseline from the Classification and Regression chapter.

Answer to Exercise 22 (on page 29)

Replace the voting line with

```
1 return y_train[nearest].mean(axis=1)
```

and use `y = data["body_mass_g"]` as the label. The test RMSEs are about 420 g ($k = 1$), 330 g ($k = 5$), 321 g ($k = 20$), and 340 g ($k = 50$). With k between 5 and 20, k -NN beats the straight-line model, because it can follow patterns that are not linear, such as the different species.

Answer to Exercise 23 (on page 35)

(a) A stemmer such as Porter's often maps both to "happi," so they match. A lemmatizer keeps "happiness" as its own noun and "happy" as an adjective, so they do not match.
 (b) A stemmer does not link "better" to "good," because the spelling is too different. A lemmatizer that knows irregular forms can map "better" to "good."

Answer to Exercise 24 (on page 36)

The Porter stemmer reduces "connected," "connection," and "connecting" all to "connect," so the first two documents match. "Linked" means the same thing but shares no letters, so stemming cannot find it. Matching synonyms takes a different technique, such as a thesaurus or word embeddings.

Answer to Exercise 25 (on page 36)

"dresses" → "dress" (the "ses" rule). "hopping" → "hopp" → "hop" (the "ing" rule, then the doubled letter is dropped). "filled" → "fill" (the "ed" rule; the doubled "l" is kept, because "l" is on the list of letters not to undouble). "sing" stays "sing": removing "ing" would leave only one letter. "quickly" → "quick" (the "ly" rule).

Answer to Exercise 26 (on page 36)

Add ("ion", ""). It can go anywhere before ("s", ""), since no word ends in both; putting it first is simplest. "connection" becomes "connect" and "selection" becomes "select." None of the other example words end in "ion," so their stems do not change. (It does hurt some other words: "champion" becomes "champ" and "million" becomes "mill." The 3-letter check saves "lion" and "onion." Every simple rule has exceptions.)

Answer to Exercise 27 (on page 37)

```

1  LEMMAS = {"ran": "run", "better": "good", "mice": "mouse", "was": "be"}
2
3  def lemmatize(word):
4      return LEMMAS.get(word, word)

```

dict.get returns the second argument when the key is missing. A real lemmatizer needs

an entry for every form of every word in the language (plus rules for using context to tell “saw” the verb from “saw” the tool), and English has hundreds of thousands of words.

Answer to Exercise 28 (on page 47)

$0.3 \times 0.08 = 0.024$ and $0.7 \times 0.02 = 0.014$. So $P(A | \mathbf{x}) = \frac{0.024}{0.024+0.014} \approx 0.632$. The evidence favors A by 4 to 1, enough to overcome the prior that favors B.

Answer to Exercise 29 (on page 47)

“Free” never appears in the ham messages, so $P(\text{free} | \text{ham}) = 0$, the ham product is 0, and the message is spam with probability 1. A single word would overrule everything else in the message, and the model could never change its mind about a message containing “free,” however many friendly words it had.

Answer to Exercise 30 (on page 48)

Spam: $\frac{22}{60} \cdot \frac{2}{486} \cdot \frac{2}{486} \approx 6.21 \times 10^{-6}$. Ham: $\frac{38}{60} \cdot \frac{2}{526} \cdot \frac{3}{526} \approx 1.37 \times 10^{-5}$. So $P(\text{spam}) \approx \frac{6.21}{6.21+13.7} \approx 0.31$. The filter calls it ham.

Answer to Exercise 31 (on page 48)

The accuracies are 57, 57, 58, 58, and 58 out of 60. Some smoothing is essential, but the exact amount barely matters on this data. Like k in k -nearest neighbors, α is a setting you could choose by cross-validation.

Answer to Exercise 32 (on page 48)

Naive Bayes improves to 97.0% (cross-validation) and 94.0% (test); the full model to 98.5% and 95.5%. Both now say the mystery penguin is most likely a Chinstrap: 67% for naive Bayes and 96% for the full model. The flipper length rules out Gentoo, whose flippers are much longer.

Answer to Exercise 33 (on page 56)

$$P(W) = 0.9(0.5) + 0.1(0.5) = 0.5.$$

Answer to Exercise 34 (on page 56)

There are $2 \times 2 = 4$ parent combinations, so the CPT has 4 rows, each giving $P(C \mid A, B)$.

Answer to Exercise 35 (on page 57)

$$P(D \mid C, B) = \frac{0.98 \times 0.9 \times 0.11628}{0.98 \times 0.9 \times 0.11628 + 0.02 \times 0.3 \times 0.88372} = \frac{0.10256}{0.10256 + 0.00530} \approx 0.951$$

Two pieces of evidence together are stronger than either alone.

Answer to Exercise 36 (on page 57)

A node needs one number for each combination of its parents' values: $1 + 2(2) + 3(4) = 17$. The full joint table has $2^6 = 64$ entries, so it needs 63 numbers (the last one is 1 minus the others).

Answer to Exercise 37 (on page 58)

```
1 print(probability('D', {'C': True, 'B': True})) # 0.951
2 print(probability('I', {'C': True}))           # 0.330
3 print(probability('I', {'C': True, 'S': False})) # 0.595
```

Without snow, something else has to explain the call, and the most likely explanation is ice: explaining away in reverse.

Answer to Exercise 38 (on page 58)

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

```
2 n = 10_000
3 s = rng.random(n) < p_snow
4 i = rng.random(n) < p_ice
5 d = rng.random(n) < p_closed[s.astype(int), i.astype(int)]
6 print(np.mean(s[d]))
```

With this seed, 1,173 of the mornings are closings, and the fraction of them with snow is 0.618. (Different seeds give answers that typically differ from 0.619 by about 0.014.) This is the idea behind approximate inference by sampling.



INDEX

- accuracy, 13
- area under the ROC curve, 16
- AUC, 16
- bag of words, 34
- Bayes' rule, 39
- Bayesian classifier, 39
- Bayesian network, 51
 - inference, 51
- conditional independence, 40
- conditional probability table, 51
- confusion matrix, 13
- covariance matrix, 44
- cross-entropy, 5
- curse of dimensionality, 27
- decision boundary, 8
- directed acyclic graph, 51
- einsum, 55
- Euclidean distance, 23
- explaining away, 52
- Gaussian naive Bayes, 43, 44
- inference by enumeration, 54
- instance-based, 23
- k-nearest neighbor, 23
- Laplace smoothing, 40
- leave-one-out, 43
- leave-one-out cross-validation, 43
- lemma, 32
- Lemmatization, 32
- lemmatization, 31
 - applications of, 35
- linear classifier, 8
- log loss, 5
- log odds, 4
- Logistic regression, 3
- logistic regression, 3
 - multiple, 5
- logit, 4
- maximum likelihood, 5, 6
- multivariate Gaussian distribution, 44
- multivariate normal distribution, 44
- naive Bayes, 40
- odds, 4
- odds ratio, 4
- Parameter learning, 56
- precision, 13
- precision-recall curve, 16
- precision-recall curve, 16
- prior probability, 39
- recall, 13
- receiver operating characteristic, 16
- regular expression, 33
- ROC curve, 16

- sigmoid, [4](#)
- sigmoid function, [4](#)
- sparse, [35](#)
- sparse matrix, [35](#)
- stem, [32](#)
- Stemming, [32](#)
- stemming, [31](#)
 - applications of, [35](#)
 - in Python, [32](#)
- stop words, [31](#)
- Structure learning, [56](#)

- text normalization, [31](#)
- threshold, [8](#), [15](#)
- tokenizing, [31](#)
- tokens, [31](#)