



CONTENTS

1	Classification and Regression	3
1.1	Meet the Penguins	3
1.2	Supervised Learning	4
1.3	Regression	4
1.4	Classification	5
1.5	Measuring Errors	6
1.6	Baselines: The Dumbest Models	7
1.7	A Map of the Unit	8
2	The Training, Validation, and Testing Process	11
2.1	Memorizing Is Not Learning	11
2.2	Three Sets	13
2.3	The Rules	14
2.4	Cross-Validation	15
3	Preparing Data for Models	21
3.1	Why Scale Matters	21
3.2	Standardization	23
3.2.1	Which Methods Need It?	24
3.3	Categories Become Numbers	24
3.3.1	The Dummy Variable Trap	25
3.4	Missing Values	25
3.5	Putting It Together	25

4	Least Squares	29
4.1	Predictor and Response	29
4.2	The Best Line	29
4.3	Solving with Calculus	30
4.4	Reading the Line	31
4.5	Fitting Penguins with NumPy	32
4.6	The Matrix Form	33
4.7	Multiple Regression	34
4.8	Look at the Residuals	36
4.9	A Line That Points the Wrong Way	37
4.10	When Least Squares Is Not Enough	38
5	Linear Regression and Gradient Descent	41
5.1	Why Not Just Use the Formula?	41
5.2	The Cost Function and Its Gradient	41
5.3	The Update Rule	42
5.4	Letting the Computer Walk	42
5.5	Valleys and Learning Rates	43
5.6	Standardize First	44
5.7	Mini-Batches	46
A	Answers to Exercises	51
	Index	59

Classification and Regression

“How much will this sneaker resell for?” and “Is this sneaker fake?” sound similar, but they are different kinds of questions. The first wants a number; the second wants a category. Knowing which kind you are asking decides which tools you use and how you grade them.

This chapter opens a unit on *machine learning*: writing programs that learn how to make predictions from examples instead of following rules that a person typed in. Every chapter in the unit uses the same data, so you can compare the methods side by side.

1.1 Meet the Penguins

From 2007 to 2009, Dr. Kristen Gorman and the Palmer Station Long Term Ecological Research program measured penguins on three islands near the Antarctic Peninsula. The data, released for anyone to use as the Palmer Penguins dataset, describes 344 penguins of three species: Adelie, Chinstrap, and Gentoo. Eleven are missing a measurement, so we will use the other 333.

Column	Kind	Example
species	category	Adelie, Chinstrap, or Gentoo
island	category	Biscoe, Dream, or Torgersen
bill_length_mm	number	39.1
bill_depth_mm	number	18.7
flipper_length_mm	number	181
body_mass_g	number	3750
sex	category	male or female
year	number	2007

Each row is one penguin. The data comes in two files that you will use in every chapter of this unit: `penguins.csv`, the data itself, and `penguins.py`, which contains one function, `load_penguins`. It reads the file, skips the penguins with a missing measurement, and returns a dictionary of NumPy arrays, one per column. So `data["body_mass_g"]` is an array of 333 masses.

1.2 Supervised Learning

In *supervised learning*, you are given labeled examples: for each one, some inputs, called *features*, and the correct answer, called the *label*. The goal is a *model*, a function that takes the features of a new example and predicts its label.

We usually stack the features into a matrix X , with one row per example and one column per feature, and the labels into a vector $\mathbf{y}$:

$$X = \begin{bmatrix} 181 & 39.1 \\ 186 & 39.5 \\ 195 & 40.3 \\ \vdots & \vdots \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 3750 \\ 3800 \\ 3250 \\ \vdots \end{bmatrix}$$

Here the features are flipper length and bill length, and the label is body mass. With n examples and p features, X is $n \times p$ and $\mathbf{y}$ has n entries.

Everything in this unit is supervised learning except clustering, near the end, where there are no labels at all and the computer has to find the groups on its own. That is called *unsupervised learning*.

1.3 Regression

In *regression*, the label is a number. Can you predict a penguin's body mass from the length of its flippers? Figure 1.1 suggests you can, roughly: longer flippers go with heavier penguins.

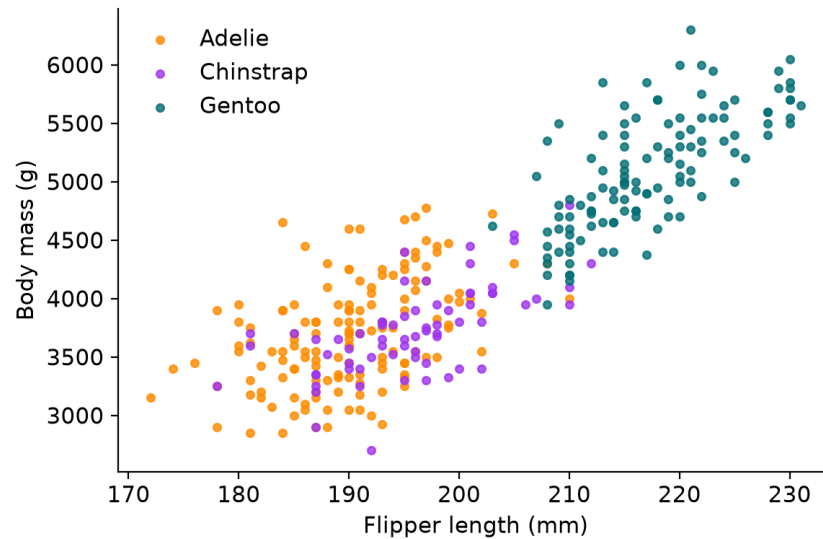


Figure 1.1: Body mass against flipper length for 333 penguins. A regression model predicts the mass (a number) from the flipper length.

Other regression problems: predicting the price of a used car from its age and mileage, tomorrow's high temperature from today's weather, or how many minutes a student will spend on homework from the number of assignments.

1.4 Classification

In *classification*, the label is a category. Can you tell a penguin's species from its bill? Figure 1.2 shows that the three species form three fairly separate clouds, so a model has a good chance.

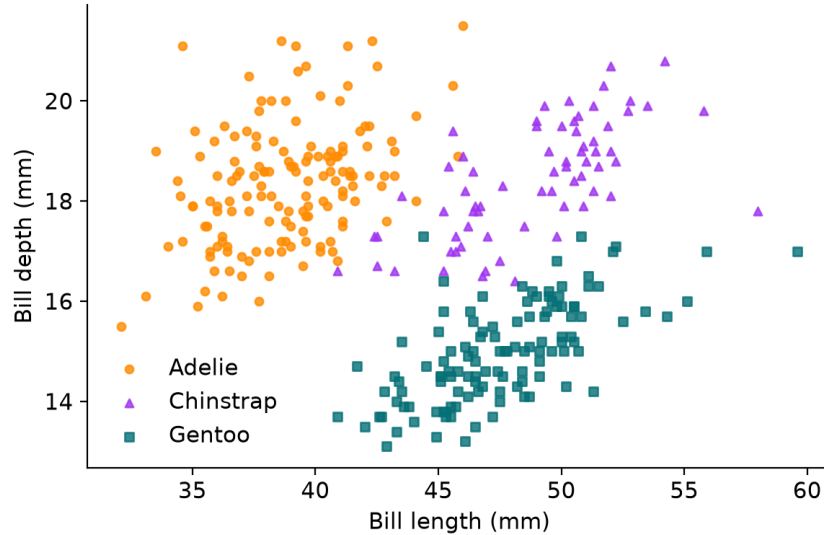


Figure 1.2: Bill depth against bill length, colored by species. A classification model predicts the species (a category) from the two measurements.

With two categories, the problem is *binary*: spam or not spam, pass or fail, male or female penguin. With more than two, it is *multi-class*: one of three penguin species, or which of 26 letters a handwritten mark is.

The same features can support either kind of problem. With each student's hours of study, "What score will this student get?" is regression, and "Will this student pass?" is classification. What decides it is the kind of label you predict.

1.5 Measuring Errors

To compare models, we need to grade them.

For regression, look at the *errors*, also called *residuals*: the true value minus the prediction, $y_i - \hat{y}_i$. (The hat means "predicted.") Two common summaries:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (\text{root mean squared error})$$

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (\text{mean absolute error})$$

Both are in the same units as the label. RMSE punishes big misses more, because it squares them first.

Example: A model predicts masses of 3550, 3900, and 3650 g for three penguins that actually weigh 3750, 3800, and 3250 g. Find the RMSE and MAE.

Solution: The errors are $3750 - 3550 = 200$, $3800 - 3900 = -100$, and $3250 - 3650 = -400$.

$$\text{RMSE} = \sqrt{\frac{200^2 + 100^2 + 400^2}{3}} = \sqrt{70,000} \approx 265 \text{ g} \quad \text{MAE} = \frac{200 + 100 + 400}{3} \approx 233 \text{ g}$$

The 400 g miss pulls RMSE above MAE.

For classification, the simplest grade is *accuracy*, the fraction of predictions that are correct. Accuracy hides a lot, though. A medical test that always says “healthy” is 99% accurate if 99% of people are healthy, and completely useless. The Evaluating Classification Systems chapter gives better measures.

1.6 Baselines: The Dumbest Models

Before building a clever model, always ask: what would a model with no brains at all score? For regression, the dumbest sensible model ignores the features and always predicts the average label. For classification, it always predicts the most common class. These are called *baselines*. A model that cannot beat its baseline has learned nothing.

Put `penguins.csv` and `penguins.py` in a folder, and create `baselines.py` next to them:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 mass = data["body_mass_g"]
6 species = data["species"]
7 print("Penguins:", len(mass))
8
9 # Regression baseline: always predict the average mass
10 guess = np.mean(mass)
11 errors = mass - guess
12 rmse = np.sqrt(np.mean(errors**2))
13 print(f"Always guess {guess:.0f} g: RMSE = {rmse:.0f} g")
14
15 # Classification baseline: always predict the most common species
16 names, counts = np.unique(species, return_counts=True)
17 most_common = names[np.argmax(counts)]
18 accuracy = np.mean(species == most_common)
19 print(f"Always guess {most_common}: accuracy = {accuracy:.1%}")

```

Penguins: 333

Always guess 4207 g: RMSE = 804 g

Always guess Adelie: accuracy = 43.8%

`np.unique` with `return_counts=True` returns each distinct value and how often it appears, and `species == most_common` makes an array of True and False, whose mean is the fraction that are True.

So any mass model needs an RMSE well under 804 g, and any species model needs to beat 43.8%. (Spoiler: a straight line through Figure 1.1 gets the RMSE down to about 392 g, and several methods in this unit identify the species correctly more than 95% of the time.)

1.7 A Map of the Unit

Method	Task	Penguin question
Least squares (linear regression)	regression	mass from flipper length, and more
Gradient descent	both	the same line, found a new way
Logistic regression	binary classification	male or female
k-nearest neighbors	classification	species
Naive Bayes	classification	species, and spam filtering
Decision trees and random forests	both	species
Boosting	both	sex, and mass
Generalized linear models	counts and more	counts
k-means clustering	unsupervised	groups with no labels
Neural networks	both	mass and species

Exercise 1 Name the task

For each task, say whether it is classification or regression, and whether it is binary or multi-class: (a) predicting tomorrow's temperature; (b) sorting a photo into cat, dog, or bird; (c) deciding whether a loan will default.

Working Space

Answer on Page 51

Exercise 2 Choose a measure

A model predicts house prices. Which error measure is appropriate, and why is accuracy not?

Working Space

Answer on Page 51

Exercise 3 Same data, two questions

A school has each student's hours of study and final exam score. Describe one regression question and one classification question you could ask with this data.

Working Space

Answer on Page 51

Exercise 4 RMSE by hand

A model predicts 10, 12, and 20 for three examples whose true labels are 11, 12, and 17. Find the RMSE and the MAE.

Working Space

Answer on Page 51

Exercise 5 A smarter baseline

Instead of predicting the overall average mass for every penguin, predict the average mass of its own species. Add this to `baselines.py`. What is the RMSE now? Why is this not a fair comparison with the first baseline?

Working Space

Answer on Page 51

Exercise 6 Baseline for sex

What accuracy does the majority-class baseline get for predicting a penguin's sex? What does that tell you about any sex classifier that scores 52%?

Working Space

Answer on Page 52

The Training, Validation, and Testing Process

Suppose a friend gets a copy of the answer key before a math test and memorizes it: question 1 is 42, question 2 is -3 , and so on. They score 100%. Do they know any math? You cannot tell from that test. You would have to give them new questions.

Machine learning has exactly this problem. A flexible model can memorize the examples it learned from, so its score on those examples says little about how it will do on new ones. This chapter shows how to grade a model honestly.

2.1 Memorizing Is Not Learning

Imagine you had measured only 12 penguins, and you want to predict body mass from flipper length. You could fit a straight line through the 12 points. Or a curve: a polynomial of degree 3, or degree 8. A higher degree means a more flexible curve, which can pass closer to every point. (The `np.polyfit` function finds the best polynomial of a given degree; the Least Squares chapter shows how.)

To judge the curves fairly, we also look at 50 *other* penguins that the curves never saw. Figure 2.1 shows what happens.

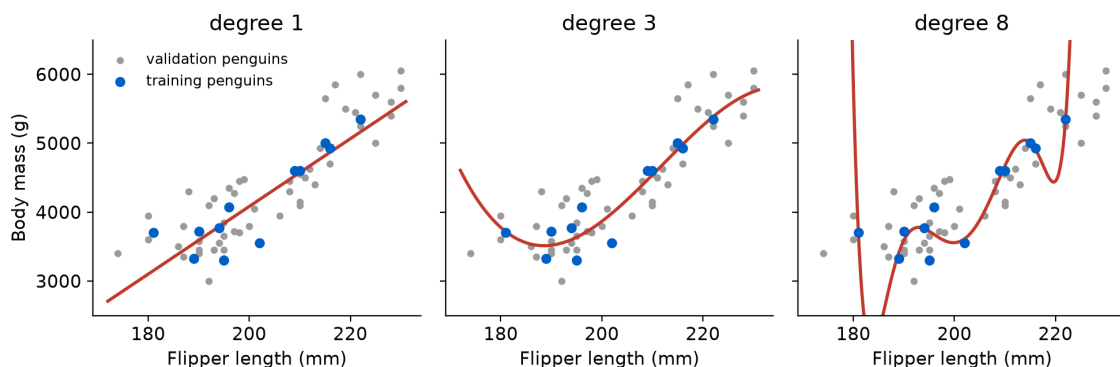


Figure 2.1: Curves fit to 12 training penguins (blue), judged on 50 validation penguins (gray). The degree-8 curve passes near every training penguin and is wildly wrong almost everywhere else.

The degree-8 curve wiggles through the training penguins, so its error on them is small. Between and beyond them, it shoots off to absurd masses. It has memorized the 12 penguins instead of learning the pattern. This is called *overfitting*. The opposite problem, a model too simple to capture the pattern, is called *underfitting*.

Here is the experiment in code. Put `penguins.csv` and `penguins.py` in a folder and create `overfit.py`:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 x = data["flipper_length_mm"]
6 y = data["body_mass_g"]
7
8 rng = np.random.default_rng(seed=4)
9 order = rng.permutation(len(x))
10 train, val = order[:12], order[12:62]    # only 12 penguins to learn from
11
12 # Rescale x so that high powers of it stay reasonable
13 mean, sd = x[train].mean(), x[train].std()
14 z = (x - mean) / sd
15
16 def rmse(actual, predicted):
17     return np.sqrt(np.mean((actual - predicted)**2))
18
19 print("degree  train RMSE  val RMSE")
20 for degree in range(1, 9):
21     coefficients = np.polyfit(z[train], y[train], degree)
22     train_error = rmse(y[train], np.polyval(coefficients, z[train]))
23     val_error = rmse(y[val], np.polyval(coefficients, z[val]))
24     print(f"{degree:6d} {train_error:11.0f} {val_error:9.0f}")

```

degree	train RMSE	val RMSE
1	312	414
2	239	410
3	224	386
4	220	429
5	219	564
6	179	6439
7	179	6562
8	179	20234

`rng.permutation(n)` shuffles the numbers 0 to $n - 1$, so `order[:12]` picks 12 random penguins. Fixing the seed makes the shuffle the same every time you run the program.

The two columns tell opposite stories (Figure 2.2). The training error falls every time the curve gets more flexible: more wiggle room always fits the training data better. The

validation error falls a little, then climbs, and finally explodes. **Only the validation error tells you how the model will do on penguins it has not seen.**

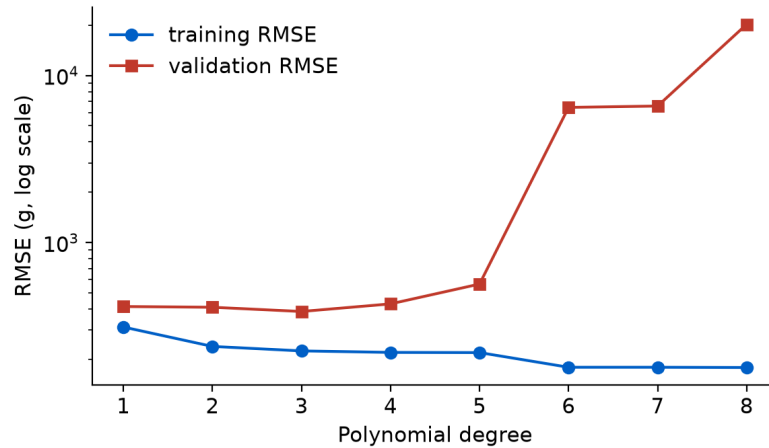


Figure 2.2: Training error keeps falling as the degree rises. Validation error has a minimum, then rises sharply. The gap between them is the signature of overfitting.

2.2 Three Sets

To grade honestly, split the data into three sets before you do anything else. Think of a math class: homework is for learning (training), practice tests tell you whether your study strategy is working so you can adjust it (validation), and the final exam measures what you really know, and it only counts if you have never seen its questions (testing).

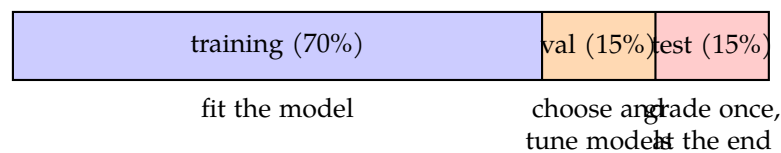


Figure 2.3: A common way to split the data. Shuffle first.

- The *training set* is used to fit the model: to find the coefficients of a curve, or the splits in a decision tree.
- The *validation set* is used to compare models and to choose *hyperparameters*: settings that the training does not choose for you, such as the degree of a polynomial or the number of neighbors k in the k -nearest-neighbors method.
- The *test set* is used once, at the very end, to estimate how the final model will do on new data.

Example: A data set has 1,000 rows. How many go in each set with a 70/15/15 split?

Solution: 700 training rows, 150 validation rows, and 150 test rows.

Shuffle the rows first, so each set looks like the whole data set. This matters most when the file is sorted: `penguins.csv` lists all the Adelies first, then the Gentoos, then the Chinstraps, so without shuffling, the test set would be 50 Chinstraps and nothing else. Here is a shuffled split, `split.py`:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 n = len(data["species"])
6
7 rng = np.random.default_rng(seed=1)      # a fixed seed makes the shuffle repeatable
8 order = rng.permutation(n)               # the numbers 0 to n-1, shuffled
9
10 n_train = int(0.70 * n)
11 n_val = int(0.15 * n)
12 train = order[:n_train]
13 val = order[n_train:n_train + n_val]
14 test = order[n_train + n_val:]
15 print("Sizes:", len(train), len(val), len(test))
16
17 for name, rows in [("train", train), ("val", val), ("test", test)]:
18     kinds, counts = np.unique(data["species"][rows], return_counts=True)
19     shares = ", ".join(f"{k} {c / len(rows):.0%}" for k, c in zip(kinds, counts))
20     print(f"{name:5s} {shares}")

```

Sizes: 233 49 51

train Adelie 45%, Chinstrap 18%, Gentoo 37%

val Adelie 39%, Chinstrap 29%, Gentoo 33%

test Adelie 45%, Chinstrap 24%, Gentoo 31%

`train`, `val`, and `test` are arrays of row numbers. Indexing an array with an array of row numbers, as in `data["species"][rows]`, picks out just those rows.

Even after shuffling, the small sets wander: Chinstraps are 18% of the training set but 29% of the validation set. For classification, it is often better to shuffle and split each class separately, so every set keeps the same mix. That is called a *stratified split* (an exercise asks you to write one).

2.3 The Rules

The three sets only work if they stay separate.

- **Never fit anything on the validation or test set.** That includes the coefficients, and also things like the mean and standard deviation you use to rescale the data. In `overfit.py`, the rescaling uses `x[train]` only.
- **Never choose a model because of its test score.** If you try ten models and keep the one with the best test score, the test set has become a second validation set, and its score is too optimistic.
- **Watch for leaks.** *Data leakage* is any way information from the validation or test set sneaks into training.

Leaks are easy to create by accident:

Leak	Why it is a problem
Rescaling with the mean of all the data	The test rows helped shape the training data.
The same penguin appears twice, once in training and once in testing	The model is graded on a penguin it memorized.
Predicting next week's sales with a random split of the dates	The model learns from the future to "predict" the past. For data over time, train on earlier dates and test on later ones.
A feature that is really the answer, such as "was flagged as spam" in a spam filter	The model looks perfect and fails on real data, where that feature does not exist yet.

2.4 Cross-Validation

With only a few hundred penguins, a 50-penguin validation set is small, and its score depends a lot on which penguins happened to land in it. *k-fold cross-validation* uses the training data more thoroughly. Split the training rows into k equal groups, called *folds*. Train k times, each time holding out a different fold for validation, and average the k validation scores (Figure 2.4). The test set stays locked away the whole time.

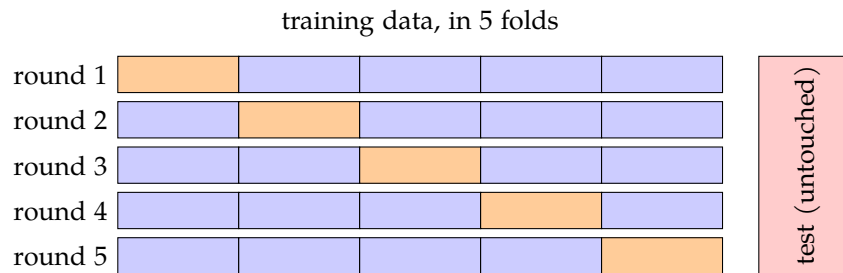


Figure 2.4: 5-fold cross-validation. In each round, the orange fold is the validation set and the blue folds are for training. Every training row is used for validation exactly once.

Let us use 5-fold cross-validation, with all the penguins this time, to choose the polynomial degree. Create `crossval.py`:

```

1  import numpy as np
2  from penguins import load_penguins
3
4  data = load_penguins()
5  x = data["flipper_length_mm"]
6  y = data["body_mass_g"]
7
8  rng = np.random.default_rng(seed=1)
9  order = rng.permutation(len(x))
10 n_test = 50
11 test, train = order[:n_test], order[n_test:] # lock the test set away
12
13 mean, sd = x[train].mean(), x[train].std()
14 z = (x - mean) / sd
15
16 def rmse(actual, predicted):
17     return np.sqrt(np.mean((actual - predicted)**2))
18
19 k = 5
20 folds = np.array_split(train, k) # 5 groups of training rows
21
22 print("degree cross-validated RMSE")
23 scores = {}
24 for degree in range(1, 6):
25     errors = []
26     for i in range(k):
27         held_out = folds[i]
28         fit_rows = np.concatenate([folds[j] for j in range(k) if j != i])
29         coefficients = np.polyfit(z[fit_rows], y[fit_rows], degree)
30         errors.append(rmse(y[held_out], np.polyval(coefficients, z[held_out])))
31     scores[degree] = np.mean(errors)
32     print(f"{degree:6d} {scores[degree]:12.1f}")
33
34 best = min(scores, key=scores.get)
35 coefficients = np.polyfit(z[train], y[train], best)

```

```

36 print(f"Chose degree {best}. Test RMSE: {rmse(y[test], np.polyval(coefficients,
    ↪ z[test])):.1f}")

```

degree	cross-validated RMSE
1	389.6
2	376.1
3	371.1
4	370.6
5	371.5

Chose degree 4. Test RMSE: 407.2

`np.array_split` cuts an array into nearly equal pieces, and `min(scores, key=scores.get)` finds the dictionary key with the smallest value.

With 283 penguins to learn from instead of 12, overfitting is much milder: degrees 2 through 5 all score within about 6 g of each other. The program picks degree 4, but its lead over degree 3 is half a gram, far less than the variation between folds (about 16 g). When scores are this close, most data scientists choose the *simplest* model that is about as good as the best one, since simpler models are easier to understand and less likely to have overfit. One exercise makes this precise.

Notice also that the test RMSE (407 g) is higher than the cross-validated estimate (371 g). That is normal: choosing the best of several models makes its validation score a little optimistic, which is exactly why the untouched test set exists.

Exercise 7 Count the rows

A data set has 2,400 rows, split 70% training, 15% validation, and 15% testing. How many rows are in each set?

Working Space

Answer on Page 52

Exercise 8 Spot the leak

A student rescales every feature using the mean and standard deviation of all 1,000 rows, and then splits the data. Ex-

Working Space

plain why this is a problem.

Answer on Page 52

Exercise 9 Which set?

A researcher tries 10 values of k for a k -nearest-neighbors model and picks the one with the best test score. What is wrong with this?

Working Space

Answer on Page 52

Exercise 10 Read the curve

In Figure 2.2, which degrees underfit, and which overfit? Which degree would you choose using the validation set?

Working Space

Answer on Page 52

Exercise 11 A stratified split

Change `split.py` so that it shuffles and splits each species separately, putting 70% of each species in training, 15% in validation, and the rest in testing. Check that the three sets now have nearly the same species mix.

Working Space

Answer on Page 52

Exercise 12 The one-standard-error rule

In `crossval.py`, also compute the *standard error* of each degree's five fold scores, which is their standard deviation divided by $\sqrt{5}$. The *one-standard-error rule* says: find the best score, add its standard error, and choose the simplest model whose score is below that line. Which degree does it choose?

Working Space

Answer on Page 53

Preparing Data for Models

Here is a mystery penguin: bill 40.3 mm long and 18.5 mm deep, flipper 196 mm, mass 4,350 g. Which penguin in the data set is most like it? A natural idea is to treat each penguin as a point with four coordinates and find the closest point, the way you would find the distance between two points on a graph:

$$d = \sqrt{(\Delta \text{bill length})^2 + (\Delta \text{bill depth})^2 + (\Delta \text{flipper})^2 + (\Delta \text{mass})^2}$$

Try it, and the answer is a Gentoo with exactly the same mass, even though its bill is 5 mm shallower and its flipper 12 mm longer. Meanwhile, an Adelie that matches the mystery penguin almost perfectly (41.5, 18.3, 195, 4,300) is “farther away,” because its mass differs by 50. Fifty *grams* is nothing for a penguin, but the formula cannot tell grams from millimeters; it just sees the number 50.

Real data needs preparing before most models can use it. This chapter covers the two most common jobs: putting numbers on comparable scales, and turning categories into numbers.

3.1 Why Scale Matters

Each feature adds its squared difference to the distance. Figure 3.1 shows how much of the squared distance between the mystery penguin and that near-twin Adelie comes from each measurement.

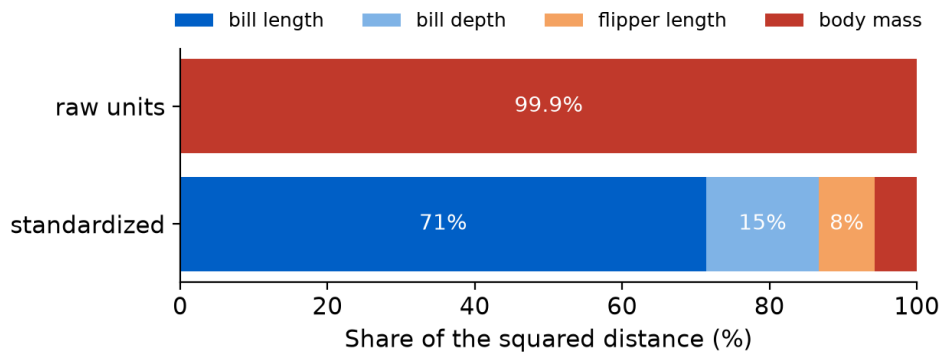


Figure 3.1: Where the distance comes from. In raw units, body mass accounts for 99.9% of it, because masses are measured in thousands. After standardizing, all four measurements contribute.

In raw units, the bills and flippers might as well not exist. This matters for any method that combines features by adding them up: nearest neighbors, clustering, gradient descent, and neural networks.

Here is the experiment, in `nearest.py`:

```

1  import numpy as np
2  from penguins import load_penguins
3
4  data = load_penguins()
5  names = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]
6  X = np.column_stack([data[name] for name in names])  # 333 rows, 4 columns
7  species = data["species"]
8
9  def nearest(features, i):
10     """Row number of the penguin closest to penguin i."""
11     distances = np.sqrt(np.sum((features - features[i])**2, axis=1))
12     distances[i] = np.inf  # do not count the penguin itself
13     return np.argmin(distances)
14
15  mystery = 91
16  print("Mystery penguin:", species[mystery], X[mystery])
17
18  j = nearest(X, mystery)
19  print("Nearest in raw units:", species[j], X[j])
20
21  Z = (X - X.mean(axis=0)) / X.std(axis=0)  # standardize every column
22  j = nearest(Z, mystery)
23  print("Nearest after standardizing:", species[j], X[j])

```

Mystery penguin: Adelia [40.3 18.5 196. 4350.]

```
Nearest in raw units: Gentoo [ 44.    13.6  208.  4350. ]
Nearest after standardizing: Adelie [ 41.5   18.3  195.  4300. ]
```

`np.column_stack` puts the four arrays side by side as columns. In `features - features[i]`, NumPy subtracts one penguin's row from every row at once (this is called *broadcasting*), and `axis=1` sums across each row. The line that makes `Z` is the subject of the next section.

This is not a lucky example. If you label every penguin with the species of its nearest neighbor, you are right 86.8% of the time in raw units and 98.2% of the time after standardizing.

3.2 Standardization

Standardization rescales a feature so its mean is 0 and its standard deviation is 1. For a feature with mean μ and standard deviation σ ,

$$z = \frac{x - \mu}{\sigma}$$

Each standardized value says how many standard deviations the original value is above or below the mean. It is the same *z*-score as in the Shape, Position, and Outliers chapter.

Example: Standardize the values 2, 4, 6, 8.

Solution: The mean is $\mu = 5$. Using the population standard deviation (dividing by n , as NumPy's `std` does), $\sigma = \sqrt{\frac{9+1+1+9}{4}} = \sqrt{5} \approx 2.236$. So

$$\frac{2-5}{2.236} \approx -1.342, \quad \frac{4-5}{2.236} \approx -0.447, \quad \frac{6-5}{2.236} \approx 0.447, \quad \frac{8-5}{2.236} \approx 1.342$$

Standardization changes the axis, not the shape (Figure 3.2). Skewed data stays skewed, and outliers stay outliers.

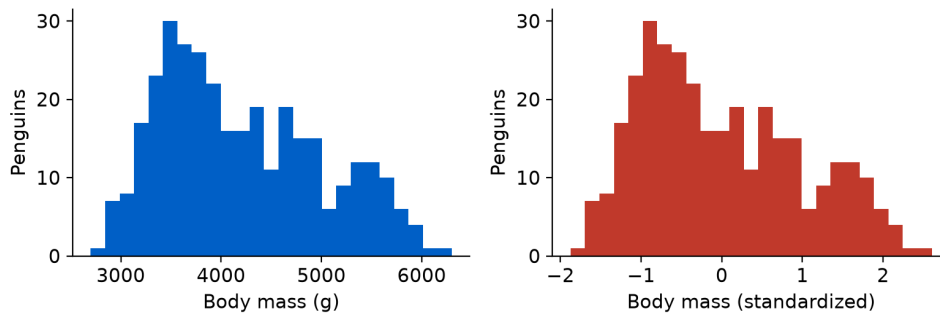


Figure 3.2: Penguin body masses before and after standardizing. The histograms are identical except for the numbers on the horizontal axis.

Another option, *min-max scaling*, squeezes each feature into the range 0 to 1:

$$x' = \frac{x - \min}{\max - \min}$$

It is common for things that are naturally bounded, like pixel brightness. Standardization is the usual default, because a single extreme value can squash all the others into a tiny corner of $[0, 1]$.

3.2.1 Which Methods Need It?

Method	Needs scaling?	Why
Least squares regression	no	gives the same predictions either way; scaling only makes coefficients easier to compare
Gradient descent	yes	unscaled features make it crawl
k-nearest neighbors, k-means	yes	they are built on distances
Naive Bayes	no	it treats each feature separately
Decision trees and forests	no	a split like “mass < 4000” works the same in any units
Neural networks	yes	training is slow and unstable without it

3.3 Categories Become Numbers

Species, island, and sex are *categorical variables*: their values are labels, not numbers. Most models need numbers. Why not just number the islands, Biscoe = 1, Dream = 2, Torgersen

= 3? Because then a model would believe that Torgersen is “bigger” than Biscoe, and that Dream sits exactly halfway between them. The islands have no such order.

One-hot encoding gives each category its own column, which is 1 when the row has that category and 0 otherwise:

Island	is_Biscoe	is_Dream	is_Torgersen
Torgersen	0	0	1
Biscoe	1	0	0
Dream	0	1	0
Biscoe	1	0	0

Every row has exactly one 1. A feature with two categories, such as sex, needs only a single 0-or-1 column.

3.3.1 The Dummy Variable Trap

One of those three columns is redundant. If you know `is_Dream` and `is_Torgersen`, you know `is_Biscoe`: it is $1 - \text{is_Dream} - \text{is_Torgersen}$. In a linear regression, which already has an intercept, the three columns add up to the column of 1s that the intercept multiplies, so the regression cannot tell their effects apart. The fix is to drop one column. The dropped category becomes the *baseline*, and each remaining coefficient measures the difference from it. Trees and neural networks do not mind the extra column.

For categories that *do* have an order, like small, medium, and large, or freshman through senior, a single column of numbers (1, 2, 3, 4) that respects the order is often better.

3.4 Missing Values

Eleven of the 344 penguins are missing a measurement, so `load_penguins` quietly drops them. With so few, that is fine. With more, you would lose too much data, and the usual fix is to *fill in*, or *impute*, each missing value with the training set’s mean (for numbers) or most common value (for categories). Sometimes you also add a 0-or-1 column recording that the value was missing, since being missing can itself carry information.

3.5 Putting It Together

Here is a full feature matrix for the penguins: four standardized measurements, two island columns, and a sex column. Note that the mean and standard deviation come from the training rows only, so no information leaks from the test set. Create `prepare.py`:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 n = len(data["species"])
6 order = np.random.default_rng(seed=1).permutation(n)
7 train, test = order[:250], order[250:]
8
9 # 1. Numbers: standardize with the TRAINING mean and standard deviation
10 names = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]
11 numbers = np.column_stack([data[name] for name in names])
12 mean = numbers[train].mean(axis=0)
13 sd = numbers[train].std(axis=0)
14 scaled = (numbers - mean) / sd
15
16 # 2. Categories: one-hot encode the island, dropping one column (Biscoe)
17 islands = np.array(["Dream", "Torgersen"])
18 island_columns = (data["island"][:, None] == islands[None, :]).astype(float)
19
20 # 3. Two categories need only one column
21 male = (data["sex"] == "male").astype(float)[:, None]
22
23 X = np.hstack([scaled, island_columns, male])
24 print("Feature matrix:", X.shape)
25 print("First penguin: ", np.round(X[0], 2))
26 # Adding 0.0 turns a printed -0.0 into 0.0
27 print("Training means:", np.round(X[train, :4].mean(axis=0), 2) + 0.0)
28 print("Test means:    ", np.round(X[test, :4].mean(axis=0), 2) + 0.0)

```

```

Feature matrix: (333, 7)
First penguin: [-0.9  0.76 -1.46 -0.62  0.    1.    1. ]
Training means: [0. 0. 0. 0.]
Test means:    [-0.02  0.   -0.11 -0.24]

```

The one-hot line uses broadcasting again. `data["island"][:, None]` is a column of 333 islands, and `islands[None, :]` is a row of two; comparing them gives a 333×2 table of True and False, and `astype(float)` turns those into 1s and 0s.

The first penguin, from Torgersen, gets (0,1) in the island columns, and its last 1 says it is male. The training means are exactly 0, as they must be. The test means are close to 0 but not equal to it, and that is correct: the test penguins were scaled with the training statistics, as if they had arrived later.

Exercise 13 Standardize a value

A feature has mean 50 and standard deviation 10. What is the standardized

Working Space

value of 65?

Answer on Page 53

Exercise 14 Which test went better?

You scored 85 on a math test (class mean 75, standard deviation 5) and 90 on a science test (class mean 85, standard deviation 10). Relative to your class, which test went better?

Working Space

Answer on Page 53

Exercise 15 Leakage

A student computes μ and σ from all the data, then splits it into training and test sets. What is the problem?

Working Space

Answer on Page 54

Exercise 16 Encode a variable

Write the one-hot encoding of the pets cat, dog, fish, cat, using three columns in that order.

Working Space

Answer on Page 54

Exercise 17 **Dropping a column**

A regression uses the island variable encoded as three columns. Why should one column be dropped?

Working Space

Answer on Page 54

Exercise 18 **Min-max scaling**

Add min-max scaling to nearest.py and find the mystery penguin's nearest neighbor with it. What values does penguin 0 (bill 39.1 mm long and 18.7 mm deep, flipper 181 mm, mass 3,750 g) get after min-max scaling?

Working Space

Answer on Page 54

Exercise 19 **One-hot species**

Write one line of NumPy that one-hot encodes the species into three columns, and check that every row adds up to 1.

Working Space

Answer on Page 54

Least Squares

Weighing a penguin is a wrestling match: you have to catch it, bag it, and hold the scale steady on the ice while it complains. Measuring its flipper is easier, and you can even do it from a photo. If flipper length predicted body mass well, researchers could skip a lot of wrestling.

The tool for that prediction is a straight line, chosen by a rule called *least squares*. It is the oldest method in this unit and still one of the most used. Almost every method that comes after it is built on the same idea: choose the model that makes the errors as small as possible.

4.1 Predictor and Response

Simple linear regression describes the relationship between two quantitative variables with a line. The *predictor* x (also called the explanatory or independent variable) is used to predict the *response* y (also called the outcome or dependent variable). In the language of the last two chapters, x is a feature and y is the label. It is *simple* because there is only one predictor; later in this chapter we use several.

The fitted line is

$$\hat{y} = b_0 + b_1x$$

where b_0 is the *intercept* and b_1 is the *slope*, the predicted change in y when x goes up by one. Statisticians often write the model for the whole population with an error term:

$$y = \beta_0 + \beta_1x + \epsilon$$

The error ϵ is everything about y that the line does not capture. Our b_0 and b_1 are estimates of β_0 and β_1 , computed from a sample.

4.2 The Best Line

Five students record how many hours they studied for a quiz and the score they earned (out of 10):

Hours studied, x	1	2	3	4	5
Quiz score, y	2.1	3.9	6.2	7.8	10.1

No line passes through all five points, so which line is best? For any line, each point has a *residual*, $e_i = y_i - \hat{y}_i$: how far the point sits above (positive) or below (negative) the line. We want the residuals small, all at once. The usual way to say that is to minimize the *sum of squared errors*:

$$S(b_0, b_1) = \sum_{i=1}^n (y_i - b_0 - b_1 x_i)^2$$

The *least squares line* is the one that makes S as small as possible.

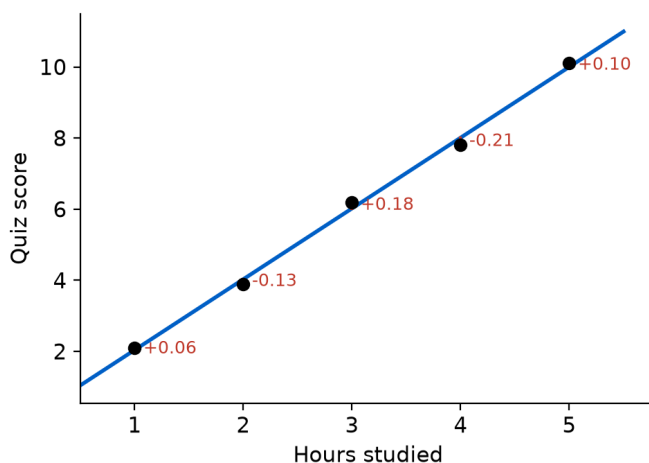


Figure 4.1: The least squares line for the quiz data. The residuals (red) are the vertical distances from the points to the line.

Why square? Squaring keeps positive and negative errors from cancelling. It punishes one big miss more than several small ones. And it makes the problem easy to solve with calculus.

4.3 Solving with Calculus

At the minimum of S , both partial derivatives are zero. Setting $\partial S / \partial b_0 = 0$ gives

$$\sum (y_i - b_0 - b_1 x_i) = 0 \implies b_0 = \bar{y} - b_1 \bar{x}$$

So the best line always passes through the point of averages, $(\bar{x}, \bar{y})$. Setting $\partial S / \partial b_1 = 0$ gives

$$\sum x_i (y_i - b_0 - b_1 x_i) = 0$$

Substituting for b_0 and simplifying:

$$b_1 = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} = \frac{S_{xy}}{S_{xx}}$$

The least squares line $\hat{y} = b_0 + b_1x$ has

$$b_1 = \frac{S_{xy}}{S_{xx}}, \quad b_0 = \bar{y} - b_1\bar{x}$$

where $S_{xx} = \sum (x_i - \bar{x})^2$ and $S_{xy} = \sum (x_i - \bar{x})(y_i - \bar{y})$.

Example: Find the least squares line for the quiz data.

Solution: The sums are

$$\sum x_i = 15, \quad \sum y_i = 30.1, \quad \sum x_i^2 = 55, \quad \sum x_i y_i = 110.2$$

so $\bar{x} = 3$ and $\bar{y} = 6.02$. A shortcut avoids subtracting the mean from every point: $S_{xx} = \sum x_i^2 - n\bar{x}^2 = 55 - 5(3)^2 = 10$ and $S_{xy} = \sum x_i y_i - n\bar{x}\bar{y} = 110.2 - 5(3)(6.02) = 19.9$. Therefore

$$b_1 = \frac{19.9}{10} = 1.99, \quad b_0 = 6.02 - 1.99(3) = 0.05$$

The line is $\hat{y} = 0.05 + 1.99x$. The residuals are 0.06, -0.13 , 0.18, -0.21 , 0.10, and the smallest possible sum of squared errors is

$$SS_{\text{res}} = 0.06^2 + 0.13^2 + 0.18^2 + 0.21^2 + 0.10^2 = 0.107$$

The residuals sum to zero. That is always true for a least squares line with an intercept; it is exactly what the equation $\partial S / \partial b_0 = 0$ says.

4.4 Reading the Line

The slope. $b_1 = 1.99$ means each extra hour of study is *associated* with about 2 more points. The word “associated” matters. A regression line describes a pattern in data. It does not show that studying *causes* higher scores, unless the data came from a well-designed experiment. Maybe students who study longer also sleep more, or already liked the subject.

The intercept. $b_0 = 0.05$ is the predicted score after zero hours of study. Nobody in the data studied zero hours, so this number is a guess about a situation the line never saw.

Extrapolation. For 6 hours, the line predicts $0.05 + 1.99(6) = 11.99$ points on a 10-point quiz. The line was fit to x between 1 and 5, and nothing guarantees the pattern continues outside that range. Predicting outside the range of the data is called *extrapolation*, and it is risky.

4.5 Fitting Penguins with NumPy

Now the penguins. Put `penguins.csv` and `penguins.py` in a folder and create `line.py`, which applies the formulas above to all 333 penguins:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 x = data["flipper_length_mm"]
6 y = data["body_mass_g"]
7
8 # The formulas from this chapter
9 x_bar, y_bar = np.mean(x), np.mean(y)
10 b1 = np.sum((x - x_bar) * (y - y_bar)) / np.sum((x - x_bar)**2)
11 b0 = y_bar - b1 * x_bar
12 print(f"mass = {b0:.1f} + {b1:.2f} * flipper")
13
14 # How good is it?
15 residuals = y - (b0 + b1 * x)
16 rmse = np.sqrt(np.mean(residuals**2))
17 print(f"RMSE = {rmse:.0f} g")
18 print(f"Sum of residuals = {round(np.sum(residuals), 6) + 0.0}")
19
20 # A 200 mm flipper
21 print(f"Prediction for 200 mm: {b0 + b1 * 200:.0f} g")

```

```

mass = -5872.1 + 50.15 * flipper
RMSE = 392 g
Sum of residuals = 0.0
Prediction for 200 mm: 4159 g

```

Each extra millimeter of flipper goes with about 50 more grams of penguin. The RMSE of 392 g is less than half the 804 g of the always-guess-the-average baseline from the Classification and Regression chapter. (The `round(...) + 0.0` turns a sum like `-0.0000000004`, which is zero up to rounding error, into a plain `0.0`.)

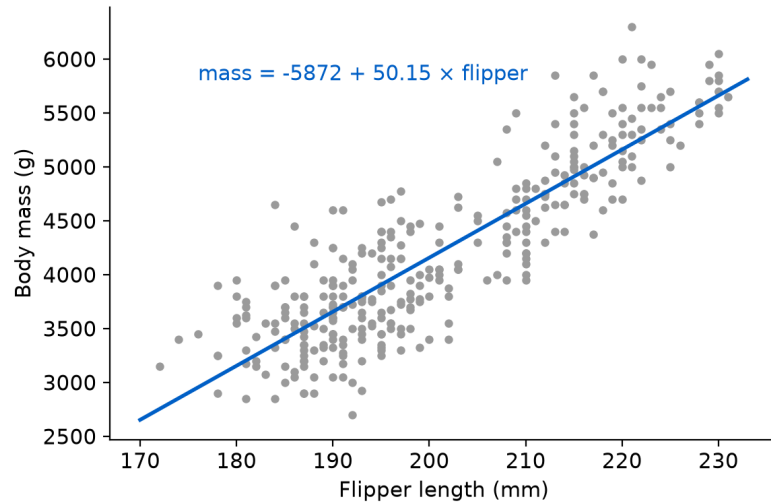


Figure 4.2: The least squares line for body mass against flipper length.

The intercept, -5872 g, is a penguin with no flippers weighing less than nothing. Like the quiz intercept, it is an extrapolation far outside the data (flippers here run from 172 to 231 mm), and you should not try to interpret it.

4.6 The Matrix Form

Formulas for b_0 and b_1 are fine for one feature, but we will want many. Linear algebra handles any number at once.

Make a *design matrix* X whose first column is all ones and whose second column holds the x values, and let $\mathbf{b} = (b_0, b_1)$. Then the predictions are $\hat{\mathbf{y}} = X\mathbf{b}$, and we want to minimize $\|\mathbf{y} - X\mathbf{b}\|^2$. Setting the gradient to zero gives the *normal equations*:

$$X^T X \mathbf{b} = X^T \mathbf{y}$$

For the quiz data,

$$X = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix}, \quad X^T X = \begin{pmatrix} 5 & 15 \\ 15 & 55 \end{pmatrix}, \quad X^T \mathbf{y} = \begin{pmatrix} 30.1 \\ 110.2 \end{pmatrix}$$

Notice the entries: n , $\sum x_i$, $\sum x_i^2$, $\sum y_i$, and $\sum x_i y_i$, the same sums we used by hand. Solving this 2×2 system gives $\mathbf{b} = (0.05, 1.99)$ again.

The normal equations also have a picture. They say $X^T(\mathbf{y} - X\mathbf{b}) = \mathbf{0}$: the residual vector is perpendicular to every column of X . Least squares drops a perpendicular from $\mathbf{y}$ onto the space of all possible predictions, exactly like finding the closest point on a plane.

In `normal.py`, NumPy solves the normal equations, and then solves the whole problem with `np.linalg.lstsq`, which is what you should use in practice (it is more accurate when the columns of X are nearly dependent):

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 x = data["flipper_length_mm"]
6 y = data["body_mass_g"]
7
8 # Design matrix: a column of ones, then the flipper lengths
9 X = np.column_stack([np.ones(len(x)), x])
10 print("X has shape", X.shape)
11
12 # Solve the normal equations
13 b = np.linalg.solve(X.T @ X, X.T @ y)
14 print("Normal equations:", np.round(b, 2))
15
16 # Let NumPy do the whole least squares problem
17 b, *_ = np.linalg.lstsq(X, y)
18 print("lstsq:          ", np.round(b, 2))
19
20 # Residuals are perpendicular to every column of X
21 residuals = y - X @ b
22 print("X^T e =", np.round(X.T @ residuals, 6) + 0.0)

```

```

X has shape (333, 2)
Normal equations: [-5872.09    50.15]
lstsq:          [-5872.09    50.15]
X^T e = [0.  0.]

```

`lstsq` returns four things (the solution, the residual sum of squares, the rank of X , and its singular values); `b, *_` keeps the first and throws the rest away.

4.7 Multiple Regression

Nothing in $X^T X \mathbf{b} = X^T \mathbf{y}$ cares how many columns X has. Add a column for each feature, and least squares finds the best *multiple regression*

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_p x_p$$

Category features go in as one-hot columns, as in the Preparing Data for Models chapter, dropping one category so the columns are not dependent. Here `multiple.py` predicts mass from flipper length, then adds sex, then adds species. It trains on 250 penguins and tests on the other 83, using the same split as the Preparing Data for Models chapter:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 y = data["body_mass_g"]
6 n = len(y)
7
8 columns = {
9     "intercept": np.ones(n),
10    "flipper": data["flipper_length_mm"],
11    "male": (data["sex"] == "male").astype(float),
12    "Chinstrap": (data["species"] == "Chinstrap").astype(float),
13    "Gentoo": (data["species"] == "Gentoo").astype(float),
14 }
15
16 # The same split as the Preparing Data chapter
17 order = np.random.default_rng(1).permutation(n)
18 train, test = order[:250], order[250:]
19
20 def rmse(X, b, rows):
21     return np.sqrt(np.mean((y[rows] - X[rows] @ b)**2))
22
23 models = [
24     ["intercept", "flipper"],
25     ["intercept", "flipper", "male"],
26     ["intercept", "flipper", "male", "Chinstrap", "Gentoo"]
27 ]
28 for names in models:
29     X = np.column_stack([columns[c] for c in names])
30     b, *_ = np.linalg.lstsq(X[train], y[train])
31     print(f"{len(names) - 1} features: train RMSE {rmse(X, b, train):.0f}, "
32           f"test RMSE {rmse(X, b, test):.0f}")
33
34 for name, coef in zip(names, b):
35     print(f"    {name:10s} {coef:8.1f}")

```

```

1 features: train RMSE 408, test RMSE 343
2 features: train RMSE 365, test RMSE 324
4 features: train RMSE 298, test RMSE 283
intercept    -220.6
flipper      19.1
male         568.4
Chinstrap    -92.0
Gentoo       887.6

```

Features	Training RMSE (g)	Test RMSE (g)
none (always guess the average)	804	—
flipper	408	343
flipper, sex	365	324
flipper, sex, species	298	283

Each feature helps, on the test set as well as the training set. (The test RMSE is lower than the training RMSE only because these 83 penguins happen to be a little easier to predict; with a different split, it could go the other way.)

Read the last model's coefficients carefully. Each one is the change in predicted mass when that feature goes up by one *and the others stay the same*:

- A male weighs about 568 g more than a female of the same species and flipper length.
- A Gentoo weighs about 888 g more than an Adelie (the dropped category) of the same sex and flipper length.
- The flipper slope fell from 50 to 19 g per mm. Much of what looked like a flipper effect was really a species effect: Gentoos have long flippers *and* heavy bodies.

4.8 Look at the Residuals

An RMSE is one number. A *residual plot* shows where the misses are. If the model has captured the pattern, the residuals should look like random noise around zero.

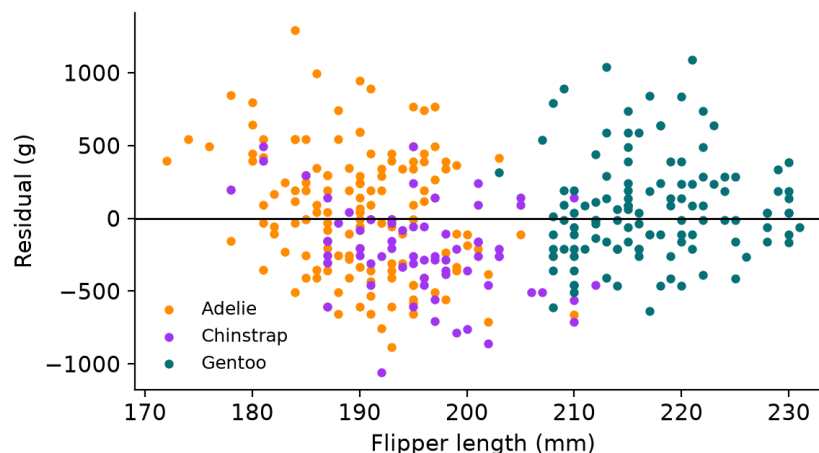


Figure 4.3: Residuals of the flipper-only line, colored by species.

In Figure 4.3, the Chinstrap residuals are mostly below zero; on average, the flipper-only line overestimates a Chinstrap by 216 g. That is a pattern the model missed, and the

species columns in the multiple regression fix it. Whenever residuals sort themselves by some variable, that variable belongs in the model.

4.9 A Line That Points the Wrong Way

Fit body mass against bill depth for all 333 penguins and you get a slope of -193 g per mm: deeper bills, lighter penguins. Fit it within each species, and the slope is positive every time.

Penguins	Slope (g per mm of bill depth)
all 333	-193
Adelie	$+218$
Chinstrap	$+205$
Gentoo	$+368$

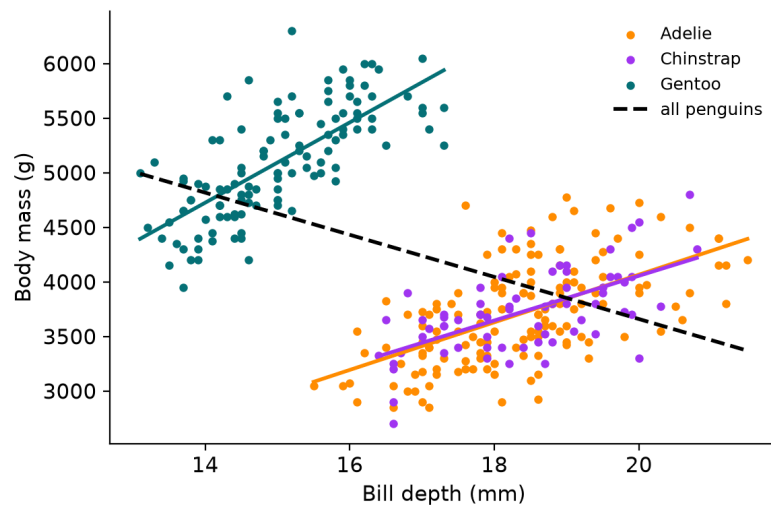


Figure 4.4: Within each species, deeper bills go with heavier penguins. Across all penguins, the line slopes the other way, because Gentoos have shallow bills and heavy bodies.

A trend that reverses when you split the data into groups is called *Simpson's paradox*. The species is a *lurking variable*: it affects both bill depth and mass. A line fit to the whole group tells you how the variables move together in this collection of penguins, not what would happen to one penguin if its bill got deeper. Putting species in the model, as multiple regression does, is one way to protect yourself.

4.10 When Least Squares Is Not Enough

Least squares works best when the relationship is close to linear and the errors are about the same size everywhere. Because it squares errors, a single wild point can drag the line toward it; check for outliers before you trust a fit, and consider *robust* methods, which use other measures of error, when outliers are real. For relationships that curve, you can still use least squares by adding columns like x^2 , as the Training, Validation, and Testing chapter did. Too many such columns, though, and you are back to overfitting.

Exercise 20 Solve by hand

Find the least squares line for the points $(0, 1)$, $(1, 3)$, $(2, 4)$, $(3, 8)$.

Working Space

Answer on Page 55

Exercise 21 Predict and check

Using the line from the previous exercise, predict y at $x = 4$, and find SS_{res} .

Working Space

Answer on Page 55

Exercise 22 **Predict a value**

Using the quiz line $\hat{y} = 0.05 + 1.99x$, predict the score for a student who studies 4 hours. What is the residual for the student who actually studied 4 hours?

*Working Space**Answer on Page 55***Exercise 23** **Why square?**

Give one reason to minimize the sum of squared errors instead of the sum of absolute errors, and one reason not to.

*Working Space**Answer on Page 55***Exercise 24** **Spot the problem**

A study of 200 cities finds that cities with more ice cream sales have more drownings, with a positive regression slope. A news site says, “Ice cream causes drowning.” What is wrong with this conclusion?

*Working Space**Answer on Page 55***Exercise 25** **Read the coefficients**

What does the coefficient of -92.0 in the last model mean? Why is there no coef-

Working Space

ficient for Adelie penguins?

Answer on Page 55

Exercise 26 **Your own Simpson's paradox**

Modify `line.py` to reproduce the table of bill-depth slopes: one line for all penguins and one for each species. Then fit mass from flipper length within each species. Does the flipper slope change sign?

Working Space

Answer on Page 56

Exercise 27 **Everything in**

Add bill length and bill depth to the last model in `multiple.py`. What happens to the training and test RMSE? Is this a big improvement?

Working Space

Answer on Page 56

Linear Regression and Gradient Descent

You are hiking down a mountain when the fog rolls in. You cannot see the parking lot, or even the trail. But you can feel which way the ground slopes under your feet, so you take a step downhill. Then another. As long as the mountain has one lowest point, you will eventually get there.

That is *gradient descent*, and it is how nearly every model after this chapter is trained, including the neural networks at the end of the unit. Here we use it on a problem we have already solved, linear regression, so you can check every answer against the least squares formula.

5.1 Why Not Just Use the Formula?

The Least Squares chapter gave an exact answer: solve $X^T X \mathbf{b} = X^T \mathbf{y}$. So why walk downhill in the fog when you have a map?

- **Most models have no formula.** Logistic regression, in the next chapter, and neural networks have no equation you can solve for the best parameters. You can only search for them.
- **Big problems.** With a million features, $X^T X$ has a trillion entries. Gradient descent never forms it.
- **Big data.** Gradient descent can learn from a small batch of examples at a time, so the data does not have to fit in memory at once.

5.2 The Cost Function and Its Gradient

We call the parameters θ (theta), which is the usual name in machine learning; for a line, $\theta = (\theta_0, \theta_1)$ is the intercept and the slope. The “height of the mountain” is the *cost function*, here the mean squared error:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i^T \theta - y_i)^2 = \frac{1}{n} \|X\theta - \mathbf{y}\|^2$$

where $\mathbf{x}_i$ is row i of the design matrix (a 1 followed by the features). The gradient, the direction of steepest uphill, is

$$\nabla J(\boldsymbol{\theta}) = \frac{2}{n} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Read it from the inside out: $\mathbf{X}\boldsymbol{\theta} - \mathbf{y}$ is the vector of prediction errors, and multiplying by $\mathbf{X}^T$ asks, for each parameter, “if I nudged this one, how would the errors change?”

5.3 The Update Rule

Start with a guess $\boldsymbol{\theta}_0$, usually all zeros. Then repeat:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla J(\boldsymbol{\theta})$$

The minus sign means “go downhill.” The *learning rate* α sets the step size. Too small and you creep; too large and you overshoot the bottom of the valley and land higher up the other side.

Example: Take one step on the quiz data from the Least Squares chapter, $x = 1, \dots, 5$ and $y = 2.1, 3.9, 6.2, 7.8, 10.1$, starting at $\boldsymbol{\theta}_0 = (0, 0)$ with $\alpha = 0.01$.

Solution: With $\boldsymbol{\theta}_0 = \mathbf{0}$, every prediction is 0, so $\mathbf{X}\boldsymbol{\theta}_0 - \mathbf{y} = -\mathbf{y}$. The first entry of $\mathbf{X}^T(-\mathbf{y})$ is $-\sum y_i = -30.1$, and the second is $-\sum x_i y_i = -110.2$. With $n = 5$,

$$\nabla J(\boldsymbol{\theta}_0) = \frac{2}{5}(-30.1, -110.2) = (-12.04, -44.08)$$

$$\boldsymbol{\theta}_1 = (0, 0) - 0.01(-12.04, -44.08) = (0.1204, 0.4408)$$

The cost drops from $J(\boldsymbol{\theta}_0) = 44.18$ to $J(\boldsymbol{\theta}_1) = 25.77$. The answer is $(0.05, 1.99)$, so we are heading the right way, with a long way to go.

5.4 Letting the Computer Walk

By hand, two steps is plenty. A computer can take thousands. Here is `quiz_descent.py`:

```

1 import numpy as np
2
3 x = np.array([1, 2, 3, 4, 5])
4 y = np.array([2.1, 3.9, 6.2, 7.8, 10.1])
5 X = np.column_stack([np.ones(len(x)), x])
6
7 def descend(alpha, steps):
```

```

8     theta = np.zeros(2)
9     for _ in range(steps):
10         gradient = 2 / len(y) * X.T @ (X @ theta - y)
11         theta = theta - alpha * gradient
12     return theta
13
14 print(descend(0.05, 50))
15 print(descend(0.05, 5000))
16 print(descend(0.1, 50))

```

```
[0.24846555  1.93502823]
```

```
[0.05  1.99]
```

```
[-3074524.42217419 -11100007.21998943]
```

With $\alpha = 0.05$, 50 steps get close and 5000 steps land on the least squares answer. Doubling the learning rate to 0.1 makes every step overshoot by more than the last, and after 50 steps the parameters are in the millions.

5.5 Valleys and Learning Rates

Figure 5.1 draws the cost as a contour map, like a hiking map, with the minimum marked by a star. The contours are long, thin ovals: the cost changes quickly if you change the slope θ_1 , but slowly if you change the intercept θ_0 . That shape causes most of the trouble with gradient descent.

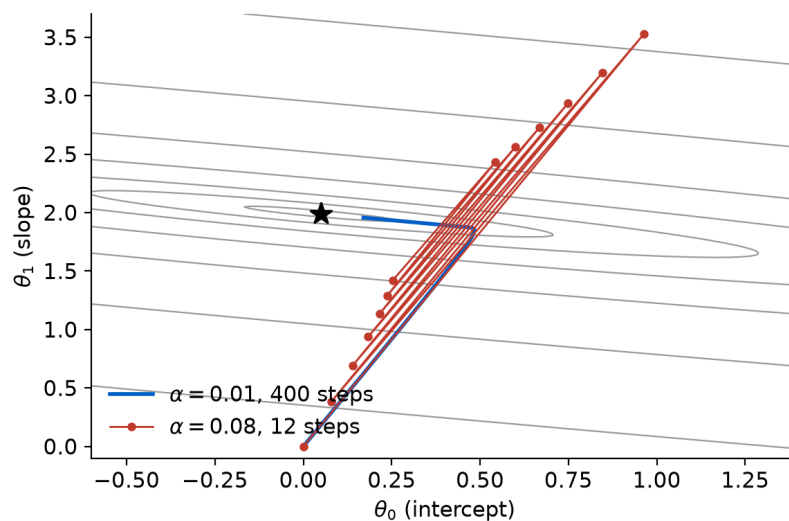


Figure 5.1: Contours of the cost for the quiz data, and two gradient descent paths from $(0,0)$.

With $\alpha = 0.01$ (blue), the path heads quickly down to the floor of the valley, then crawls along it. With $\alpha = 0.08$ (red), each step is so big that it crosses the valley, and the path zigzags from wall to wall while slowly working its way toward the star. A little bigger and the zigzags grow instead of shrinking. Figure 5.2 shows the cost after each step for four learning rates.

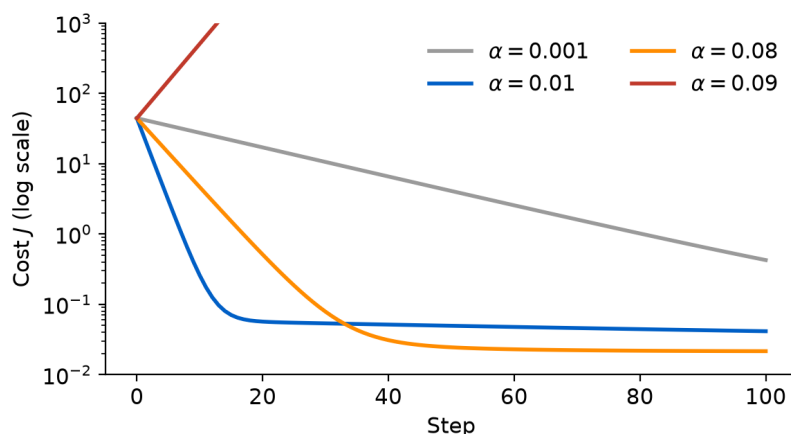


Figure 5.2: Cost against step on the quiz data. The vertical axis is logarithmic.

Learning rate	What happens on the quiz data
0.001	steady but slow; still far from the bottom after 100 steps
0.01	drops fast, then crawls along the valley floor
0.08	zigzags, but ends lowest after 100 steps
0.09	each zigzag is bigger than the last: the cost explodes

The cutoff between 0.08 and 0.09 is not luck. For a squared-error cost, gradient descent converges exactly when α is less than 2 divided by the largest eigenvalue of the matrix $\frac{2}{n}X^T X$. For the quiz data, that is about 0.0845. In practice, people just try a few values, as in the table, and keep the one whose cost goes down fastest without blowing up.

5.6 Standardize First

Now the penguins. In the Preparing Data for Models chapter, standardizing features mattered because of distances. For gradient descent, it matters even more.

With raw flipper lengths around 200 mm, the cost valley is stretched more than 8 million times longer in one direction than the other. The largest learning rate that does not blow up is about 0.0000246, so steps along the valley are microscopic. Standardize the flipper lengths and the valley becomes a perfectly round bowl. Here is `penguin_descent.py`:

```

1 import numpy as np
2 from penguins import load_penguins
3
4 data = load_penguins()
5 flipper = data["flipper_length_mm"]
6 y = data["body_mass_g"]
7 n = len(y)
8
9 def descend(X, alpha, steps):
10     theta = np.zeros(X.shape[1])
11     for _ in range(steps):
12         gradient = 2 / n * X.T @ (X @ theta - y)
13         theta = theta - alpha * gradient
14         rmse = np.sqrt(np.mean((X @ theta - y)**2))
15     return theta, rmse
16
17 # Raw flipper lengths: anything much above 2e-5 blows up
18 X_raw = np.column_stack([np.ones(n), flipper])
19 for steps in [100, 10_000, 1_000_000]:
20     theta, rmse = descend(X_raw, 2e-5, steps)
21     print(f"raw, {steps:>9} steps:", np.round(theta, 1) + 0.0, f"RMSE {rmse:.0f}")
22
23 # Standardized flipper lengths
24 mean, sd = np.mean(flipper), np.std(flipper)
25 X_std = np.column_stack([np.ones(n), (flipper - mean) / sd])
26 for steps in [10, 30, 100]:
27     theta, rmse = descend(X_std, 0.1, steps)
28     print(f"standardized, {steps:>3} steps:", np.round(theta, 1),
29           f"RMSE {rmse:.0f}")
30
31 # Convert back to grams per millimeter
32 b1 = theta[1] / sd
33 b0 = theta[0] - b1 * mean
34 print(f"mass = {b0:.1f} + {b1:.2f} * flipper")

```

```

raw,      100 steps: [ 0.  21.1] RMSE 566
raw,    10,000 steps: [-11.2  21.1] RMSE 565
raw, 1,000,000 steps: [-1030.7  26.2] RMSE 517
standardized,  10 steps: [3755.3  626.5] RMSE 603
standardized,  30 steps: [4201.8  701. ] RMSE 392
standardized, 100 steps: [4207.1  701.9] RMSE 392
mass = -5872.1 + 50.15 * flipper

```

Flipper length	Steps	RMSE (g)
raw	1,000,000	517
standardized	30	392
least squares formula	—	392

A million steps on raw data still have not found the line, because the intercept needs

to reach -5872 and it is moving a few thousandths of a gram per step. Thirty steps on standardized data match the formula. Converting back gives the same line as the Least Squares chapter. On the standardized scale, the coefficients are easy to read, too: $\theta_0 = 4207$ g is the average mass, and $\theta_1 = 702$ g is the change in mass for a flipper one standard deviation (14 mm) longer.

5.7 Mini-Batches

Each step above used all 333 penguins to compute the gradient. With a billion examples, one step would take a very long time. *Mini-batch gradient descent* estimates the gradient from a small random batch, say 32 examples, and steps right away. The estimate is noisy, but you get many more steps for the same amount of work. (With a batch of a single example, the method is called *stochastic gradient descent*, or SGD; people often use that name for mini-batches, too.)

One full pass through the data is called an *epoch*. Here is `minibatch.py`:

```

1  import numpy as np
2  from penguins import load_penguins
3
4  data = load_penguins()
5  flipper = data["flipper_length_mm"]
6  y = data["body_mass_g"]
7  n = len(y)
8  X = np.column_stack([np.ones(n), (flipper - np.mean(flipper)) / np.std(flipper)])
9
10 rng = np.random.default_rng(0)
11 theta = np.zeros(2)
12 alpha, batch_size = 0.1, 32
13 for epoch in range(1, 6):
14     order = rng.permutation(n)           # shuffle once per epoch
15     for start in range(0, n, batch_size):
16         rows = order[start:start + batch_size]
17         Xb, yb = X[rows], y[rows]
18         gradient = 2 / len(rows) * Xb.T @ (Xb @ theta - yb)
19         theta = theta - alpha * gradient
20     rmse = np.sqrt(np.mean((X @ theta - y)**2))
21     print(f"epoch {epoch}: theta = {np.round(theta, 1)}, RMSE = {rmse:.1f}")

```

```

epoch 1: theta = [3822.4  648.1], RMSE = 551.9
epoch 2: theta = [4179.6  685.6], RMSE = 393.5
epoch 3: theta = [4219.4  685.9], RMSE = 392.7
epoch 4: theta = [4181.5  712. ], RMSE = 393.1
epoch 5: theta = [4207.4  692.1], RMSE = 392.3

```

After two epochs, the mini-batch version is within 2 g of the best possible RMSE. The full-

batch version, with one step per epoch, would still be above 2700 g (Figure 5.3). Notice that the mini-batch θ never settles down exactly; it jitters around the answer, because each batch pulls in a slightly different direction. In practice, people often shrink α over time to calm the jitter.

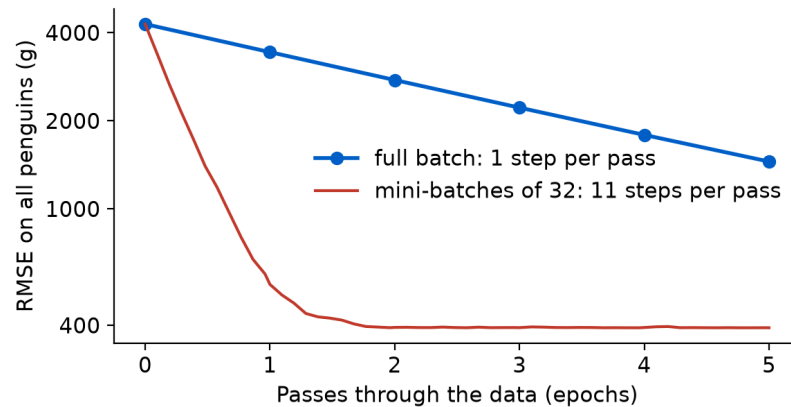


Figure 5.3: RMSE on the penguins against the number of passes through the data. The vertical axis is logarithmic.

You will see this exact loop (shuffle, slice a batch, compute the gradient, step) again when you train neural networks with PyTorch.

Exercise 28 One more step

Starting from $\theta_1 = (0.1204, 0.4408)$, take one gradient step with $\alpha = 0.01$ on the quiz data. Find θ_2 , rounding to four decimal places.

Working Space

Answer on Page 56

Exercise 29 Learning rate

What happens to gradient descent if α is made very large? What if it is made very small?

Working Space

Answer on Page 56

Exercise 30 A one-parameter mountain

Fit a line through the origin, $\hat{y} = \theta x$, to the points $(1, 2)$ and $(2, 3)$, so

$$J(\theta) = \frac{1}{2} \left[(\theta - 2)^2 + (2\theta - 3)^2 \right]$$

Find $J'(\theta)$. Then, starting at $\theta = 0$ with $\alpha = 0.2$, take two steps. What value is gradient descent heading for?

Working Space

Answer on Page 57

Exercise 31 Find the cliff

On the quiz data, try learning rates from 0.080 to 0.090 in steps of 0.002, with 2000 steps each. Which ones find the answer?

Working Space

Where is the cutoff?

Answer on Page 57

Exercise 32 Two features

Extend `penguin_descent.py` to predict mass from standardized flipper length *and* standardized bill depth (three parameters). Run 200 steps with $\alpha = 0.1$, and compare your answer with the one from `lstsq`.

Working Space

Answer on Page 57

Exercise 33 Batch size

In `minibatch.py`, try batch sizes of 1, 8, 32, and 333 for five epochs each. Which gets close in the fewest epochs? Which ends with the steadiest θ ? Explain the trade-off.

Working Space

Answer on Page 57

Answers to Exercises

Answer to Exercise 1 (on page 8)

(a) Regression, since temperature is a number. (b) Classification, multi-class, with three categories. (c) Classification, binary, since default is yes or no.

Answer to Exercise 2 (on page 9)

RMSE or MAE. Accuracy requires an exact match, and a predicted price is almost never exactly equal to the true price, so accuracy is not meaningful for a number.

Answer to Exercise 3 (on page 9)

Regression: predict a student's exam score (a number) from hours of study. Classification: predict whether a student passes (yes or no) from hours of study. The inputs are the same; only the kind of label changes.

Answer to Exercise 4 (on page 9)

The errors are 1, 0, and -3 . $RMSE = \sqrt{(1+0+9)/3} = \sqrt{10/3} \approx 1.83$ and $MAE = (1+0+3)/3 \approx 1.33$.

Answer to Exercise 5 (on page 10)

```
1 prediction = np.zeros(len(mass))
2 for name in names:
3     same = species == name
4     prediction[same] = np.mean(mass[same])
5 rmse = np.sqrt(np.mean((mass - prediction)**2))
6 print(f"Species average: RMSE = {rmse:.0f} g")
```

The RMSE drops to about 459 g. It is not a fair baseline, because it uses a feature (the species); it is really a simple model. It shows how much the species alone tells you about mass.

Answer to Exercise 6 (on page 10)

The data has 168 males and 165 females, so always guessing “male” is right $168/333 \approx 50.5\%$ of the time. A classifier that scores 52% is barely better than not looking at the penguin at all.

Answer to Exercise 7 (on page 17)

Training: $0.70 \cdot 2400 = 1680$. Validation: $0.15 \cdot 2400 = 360$. Testing: 360.

Answer to Exercise 8 (on page 17)

The mean and standard deviation of the validation and test rows were used when rescaling the training rows, so information from the test set leaked into training. The fix is to compute the rescaling from the training set only, and then apply it to the other sets.

Answer to Exercise 9 (on page 18)

The choice of k should be made on the validation set (or by cross-validation). Picking it by the test score means the test set has been used for tuning, so the reported test score is optimistic.

Answer to Exercise 10 (on page 18)

Degree 1 underfits a little: both errors are higher than they need to be. From degree 5 on, the models overfit: training error keeps dropping while validation error rises, sharply after degree 5. Degree 3 has the lowest validation error (386 g), so the validation set picks degree 3.

Answer to Exercise 11 (on page 18)

```

1 species = data["species"]
2 train, val, test = [], [], []
3 for kind in np.unique(species):
4     rows = rng.permutation(np.flatnonzero(species == kind))
5     a, b = int(0.70 * len(rows)), int(0.85 * len(rows))
6     train += list(rows[:a])
7     val += list(rows[a:b])
8     test += list(rows[b:])

```

`np.flatnonzero(species == kind)` gives the row numbers of one species. Printing the shares as before gives

```

train 232 Adelie 44%, Chinstrap 20%, Gentoo 36%
val 50 Adelie 44%, Chinstrap 20%, Gentoo 36%
test 51 Adelie 43%, Chinstrap 22%, Gentoo 35%

```

Answer to Exercise 12 (on page 19)

Add `np.std(errors, ddof=1) / np.sqrt(k)` to the printout. The standard errors are all about 15 to 17 g:

degree	CV RMSE	standard error
1	389.6	16.6
2	376.1	16.0
3	371.1	15.4
4	370.6	15.3
5	371.5	15.6

The best score is 370.6, and $370.6 + 15.3 = 385.9$. The simplest degree under that line is degree 2. (Its test RMSE is about 412 g, essentially the same as degree 4's 407 g.)

Answer to Exercise 13 (on page 26)

$$z = \frac{65-50}{10} = 1.5.$$

Answer to Exercise 14 (on page 27)

Math: $z = \frac{85-75}{5} = 2$. Science: $z = \frac{90-85}{10} = 0.5$. Your math score is two standard deviations above the mean, and your science score only half a standard deviation above. Math went better, even though the raw score was lower.

Answer to Exercise 15 (on page 27)

The test data helped determine the scaling, so information from the test set has leaked into training. Compute μ and σ from the training set only, and apply those same values to the test set.

Answer to Exercise 16 (on page 27)

(1,0,0), (0,1,0), (0,0,1), (1,0,0).

Answer to Exercise 17 (on page 28)

The three columns add up to 1 in every row, which is the same as the intercept's column of 1s. The regression cannot separate their effects, so its coefficients are not determined. Dropping one column fixes this, and the dropped island becomes the baseline.

Answer to Exercise 18 (on page 28)

```
1 M = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
2 j = nearest(M, mystery)
3 print("Nearest after min-max scaling:", species[j], X[j])
4 print(np.round(M[0], 3))
```

The nearest neighbor is the same near-twin Adelie (41.5, 18.3, 195, 4300). Penguin 0 becomes (0.255, 0.667, 0.153, 0.292): for example, its bill length of 39.1 mm is 25.5% of the way from the shortest bill (32.1 mm) to the longest (59.6 mm).

Answer to Exercise 19 (on page 28)

```
1 kinds = np.unique(data["species"]) # Adelie, Chinstrap, Gentoo
2 onehot = (data["species"][:, None] == kinds[None, :]).astype(float)
3 print(onehot.shape, np.all(onehot.sum(axis=1) == 1))
```

It prints (333, 3) True.

Answer to Exercise 20 (on page 38)

$\bar{x} = 1.5$, $\bar{y} = 4$, $S_{xx} = 5$, and $S_{xy} = 11$. So $b_1 = \frac{11}{5} = 2.2$ and $b_0 = 4 - 2.2(1.5) = 0.7$. The line is $\hat{y} = 0.7 + 2.2x$.

Answer to Exercise 21 (on page 38)

$\hat{y} = 0.7 + 2.2(4) = 9.5$. For the four points, the predictions are 0.7, 2.9, 5.1, and 7.3. The residuals are 0.3, 0.1, -1.1 , and 0.7, so $SS_{\text{res}} = 0.09 + 0.01 + 1.21 + 0.49 = 1.80$.

Answer to Exercise 22 (on page 39)

$\hat{y} = 0.05 + 1.99(4) = 8.01$. That student scored 7.8, so the residual is $7.8 - 8.01 = -0.21$.

Answer to Exercise 23 (on page 39)

For: squared errors are differentiable everywhere, so calculus gives a formula for the answer. (Also acceptable: squaring punishes one big miss more than several small ones, so the line avoids being badly wrong anywhere.) Against: a single outlier can pull the line strongly, because its huge error gets squared.

Answer to Exercise 24 (on page 39)

The data are observational, so the slope shows only an association. A lurking variable, hot weather, drives both: people buy more ice cream and swim more on hot days, and more swimming leads to more drownings.

Answer to Exercise 25 (on page 39)

A Chinstrap is predicted to weigh about 92 g less than an Adelie of the same sex and flipper length. Adelie is the dropped category: it is the baseline the other species are compared to. Including all three species columns would make them add up to the intercept column, and $X^T X$ would not be invertible.

Answer to Exercise 26 (on page 40)

```

1 def fit_line(x, y):
2     b1 = np.sum((x - x.mean()) * (y - y.mean())) / np.sum((x - x.mean())**2)
3     return y.mean() - b1 * x.mean(), b1
4
5 depth = data["bill_depth_mm"]
6 species = data["species"]
7 print("all", fit_line(depth, y)[1])
8 for name in ["Adelie", "Chinstrap", "Gentoo"]:
9     k = species == name
10    print(name, fit_line(depth[k], y[k])[1], fit_line(x[k], y[k])[1])

```

The flipper slopes are about 33 (Adelie), 35 (Chinstrap), and 54 (Gentoo) g per mm. All are positive, like the overall slope of 50, so flipper length shows no paradox. The within-species slopes are smaller than 50 for Adelies and Chinstraps, though, because part of the overall slope comes from species differences.

Answer to Exercise 27 (on page 40)

Add "bill_length": data["bill_length_mm"] and "bill_depth": data["bill_depth_mm"] to columns and a fourth model with all six features. The training RMSE falls from 298 to 288 g, and the test RMSE from 283 to 276 g. It helps a little, but most of what can be learned from these measurements is already in flipper length, sex, and species.

Answer to Exercise 28 (on page 47)

The predictions are $X\theta_1 = (0.5612, 1.0020, 1.4428, 1.8836, 2.3244)$. The errors are $X\theta_1 - \mathbf{y} = (-1.5388, -2.8980, -4.7572, -5.9164, -7.7756)$. Then $X^T(X\theta_1 - \mathbf{y}) = (-22.886, -84.150)$, so

$$\nabla J = \frac{2}{5}(-22.886, -84.150) = (-9.1544, -33.660)$$

$$\theta_2 = (0.1204, 0.4408) - 0.01(-9.1544, -33.660) = (0.2119, 0.7774)$$

Answer to Exercise 29 (on page 48)

Very large: the steps overshoot the minimum. The parameters oscillate or diverge, and the cost can increase instead of decrease. Very small: the cost goes down every step, but so slowly that you may run out of patience (or computer time) long before reaching the

minimum.

Answer to Exercise 30 (on page 48)

$J'(\theta) = (\theta - 2) + 2(2\theta - 3) = 5\theta - 8$. Step 1: $\theta = 0 - 0.2(-8) = 1.6$. Step 2: $J'(1.6) = 0$, so θ stays at 1.6. The minimum is where $5\theta - 8 = 0$, at $\theta = 1.6$. (This learning rate happens to land on it in one step.)

Answer to Exercise 31 (on page 48)

```
1 for alpha in np.arange(0.080, 0.0901, 0.002):
2     print(f"{alpha:.3f}", np.round(descend(alpha, 2000), 3))
```

0.080, 0.082, and 0.084 converge; 0.086, 0.088, and 0.090 blow up. The cutoff is about 0.0845, matching the eigenvalue rule in the chapter.

Answer to Exercise 32 (on page 49)

```
1 depth = data["bill_depth_mm"]
2 Z = np.column_stack([np.ones(n), (flipper - mean) / sd,
3                       (depth - np.mean(depth)) / np.std(depth)])
4 theta, rmse = descend(Z, 0.1, 200)
5 print(np.round(theta, 1), round(rmse, 1))
6 print(np.round(np.linalg.lstsq(Z, y)[0], 1))
```

Both give about (4207.1, 724.5, 39.1), and the RMSE is 391 g. With two standardized features that are correlated, the bowl is no longer perfectly round, but it is round enough that 200 steps are plenty.

Answer to Exercise 33 (on page 49)

A batch size of 8 is close after one epoch (RMSE about 402 g), because it takes 42 steps per epoch. A batch size of 1 takes the most steps, but each one follows a single penguin, so θ jumps by more than 100 g at a time, and the RMSE after five epochs (about 450 g) is worse than after four. A batch of 333 is ordinary gradient descent: perfectly smooth, but only five steps in five epochs, so the RMSE is still about 1450 g. Small batches buy speed with noise; batches of a few dozen get most of the speed with little noise, which is why

they are the usual choice.



INDEX

- k-fold cross-validation, 15
- accuracy, 7
- baseline, 25
- baseline category, 25
- baseline model, 7
- baselines, 7
- binary, 6
- binary classification, 6
- broadcasting, 23
- categorical variable, 24
- categorical variables, 24
- classification, 5
- cost function, 41
- cross-validation, 15
- Data leakage, 15
- data leakage, 15
- design matrix, 33
- dummy variable trap, 25
- epoch, 46
- errors, 6
- extrapolation, 31
- feature, 4
- features, 4
- fold, 15
- folds, 15
- gradient descent, 41
- hyperparameter, 13
- hyperparameters, 13
- imputation, 25
- impute, 25
- intercept, 29
- label, 4
- learning rate, 42
- least squares, 29
- least squares line, 30
- lurking variable, 37
- machine learning, 3
- mean absolute error, 6
- min-max scaling, 24
- mini-batch, 46
- Mini-batch gradient descent, 46
- missing values, 25
- model, 4
- multi-class, 6
- multi-class classification, 6
- multiple regression, 34
- normal equations, 30, 33
 - matrix form, 33
- One-hot encoding, 25
- one-hot encoding, 25
- one-standard-error rule, 19
- overfitting, 11, 12

- Palmer penguins, 3
- predictor, 29
- predictor variable, 29

- regression, 4
 - multiple, 34
 - simple linear, 29
 - supervised, 4
- residual, 6, 30
- residual plot, 36
- residuals, 6
- response, 29
- response variable, 29
- root mean squared error, 6

- Simple linear regression, 29
- simple linear regression, 29
- Simpson's paradox, 37
- slope, 29
- Standardization, 23
- standardization, 23
 - gradient descent, 44
- stochastic gradient descent, 46
- stratified split, 14
- sum of squared errors, 30
- supervised learning, 4

- test set, 13
- training set, 13

- underfitting, 12
- unsupervised learning, 4

- validation set, 13