

# **Data Science 1**

Tobin Driscoll

2023-01-01

# Table of contents

<b>Preface</b>	<b>7</b>
Prerequisites . . . . .	7
<b>Resources</b>	<b>8</b>
Useful guides . . . . .	8
Data sources . . . . .	8
Search engines . . . . .	8
Packaged . . . . .	8
Open-ended . . . . .	9
Glossary . . . . .	9
Git . . . . .	9
Notebooks . . . . .	9
Python . . . . .	10
Editors/IDEs . . . . .	10
<b>Getting started</b>	<b>12</b>
Back end . . . . .	12
Run on your own computer . . . . .	12
Run on the cloud . . . . .	12
Front end . . . . .	13
On your own machine . . . . .	13
Tips on Jupyter success . . . . .	13
<b>1 Representation of data</b>	<b>15</b>
1.1 Quantitative data . . . . .	15
1.1.1 Special values . . . . .	16
1.1.2 Dates and times . . . . .	17
1.1.3 Random numbers . . . . .	18
1.2 Arrays . . . . .	19
1.2.1 Vectors . . . . .	20
1.2.2 Multiple dimensions . . . . .	23
1.3 Qualitative data . . . . .	27
1.3.1 Categorical . . . . .	27
1.3.2 Text . . . . .	28
1.3.3 Images . . . . .	29

1.4	Series and frames . . . . .	29
1.4.1	Common operations . . . . .	34
1.4.2	Loading from files . . . . .	35
1.4.3	Selecting rows . . . . .	36
1.5	Data preparation . . . . .	40
1.5.1	Missing values . . . . .	40
1.5.2	Loans example . . . . .	42
1.5.3	Diamonds example . . . . .	44
	Exercises . . . . .	46
<b>2</b>	<b>Descriptive statistics</b>	<b>47</b>
2.1	Summary statistics . . . . .	47
2.1.1	Mean and spread . . . . .	48
2.1.2	z-scores . . . . .	49
2.1.3	Populations and samples . . . . .	50
2.1.4	Median and quantiles . . . . .	52
2.2	Distributions . . . . .	54
2.2.1	CDF . . . . .	54
2.2.2	Empirical CDF . . . . .	55
2.2.3	PDF . . . . .	57
2.2.4	Mean and variance . . . . .	61
2.2.5	Normal distribution . . . . .	62
2.3	Grouping data . . . . .	64
2.3.1	Splitting . . . . .	65
2.3.2	Aggregation . . . . .	71
2.3.3	Transformation . . . . .	72
2.3.4	Filtering . . . . .	74
2.4	Outliers . . . . .	75
2.4.1	IQR . . . . .	76
2.4.2	Mean and STD . . . . .	79
2.5	Correlation . . . . .	81
2.5.1	Relational plots . . . . .	81
2.5.2	Covariance . . . . .	84
2.5.3	Pearson coefficient . . . . .	85
2.5.4	Spearman coefficient . . . . .	87
2.5.5	Categorical correlation . . . . .	88
2.6	Cautionary tales . . . . .	90
2.6.1	The datasaurus . . . . .	90
2.6.2	Correlation vs. dependence . . . . .	95
2.6.3	Simpson's paradox . . . . .	96
	Exercises . . . . .	98

<b>3 Classification</b>	<b>100</b>
3.1 Classification basics . . . . .	100
3.1.1 Encoding qualitative data . . . . .	101
3.1.2 Walkthrough . . . . .	102
3.2 Classifier performance . . . . .	105
3.2.1 Train–test paradigm . . . . .	106
3.2.2 Binary classifiers . . . . .	108
3.2.3 Multiclass classifiers . . . . .	113
3.3 Decision trees . . . . .	116
3.3.1 Gini impurity . . . . .	116
3.3.2 Partitioning . . . . .	117
3.3.3 Usage . . . . .	118
3.4 Nearest neighbors . . . . .	125
3.4.1 Distances and norms . . . . .	125
3.4.2 Algorithm . . . . .	126
3.4.3 Standardization . . . . .	131
3.5 Probabilistic interpretation . . . . .	133
3.5.1 ROC curve . . . . .	134
3.5.2 AUC . . . . .	140
Exercises . . . . .	141
<b>4 Model selection</b>	<b>145</b>
4.1 Bias–variance tradeoff . . . . .	146
4.1.1 Learner bias . . . . .	146
4.1.2 Variance . . . . .	147
4.1.3 Learning curves . . . . .	150
4.2 Overfitting . . . . .	152
4.2.1 Overfitting in kNN . . . . .	152
4.2.2 Overfitting in decision trees . . . . .	154
4.2.3 Overfitting and variance . . . . .	156
4.3 Ensemble methods . . . . .	160
4.4 Validation . . . . .	167
4.4.1 Cross-validation . . . . .	167
4.4.2 Hyperparameter tuning . . . . .	169
<b>5 Regression</b>	<b>175</b>
5.1 Linear regression . . . . .	176
5.1.1 Linear algebra . . . . .	177
5.1.2 Performance metrics . . . . .	178
5.2 Multilinear regression . . . . .	189
5.2.1 Polynomial regression . . . . .	197
5.3 Regularization . . . . .	202
5.4 Nonlinear regression . . . . .	208
5.4.1 Nearest neighbors . . . . .	209

5.4.2	Decision tree . . . . .	211
5.5	Logistic regression . . . . .	216
5.5.1	Loss function . . . . .	217
5.5.2	Regularization . . . . .	220
5.5.3	Multiclass classification . . . . .	222
	Exercises . . . . .	225
<b>6</b>	<b>Clustering</b>	<b>229</b>
6.1	Similarity . . . . .	233
6.1.1	Distance matrix . . . . .	234
6.1.2	Angular distance . . . . .	236
6.1.3	Distance in high dimensions . . . . .	237
6.2	Performance measures . . . . .	238
6.2.1	Rand index and ARI . . . . .	238
6.2.2	Silhouettes . . . . .	241
6.3	k-means . . . . .	248
6.3.1	Lloyd's algorithm . . . . .	249
6.3.2	Practical issues . . . . .	249
6.4	Hierarchical clustering . . . . .	258
	Exercises . . . . .	268
<b>7</b>	<b>Networks</b>	<b>270</b>
7.1	Graphs . . . . .	270
7.1.1	A graph menagerie . . . . .	275
7.1.2	Adjacency . . . . .	280
7.1.3	Degree . . . . .	282
7.2	Random graphs . . . . .	283
7.2.1	Erdős-Rényi graphs . . . . .	283
7.2.2	Watts–Strogatz graphs . . . . .	287
7.3	Clustering . . . . .	290
7.3.1	ER graphs . . . . .	294
7.3.2	WS graphs . . . . .	296
7.4	Distance . . . . .	298
7.4.1	ER graphs . . . . .	300
7.4.2	WS graphs . . . . .	303
7.4.3	Twitch network . . . . .	305
7.5	Degree distributions . . . . .	309
7.5.1	Power-law distribution . . . . .	313
7.5.2	Barabási–Albert graphs . . . . .	315
7.6	Centrality . . . . .	318
7.6.1	Degree centrality . . . . .	319
7.6.2	Betweenness centrality . . . . .	320
7.6.3	Eigenvector centrality . . . . .	325
7.7	Friendship paradox . . . . .	329

7.8 Communities . . . . .	331
7.8.1 Simulating the random walk . . . . .	334
7.8.2 Label propagation . . . . .	339
Exercises . . . . .	343

# Preface

This book covers material for Math 219, Data Science 1, at the University of Delaware. It should be considered prepublication and may have errors.

See [license file](#) for rights information.

This site was made with Quattro. To learn more about Quarto books visit <https://quarto.org/docs/books>.

[https://www.dropbox.com/s/v9rqoflj77i98x5/Book\\_intro.mp4?raw=1](https://www.dropbox.com/s/v9rqoflj77i98x5/Book_intro.mp4?raw=1)

## Prerequisites

Prior experience with single-variable calculus (basic differentiation and integration) and with base Python are expected.

# Resources

## Useful guides

- [pandas user guide](#), [pandas cheat sheet](#), [pandas long summary](#)
- [seaborn tutorial](#), [seaborn cheat sheet](#)
- [scikit-learn user guide](#), [sklearn cheat sheet](#)
- [numpy cheat sheet](#)

## Data sources

Here are places around the web with data available for download.

### Search engines

These point to a lot of other resources.

- [Google Dataset Search](#)
- [Registry of Open Data on AWS](#) Access to datasets used by governments and researchers that happen to be stored on Amazon's servers. Skewed toward large datasets.

### Packaged

These feature datasets that are essentially already packaged as CSV or Excel files, plus descriptions.

- [Five Thirty-Eight](#) Data used to support the site's journalism, mainly in politics and sports.
- [Delaware Open Data](#) Publicly available data from the state government.
- [Kaggle](#) Long-time host of data science competitions. The formal competitions are well-curated, but user contributions vary widely.
- [UCI Machine Learning Repository](#) Well-known source for datasets that have been used extensively in machine learning research, but also recent contributions.
- [Open ML](#) Sort of abandoned years ago, but lots of eclectic datasets remain.

- [IMDB Datasets](#) Information about movies and TVs. (Big files!)
- [Stanford Network Analysis Project](#) Datasets presented as networks.

## Open-ended

These require you to navigate an interface to select data from a large pool. Typically, you can make selections, preview the dataset, and then download in CSV or Excel format.

- [U.S. Census Bureau](#) Tons of demographic data about the U.S.
- [Data.gov](#) Home for all open U.S. government data.
- [UNICEF Portal](#) Worldwide data about child welfare.
- [World Bank](#) Focuses on economic and development data.
- [World Health Organization](#) Information on health and disease.

## Glossary

A much more exhaustive glossary can be found [here](#).

## Git

- **Git** Protocol for maintaining the entire file history of a project, including all versions and author attributions.
- **repository** Collection of files needed to record the history of a git project.
- [GitHub](#) Website that hosts git repositories created by private users, along with tools to help inspect and manage them.
- **commit** Collection of particular changes to the repository made by an individual and given a message.
- **stage** Temporary designation of locally modified files to be added to the next commit.
- **merge** Automatic union of non-conflicting commits from different sources.
- **conflict** Disagreement between repository versions that requires human intervention to resolve.
- **push** Sending one or more commits from a local repository to a remote repository.
- **pull** Receiving and merging all commits from a remote repository that are unknown to the local repository.

## Notebooks

- **notebook** Self-contained collection of text, math, code, output, and graphics.
- **kernel** Back-end that executes code from and returns output to the notebook.
- **cell** Atomic unit of a notebook that contains one or more lines of text or code.

- **Markdown** Simplified syntax to put boldface, italics, and other formatting within text.
- **TeX/LaTeX** Language used to express mathematical notation within a notebook.
- **Jupyter** Popular format and system for interactive editing, execution, and export of notebooks.
- **Jupyter Lab** Layer over Jupyter notebook functionality to help manage notebooks and extensions.

## Python

- **package** (or wheel) Collection of Python files distributed in a portable way to provide extra functionality.
- **numpy** Package of essential tools for numerical computation.
- **scipy** Package of tools useful in scientific and engineering computation.
- **database** Structured collection of data, usually with a formal interface for interaction with the data.
- **data frame** Tabular representation of a data set analogous to a spreadsheet, in which columns are observable quantities and rows are different observations of the quantities.
- **pandas** Package for working with data frames.
- **matplotlib** Package providing plot capabilities, modeled on MATLAB.
- **seaborn** Package built on matplotlib and providing commands to simplify creating plots from data frames.
- **scikit-learn** Package that provides capabilities for machine learning using a variety of methods.
- **tensorflow, keras, pytorch** Best-known packages for machine learning using neural networks.
- **Anaconda** Bundle of Python, most of the popular Python packages, Jupyter, and other useful tools, all within an optional point-and-click interface.

## Editors/IDEs

- **VS Code** (*recommended*) Free all-purpose editor with many extensions for working closely with git, Github, Jupyter, and Python.
- **Jupyter** Popular format and system for interactive editing, execution, and export of notebooks.
- **Jupyter Lab** Layer over Jupyter notebook functionality to help manage notebooks and extensions.
- **Google Colab** Free cloud-based service for jumping into Jupyter notebooks without installing any software locally.
- **Spyder** Free development environment that somewhat resembles MATLAB.
- **PyCharm** Feature-rich freemium development environment for Python, geared toward large, complex projects.

- **Thonny** Bare-bones development environment intended to prioritize beginners.

# Getting started

We will be using Python extensively. There are choices for two major aspects of using Python:

- How the code runs (the *back end*), and
- How you create and edit code and use the output (the *front end*).

## Back end

Here is a rundown of some pros and cons of the usual options:

Your own machine	Cloud
Available offline	Use any device
Total control /	No installations
Choose the front end	Browser only
Yours forever	Permanence is an illusion
Your hardware	You get what you get ^_^( )_/-

### 💡 Tip

You want to use Python 3.x, not Python 2.x. These coexisted for a while, during a dark time for Python. If you see guidance based on Python 2, it is either outdated or the product of a disordered mind.

## Run on your own computer

The recommended option is to download [Anaconda](#). It's a big download, but it comes with just about everything ready to go, and you can manage things by point-and-click.

## Run on the cloud

There are many choices, but a popular one is [Google Colab](#). It's free and saves you from having to install anything. The hardware is basic but most likely fine for our purposes. You will need to download your results for submissions.

## Front end

Much of data science is expressed using *notebooks*, which we will use exclusively. Specifically, you will use *Jupyter* notebooks.

### 💡 Tip

The name *IPython* refers to the direct ancestor of Jupyter. The older name continues to stick to a few of the tools, though.

A notebook is a self-contained collection of text, math, code, output, and graphics grouped into *cells*. The front end manages the cells and communicates with a back end called the *kernel*.

If you are using Colab, then you are using a Jupyter notebook, though in an interface customized by Google.

## On your own machine

Jupyter splits into “classic” Jupyter and the newer Jupyter Lab. Either is fine for us, but there is no reason to prefer the older variant. Just start it up from the Anaconda dashboard, and it will open your web browser to the front-end server.

A worthwhile alternative is to edit the notebook within [VS Code](#), which is a powerful and popular editor for Python and other languages.

## Tips on Jupyter success

### ⚠️ Warning

The order of cells that you see in a notebook is not necessarily the order in which they were executed.

*By far* the greatest source of confusion and subtle problems in a notebook is the freedom it gives you to execute the cells in whatever order you want. As you experiment and add and delete cells to try things out, you will reach a point at which the code on the screen is no longer a recipe for reaching the current state of your workspace.

### 💡 Tip

Before submitting a notebook, it’s *highly advisable* to restart the kernel and run all cells in order, just to make sure that everything still works as seen on screen.

Make use of the code completion tools. If you start typing the name of a variable, data column, or python method, you can either pause or hit TAB to see a list of possible completions. This can (a) save you typing time and (b) remind you of function names that are on the tip of your tongue.

# 1 Representation of data

First we have to discuss how to represent data, both abstractly and in Python.

## 1.1 Quantitative data

**Definition 1.1.** A **quantitative** value is one that is numerical and supports meaningful comparison and arithmetic operations.

Quantitative data is further divided into **continuous** and **discrete** types. The difference is the same as between real numbers and integers.

**Example 1.1.** Some continuous quantitative data sources:

- Temperature at noon at a given airport
- Your height
- Voltage across the terminals of a battery

Examples of discrete quantitative data:

- The number of shoes you own
- Number of people at a restaurant table
- Score on an exam

In each case, it makes sense, for example, to order different values and compute averages of them. However, averages of discrete quantities are continuous.

### Note

Sometimes there may be room for interpretation or context. For example, the retail price of a gallon of milk might be regarded as discrete data, since it technically represents a whole number of pennies. But in finance, transactions are regularly computed to much higher precision, so it might make more sense to interpret prices as continuous values.

[https://www.  
dropbox.com/  
s/  
l9fs3fa43hup08l/  
Example1\\_1.  
mp4?raw=1](https://www.dropbox.com/s/l9fs3fa43hup08l/Example1_1.mp4?raw=1)

**Example 1.2.** Not all numerical values represent truly quantitative data. ZIP codes (postal codes) in the U.S. are 5-digit numbers, and while there is some logic to how they were assigned, there is no clearly meaningful interpretation of averaging them, for instance.

Mathematically, the real and integer number sets are infinite, but that is not possible in a computer. Integers are represented exactly within some range that is determined by how many binary bits are dedicated. The computational analog of real numbers are **floating-point numbers**, or more simply, **floats**. These are bounded in range as well as discretized. The details are complicated, but essentially, the floating-point numbers have about 16 significant digits by default, which is virtually always far more precision than real data offers.

```
# This is an integer (int type)
print(3, "is a", type(3))

# This is a real number (float type)
print(3.0, "is a", type(3.0))

# Convert int to float (generally no change to numerical value)
print( "float(3) creates", float(3) )

# Truncate float to int
print( "int(3.14) creates", int(3.14) )

3 is a <class 'int'>
3.0 is a <class 'float'>
float(3) creates 3.0
int(3.14) creates 3
```

### 1.1.1 Special values

There are two additional quasi-numerical **float** values to be aware of as well.

**!** Important

For numerical work in Python, the NumPy package indispensable. We will use it often, and it is also loaded and used by most other scientifically oriented packages.

The value **inf** stands for infinity. It's greater than every finite number. Some arithmetic with infinity is well-defined:

```

import numpy as np
print( "np.inf + 5 is", np.inf + 5 )
print( "np.inf + np.inf is", np.inf + np.inf )
print( "5 - np.inf is", 5 - np.inf )

np.inf + 5 is inf
np.inf + np.inf is inf
5 - np.inf is -inf

```

However, in calculus you learned that some expressions with infinity are considered to be undefined without additional information to apply (e.g., L'Hôpital's Rule):

```

print( "np.inf / np.inf is", np.inf / np.inf )

np.inf / np.inf is nan

```

The result `nan` stands for *Not a Number*. It is the result of indeterminate arithmetic operations, like  $\infty/\infty$  and  $\infty - \infty$ . It is also used sometimes as a placeholder for missing data, i.e. to mean, “unknown value”.

### Warning

By definition, every operation that involves a `NaN` value results in a `NaN`.

One notorious consequence of this behavior is that `nan==nan` is `nan`, not `True`!

### 1.1.2 Dates and times

Handling times and dates can be tricky. Aside from headaches such as time zones and leap years, there are many different ways people and machine represent dates, and with varying amounts of precision. Python has its own inbuilt system for handling dates and times, but we will show the facilities provided by NumPy instead.

There are two basic types:

**Definition 1.2.** A **datetime** is a representation of an instant in time. A **time delta** is a representation of a duration; i.e., a difference between two datetimes.

### Important

Native Python uses a `datetime` type, while NumPy uses `datetime64`.

```
np.datetime64("2020-01-17")      # YYYY-MM-DD

numpy.datetime64('2020-01-17')

np.datetime64("1969-07-20T20:17")    # YYYY-MM-DDThh:mm

numpy.datetime64('1969-07-20T20:17')

# Current date and time, down to the second
np.datetime64("now")

numpy.datetime64('2023-01-30T17:15:07')
```

A time delta in NumPy indicates its units (granularity).

```
np.datetime64("1969-07-20T20:17") - np.datetime64("today")

numpy.timedelta64(-28153663,'m')
```

### 1.1.3 Random numbers

Generating truly random numbers on a computer is not simple. Mostly we rely on *pseudorandom* numbers, which are generated by deterministic functions called **random number generators** (RNGs) that have extremely long periods. One nice consequence is repeatability. By specifying the starting state of the RNG, you can get exactly the same pseudorandom sequence every time.

We will rely on pseudorandom numbers in two ways. First, many algorithms in data science have at least one random aspect (dividing data into subsets, for example). The library routines we will be using allow you to specify the random state and get repeatable results. Occasionally, though, we might want to generate random values for our own use.

```
from numpy.random import default_rng
rng = default_rng(19716)      # giving an initial state
```

The `uniform` generator method produces numbers distributed uniformly (i.e., every value is equally likely) between two limits you specify.

```
for _ in range(5):
    print( rng.uniform( -1, 1 ) )
```

```
0.9516875346510687
0.3153947867339544
-0.6651579991995873
0.42925720152795055
0.960762541480505
```

Another common type of random value is generated by `normal`, which produces real values distributed according to the normal or Gaussian distribution, which we'll get into with more detail later.

```
for _ in range(5):
    print( rng.normal() )
```

```
-0.6241028855083841
-0.2829164875756926
-1.2277902883810283
-0.4642829516161333
0.602421860754439
```

**Example 1.3.** In the long run, the average value of normally distributed numbers will be zero. Here is an experiment on 100,000 of them:

```
s = 0
for _ in range(100000):
    s += rng.normal()
```

```
s/100000
```

```
-0.00019715894764449414
```

## 1.2 Arrays

Most interesting phenomena are characterized and influenced by more than one factor. Collections of values therefore play a huge role in data science. The workhorse type for collections in base Python is the list. However, we're going to need some more powerful metaphors and tools as well.

### 1.2.1 Vectors

**Definition 1.3.** A **vector** is a collection of values called **elements**, all of the same type, indexed by consecutive integers.

! Important

In math, vector indexes usually begin with 1. In Python, they begin with 0.

A vector with  $n$  elements is often referred to as an  $n$ -vector, and we say that  $n$  is the **length** of the vector. In math we often use  $\mathbb{R}^n$  to denote the set of all  $n$ -vectors with real-valued elements.

The usual way to work with arrays in Python is through NumPy:

```
import numpy as np

x = np.array( [1,2,3,4,5] )
x

array([1, 2, 3, 4, 5])
```

A vector has a data type for its elements:

```
x.dtype

dtype('int64')
```

[https://www.  
dropbox.com/  
s/  
eq4n2508uhtwjir/  
Section1\\_2\\_  
a.mp4?raw=1](https://www.dropbox.com/s/eq4n2508uhtwjir/Section1_2_a.mp4?raw=1)

Any float values in the vector cause the data type of the entire vector to be **float**:

```
y = np.array( [1.0,2,3,4,5] )
y.dtype

dtype('float64')
```

Use **len** to determine the length of a vector:

```
len(x)
```

5

You can create a special type of vector called a *range* that has equally spaced elements:

```
np.arange(0,5)  
  
array([0, 1, 2, 3, 4])  
  
np.arange(1,3,0.5)  
  
array([1. , 1.5, 2. , 2.5])
```

The syntax here is `(start,stop,step)`. Note something critical and counterintuitive from the above results:



In base Python and in NumPy, the last element of a range is omitted. This is guaranteed to cause confusion if you are used to just about any other computer language.

### 1.2.1.1 Access and slicing

Use square brackets to refer to an element of a vector:

```
x[0]  
  
1  
  
x[4]  
  
5
```

[https://www.  
dropbox.com/  
s/  
adslly9ncce8ivx/  
Section1\\_2\\_  
b.mp4?raw=  
1](https://www.dropbox.com/s/adslly9ncce8ivx/Section1_2_b.mp4?raw=1)

Negative values for the index are counted from the end. The last element of a vector always has index `-1`, and more-negative values move backward through the elements:

```
x[-1]  
  
5  
  
x[-3]  
  
3
```

```
x[-len(x)]
```

```
1
```

Element references can also be on the left side of an assignment:

```
x[2] = -3
x
```

```
array([ 1,  2, -3,  4,  5])
```

Note, however, that once the data type of a vector is set, it can't be changed:

```
x[0] = 1.234
x      # float was truncated to int, without warning!
```

```
array([ 1,  2, -3,  4,  5])
```

You can also use a list in square brackets to access multiple elements at once:

```
x[ [0,2,4] ]
```

```
array([ 1, -3,  5])
```

Accessing elements via a range is known as **slicing**:

```
x[0:3]
```

```
array([ 1,  2, -3])
```

```
x[-3:-1]
```

```
array([-3,  4])
```

As with ranges, the syntax of a slice is `start:stop:step`, and the last element of the range is *not* included. This causes headaches and bugs, though it does imply that the range  $i:j$  has  $j - i$  elements, not  $j - i + 1$ .

When the `start` of the range is omitted, it means “from the beginning”, and when `stop` is omitted, it means “through to the end.” Hence, `[:k]` means “first  $k$  elements” and `[-k:]` means “last  $k$  elements”:

```
x[:3]  
array([ 1,  2, -3])
```

```
x[-3:]  
array([-3,  4,  5])
```

And we also have this idiom:

```
x[::-1]    # reverse the vector  
array([ 5,  4, -3,  2,  1])
```

NumPy will happily allow you to reference invalid indexes. It will just return as much as is available without warning or error. :::

```
x[:10]  
array([ 1,  2, -3,  4,  5])
```

### 1.2.2 Multiple dimensions

A vector is an important special case of a more general construct.

**Definition 1.4.** An **array** is a collection of values called **elements**, all of the same type, indexed by one or more sets of consecutive integers. The number of indexes needed to specify a value is the **dimension** of the array.

**i** Note

A dimension is called an **axis** in NumPy and related packages.

**i** Note

The term **matrix** is often used simply to mean a 2D array. Technically, though, a matrix should have only numerical values, and matrices obey certain properties that

make them important mathematical objects. These properties and their consequences are studied in linear algebra.

### 1.2.2.1 Construction

One way to construct an array is by a list comprehension:

```
A = np.array([ [j-i for j in range(6)] for i in range(4) ])
A

array([[ 0,  1,  2,  3,  4,  5],
       [-1,  0,  1,  2,  3,  4],
       [-2, -1,  0,  1,  2,  3],
       [-3, -2, -1,  0,  1,  2]])
```

[https://www.  
dropbox.com/  
s/  
xk5nyx52miba267/  
Section1\\_2\\_  
c.mp4?raw=1](https://www.dropbox.com/s/xk5nyx52miba267/Section1_2_c.mp4?raw=1)

The `shape` of an array is what we would often call the *size*:

```
A.shape
```

```
(4, 6)
```

There is no difference between a vector and a 1D array:

```
x.shape
```

```
(5,)
```

There is also no difference between a 2D array and a vector of vectors that give the rows of the array.

```
R = np.array( [ [1,2,3], [4,5,6] ] )
R

array([[1, 2, 3],
       [4, 5, 6]])
```

Here are some other common ways to construct arrays.

```

np.ones(5)

array([1., 1., 1., 1., 1.])

np.zeros( (3,6) )

array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])  

# See earlier section for definition of rng
rng.normal( size=(3,4) )

array([-1.27264664, -0.62756222, -0.30866094,  0.02293842],
      [ 0.29444108, -0.80103727,  0.34504746, -0.11060255],
      [-1.72285615,  0.22140352, -1.06020551, -0.05159689]])  

np.repeat(np.pi, 3)

array([3.14159265, 3.14159265, 3.14159265])

```

You can also stack arrays vertically or horizontally to create new arrays.

```

np.hstack( ( np.ones((2,2)), np.zeros((2,3)) ) )

array([[1., 0., 0., 0.],
       [1., 0., 0., 0.]])  

np.vstack( (range(5), range(5,0,-1)) )

array([[0, 1, 2, 3, 4],
       [5, 4, 3, 2, 1]])

```

### 1.2.2.2 Indexing and slicing

We can use successive brackets to refer to an element, row then column:

[https://www.  
dropbox.com/  
s/  
b8187f40a2vghjv/  
Section1\\_2  
d.mp4?raw=  
1](https://www.dropbox.com/s/b8187f40a2vghjv/Section1_2.d.mp4?raw=1)

```
R[1] [2]      # second row, third column
```

6

But it's more convenient to use a single bracket set with indexes separated by commas:

```
R[1, 2]      # second row, third column
```

6

You can slice in each dimension individually.

```
R[:1, -2:]    # first row, last two columns
```

```
array([[2, 3]])
```

The result above is another 2D array. Note how this result is subtly different:

```
R[0, -2:]
```

```
array([2, 3])
```

Because we accessed an individual row, not a slice, the result is one dimension lower—a vector. Finally, a `:` in one slice position means to keep everything in that dimension.

```
A[:, :2]      # all rows, first 2 columns
```

```
array([[ 0,  1],
       [-1,  0],
       [-2, -1],
       [-3, -2]])
```

### 1.2.2.3 Reductions

A common task is to *reduce* an array along one dimension, called an *axis* in numpy, resulting in an array of one less dimension. It's easiest to explain by some examples.

[https://www.dropbox.com/s/nvt91p4okytb33e/Section1\\_2\\_e.mp4?raw=1](https://www.dropbox.com/s/nvt91p4okytb33e/Section1_2_e.mp4?raw=1)

```
np.sum(A, axis=0)      # sum along the rows  
  
array([-6, -2,  2,  6, 10, 14])  
  
np.sum(A, axis=1)      # sum along the columns  
  
array([15,  9,  3, -3])
```

If you don't give an axis, the reduction occurs over all directions at once, resulting in a single number.

```
np.sum(A)
```

24

You can also do reductions with maximum, minimum, mean, etc.

## 1.3 Qualitative data

A qualitative value is one that is not quantitative. However, in order to work with such data, we usually have to encode it in some numerical form,

### 1.3.1 Categorical

**Definition 1.5.** **Categorical** data has values drawn from a finite set  $S$  of categories. If the members of  $S$  support meaningful ordering comparisons, then the data is **ordinal**; otherwise, it is **nominal**.

**Example 1.4.** Examples of ordinal categorical data:

- Seat classes on a commercial airplane (e.g., economy, business, first)
- Letters of the alphabet

Examples of nominal categorical data:

- Yes/No responses
- Marital status
- Make of a car

There are nuanced cases. For instance, letter grades are themselves ordinal categorical data. However, schools convert them to discrete quantitative data and then compute a continuous quantitative GPA.

One way to quantify ordinal categorical data is to assign integer values to the categories in a manner that preserves ordering. This approach can succeed, but it can be questionable when it comes to operations such as averaging or computing a distance between values.

Another means of quantifying categorical data is called **dummy variables** in classical statistics and **one-hot encoding** in much of machine learning. Suppose a variable  $x$  has values in a category set that has  $m$  members we label  $c_1, \dots, c_m$ . Then we can replace  $x$  with introduce  $m - 1$  new variables  $x_1, \dots, x_{m-1}$ , where

$$x_i = \begin{cases} 1, & x = c_i, \\ 0, & x \neq c_i. \end{cases}$$

At most one of the  $x_i$  can be 1. If all of the  $x_i$  are 0, then we know that  $x = c_m$ .

**Example 1.5.** Suppose that the *stooge* variable can take the values Moe, Larry, Curly, or Shemp. Then the vector

[Curly, Moe, Curly, Shemp]

would be replaced by the array

[ [0,0,1], [1,0,0], [0,0,1], [0,0,0] ]

### i Note

Sometimes an  $m$ -fold categorical variable is replaced by  $m$  indicator (dummy) variables, rather than just  $m - 1$ .

## 1.3.2 Text

Text is a ubiquitous data source. One way to quantify text is to use a dictionary of interesting keywords  $w_1, \dots, w_n$ . Given a collection of documents  $d_1, \dots, d_m$ , we can define an  $m \times n$  **document-term matrix**  $T$  by letting  $T_{ij}$  be the number of times term  $j$  appears in document  $i$ .

### 1.3.3 Images

The most straightforward way to represent an image is as a 3D array of values representing intensities representing of red, green and blue in each pixel. Sometimes it might be preferable to represent the image by a vector of statistics about these values, or by presence or absence of detected objects, etc.

## 1.4 Series and frames

The most popular Python package for manipulating and analyzing data is [pandas](#). We will use the paradigm it presents, which is fairly well understood throughout data science.

**Definition 1.6.** A **series** is a vector that is indexed by a finite ordered set. A **data frame** is a collection of series that all share the same index set.

We can conceptualize a series as a vector plus an index list, and a data frame as a 2D array with index sets for the rows and the columns. In that sense, they are simply syntactic sugar. However, since people are much better at remembering the meaning of words than arbitrarily assigned integers, data frames serve to prevent errors and misunderstandings.

**Example 1.6.** Some data that can be viewed as series:

- The length, width, and height of a box can be expressed as a 3-vector of positive real numbers. If we index the vector by the names of the measurements, it becomes a series.
- The number of steps taken by an individual over the course of a week can be expressed as a 7-vector of nonnegative integers. We could index it by the integers 1–7, or in a series by the names of the days of the week.
- The bid prices of a stock at the end of each trading day can be represented as a *time series*, in which the index is drawn from timestamps.
- The scores of gymnasts on multiple apparatus types can be represented as a data frame whose rows are indexed by the names of the gymnasts and whole columns are indexed by the names of the apparatuses.

**Example 1.7.** Here is a pandas series for the wavelengths of light corresponding to rainbow colors:

```
import pandas as pd

wavelength = pd.Series(
    [400, 470, 520, 580, 610, 710],      # values
    index=["violet", "blue", "green", "yellow", "orange", "red"],
```

```
    name="wavelength"
)
```

```
wavelength
```

wavelength	
violet	400
blue	470
green	520
yellow	580
orange	610
red	710

We can use an index value (one of the colors) to access a value in the series:

```
wavelength["blue"]
```

```
470
```

If we access multiple values, we get a series that is a subset of the original:

```
wavelength[ ["violet", "red"] ]
```

wavelength	
violet	400
red	710

We can also use the `iloc` property to access the underlying vector by NumPy slicing:

```
wavelength.iloc[:4]
```

wavelength	
violet	400
blue	470
green	520
yellow	580

Here is a series of NFL teams based on the same index:

```
team = pd.Series(
    ["Vikings", "Bills", "Eagles", "Chargers", "Bengals", "Cardinals"],
```

```

    index = wavelength.index,
    name="team"
)

team["green"]

'Eagles'

```

Now we can create a data frame using these two series as columns:

```

rainbow = pd.DataFrame( {"wavelength": wavelength, "team name": team} )
rainbow

```

	wavelength	team name
violet	400	Vikings
blue	470	Bills
green	520	Eagles
yellow	580	Chargers
orange	610	Bengals
red	710	Cardinals

### 💡 Tip

Curly braces { } are used to construct a dictionary in Python.

We can access a single column using simple bracket notation:

```
rainbow["team name"]
```

	team name
violet	Vikings
blue	Bills
green	Eagles
yellow	Chargers
orange	Bengals
red	Cardinals

We can add a column after the fact by using a bracket access on the left side of the assignment:

```

rainbow["flower"] = [
    "Lobelia",
    "Cornflower",
    "Bells-of-Ireland",
    "Daffodil",
    "Butterfly weed",
    "Rose"
]

```

```
rainbow
```

	wavelength	team name	flower
violet	400	Vikings	Lobelia
blue	470	Bills	Cornflower
green	520	Eagles	Bells-of-Ireland
yellow	580	Chargers	Daffodil
orange	610	Bengals	Butterfly weed
red	710	Cardinals	Rose

We can access a row by using brackets with the `loc` property of the frame, getting a series indexed by the column names of the frame:

```
rainbow.loc["orange"]
```

	orange
wavelength	610
team name	Bengals
flower	Butterfly weed

We are also free to strip away the index and get an ordinary array:

```
rainbow.loc["red"].to_numpy()
```

```
array([710, 'Cardinals', 'Rose'], dtype=object)
```

Here is another way to construct a data frame in pandas. We give a list of rows, plus (optionally) the index and the names of the columns:

```

letters = pd.DataFrame(
    [ ("a", "A"), ("b", "B"), ("c", "C") ],
    columns=["lowercase", "uppercase"]
)

```

[https://www.dropbox.com/s/t74l0mltts3s9vz/Section1\\_4\\_a.mp4?raw=1](https://www.dropbox.com/s/t74l0mltts3s9vz/Section1_4_a.mp4?raw=1)

```
)
```

```
letters
```

	lowercase	uppercase
0	a	A
1	b	B
2	c	C

When we specify data by column, the columns can be just lists or vectors, not necessarily series:

```
letters = pd.DataFrame(  
    { "lowercase": ["a","b","c"], "uppercase": ["A","B","C"] },  
)
```

```
letters
```

	lowercase	uppercase
0	a	A
1	b	B
2	c	C

Pandas has facilities for dealing with categorical variables.

**Example 1.8.** Here is a vector of chess pieces at the start of a game:

```
pieces = np.hstack( [  
    np.repeat("pawn", 8),  
    np.repeat(["knight", "bishop", "rook"], 2),  
    "queen",  
    "king"  
] )  
  
pieces  
  
array(['pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn',  
       'knight', 'knight', 'bishop', 'bishop', 'rook', 'rook', 'queen',  
       'king'], dtype='<U6')
```

We can tell pandas to regard these strings as elements of a category set:

```

pd.Categorical(pieces)

['pawn', 'pawn', 'pawn', 'pawn', 'pawn', ..., 'bishop', 'rook', 'rook', 'queen', 'king']
Length: 16
Categories (6, object): ['bishop', 'king', 'knight', 'pawn', 'queen', 'rook']

```

Notice above that the unique categories were found automatically.

### 1.4.1 Common operations

Table 1.1: Operations on pandas series

Description	Syntax	Result
First or last entries	s.head(), s.tail()	Series
Length	len(s)	integer
Number of entries	s.shape	tuple
All of the values	s.values	array
Convert to list	list(s)	list
Index	s.index	Index
Unique values	s.unique()	array
Appearance counts for values	s.value_counts()	Series
Extreme values	s.min(), s.max()	number
Sum	s.sum()	number
Comparison	s > 1, s=="foo"	boolean Series
Locate missing	s.isna(), s.isnull()	boolean Series
Arithmetic	s + 1, s * t	Series
Delete one or more rows	s.drop(0)	Series

Pandas offers many methods for manipulating series and dataframes. Table 1.1 shows common operations that can be applied to a series. Since a column of a dataframe is a series, these operations can be applied in that context, too. There is an [exhaustive list](#) in the pandas documentation.



#### Caution

As always in Python, you need to pay attention to the difference between applying a function, like `foo(bar)`, accessing a property, like `foo.bar`, and calling an object method, `foo.bar()`. Extra or missing parentheses groups can cause errors.

Table 1.2: Operations on pandas dataframes

Description	Syntax	Result
First or last rows	<code>df.head()</code> , <code>df.tail()</code>	DataFrame
Number of rows	<code>len(df)</code>	integer
Number of rows and columns	<code>df.shape</code>	tuple
All of the values	<code>df.values</code>	array
Row index	<code>df.index</code>	Index
Column names	<code>list(df)</code>	list
Column names	<code>df.columns</code>	Index
Access by column name(s)	<code>df["name"]</code>	Series or DataFrame
Access by name	<code>df.loc[rows, "name"]</code>	(varies)
Access by position	<code>df.iloc[i, j]</code>	(varies)
Sum	<code>df.sum()</code>	Series
Comparison	<code>s &gt; 1, s=="foo"</code>	boolean DataFrame
Delete a column	<code>df.drop(name, axis=1)</code>	DataFrame
Apply columnwise	<code>df.apply(myfun)</code>	(varies)

Table 1.2 shows additional operations that can be applied to an entire dataframe (or to any subset). It takes a lot of space to demonstrate all the ways in which these can be used, as is done in the [pandas user guide](#), so they are just collected here for future reference. There is also an [exhaustive list](#) of them in the pandas documentation.

### 1.4.2 Loading from files

Datasets can be presented in many forms. In this course, we assume that they are given as spreadsheets or in *comma-separated value* (CSV) files. These can be read by pandas locally or over the web.

**Example 1.9.** The pandas function we use to load a dataset is `read_csv`. Here we use it to read a file that is available over the web:

```
ads = pd.read_csv("https://raw.githubusercontent.com/tobydriscoll/ds1book/master/advertisements.csv")
ads.head(6)      # show the first 6 rows
```

	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9
5	8.7	48.9	75.0	7.2

Note above that we used `head(6)` to see just the first 6 rows.

A data frame has a few properties that describe its contents:

```
ads.shape      # number of rows, number of columns
(200, 4)

ads.columns    # names of the columns
Index(['TV', 'Radio', 'Newspaper', 'Sales'], dtype='object')

ads.dtypes     # data types of the columns

```

	0
TV	float64
Radio	float64
Newspaper	float64
Sales	float64

It's also possible to import data from most other general-purpose formats you might encounter, such as Excel spreadsheets (though possibly requiring one-time installation of additional libraries). There are many functions starting with `pd.read_` showing the formats pandas understands.

[https://www.  
dropbox.com/  
s/  
1r9is5thcryuarl/  
Section1\\_4\\_  
.b.mp4?raw=  
1](https://www.dropbox.com/s/1r9is5thcryuarl/Section1_4_.b.mp4?raw=1)

### 1.4.3 Selecting rows

Above we saw that you can use the `loc` property to access a frame's row by name, or `iloc` to access by position. It's more common, though, to select them by some criteria.

**Example 1.10.** Here's a local file that contains daily weather summaries from Newark, Delaware:

```

weather = pd.read_csv("_datasets/ghcn_newark.csv")
weather.head()

```

	STATION	DATE	LATITUDE	LONGITUDE	ELEVATION	NAME
0	USC00076410	1894-04-01	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
1	USC00076410	1894-04-02	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
2	USC00076410	1894-04-03	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
3	USC00076410	1894-04-04	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
4	USC00076410	1894-04-05	39.6682	-75.74569	32.3	NEWARK AG FARM, DE

If we apply a relational operator to a column of a frame, we get a series of Boolean values:

```

isprecip = (weather["PRCP"] > 10)
isprecip.head()

```

	PRCP
0	False
1	False
2	False
3	True
4	False

This kind of series can then be used to select the rows that have value True:

```

precip = weather[isprecip]
precip.head()

```

	STATION	DATE	LATITUDE	LONGITUDE	ELEVATION	NAME
3	USC00076410	1894-04-04	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
7	USC00076410	1894-04-08	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
10	USC00076410	1894-04-11	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
11	USC00076410	1894-04-12	39.6682	-75.74569	32.3	NEWARK AG FARM, DE
21	USC00076410	1894-04-22	39.6682	-75.74569	32.3	NEWARK AG FARM, DE

You can use logical operators & (and), | (or), and ~ (not) on these Boolean series:

```

iscold = (weather["TMAX"] < 0)
dry_cold = weather[ iscold & ~isprecip ]
dry_cold.head(6)

```

	STATION	DATE	LATITUDE	LONGITUDE	ELEVATION	NAME
240	USC00076410	1894-12-28	39.6682	-75.74569	32.3	NEWARK AG FARM, D
241	USC00076410	1894-12-29	39.6682	-75.74569	32.3	NEWARK AG FARM, D
242	USC00076410	1894-12-30	39.6682	-75.74569	32.3	NEWARK AG FARM, D
243	USC00076410	1894-12-31	39.6682	-75.74569	32.3	NEWARK AG FARM, D
244	USC00076410	1895-01-01	39.6682	-75.74569	32.3	NEWARK AG FARM, D
245	USC00076410	1895-01-02	39.6682	-75.74569	32.3	NEWARK AG FARM, D

When a column of a dataset is a time stamp, it can be used as the index of the data frame, which makes some time-centric operations easier.

**Example 1.11.** Let's reload the dataset using the *DATE* column (which is second in the file) as a `datetime` index:

```
weather = pd.read_csv("_datasets/ghcn_newark.csv", index_col=1, parse_dates=True)
weather.head()
```

[https://www.  
dropbox.com/  
s/  
uwrmgpz15mm5aj9  
Example1\\_  
10.mp4?raw=1](https://www.dropbox.com/s/uwrmgpz15mm5aj9Example1_10.mp4?raw=1)

DATE	STATION	LATITUDE	LONGITUDE	ELEVATION	NAME
1894-04-01	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1894-04-02	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1894-04-03	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1894-04-04	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1894-04-05	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US

We can extract a row by referencing the date in the index with `loc`:

```
weather.loc["1979-03-28"]
```

1979-03-28	
STATION	USC00076410
LATITUDE	39.6682
LONGITUDE	-75.74569
ELEVATION	32.3
NAME	NEWARK AG FARM, DE US
PRCP	0.0
PRCP_ATTRIBUTES	,0,1700
SNOW	0.0
SNOW_ATTRIBUTES	,0
SNWD	0.0
SNWD_ATTRIBUTES	,0
TMAX	111.0
TMAX_ATTRIBUTES	,0
TMIN	-33.0
TMIN_ATTRIBUTES	,0
DAEV	NaN
DAEV_ATTRIBUTES	NaN
DAPR	NaN
DAPR_ATTRIBUTES	NaN
DAWM	NaN
DAWM_ATTRIBUTES	NaN
EVAP	NaN
EVAP_ATTRIBUTES	NaN
MDEV	NaN
MDEV_ATTRIBUTES	NaN
MDPR	NaN
MDPR_ATTRIBUTES	NaN
MDWM	NaN
MDWM_ATTRIBUTES	NaN
MNPN	NaN
MNPN_ATTRIBUTES	NaN
MXPN	NaN
MXPN_ATTRIBUTES	NaN
TOBS	100.0
TOBS_ATTRIBUTES	,0,1700
WDMV	NaN
WDMV_ATTRIBUTES	NaN
WT01	NaN
WT01_ATTRIBUTES	NaN
WT03	NaN
WT03_ATTRIBUTES	NaN
WT04	NaN
WT04_ATTRIBUTES	NaN
WT05	NaN
WT05_ATTRIBUTES	NaN
WT08	NaN
WT08_ATTRIBUTES	NaN
WT11	NaN
WT11_ATTRIBUTES	NaN

But we can also easily select all the rows from a particular month or year:

```
weather.loc["1979-03"].head()
```

DATE	STATION	LATITUDE	LONGITUDE	ELEVATION	NAME
1979-03-01	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1979-03-02	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1979-03-03	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1979-03-04	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US
1979-03-05	USC00076410	39.6682	-75.74569	32.3	NEWARK AG FARM, DE US

💡 Tip

The pandas user guide has a handy [section on selections](#).

[https://www.dropbox.com/s/h3ezvnpyn8xe5ch/Section1\\_4\\_c.mp4?raw=1](https://www.dropbox.com/s/h3ezvnpyn8xe5ch/Section1_4_c.mp4?raw=1)

💡 Tip

For practice with pandas fundamentals, try the [Kaggle course](#).

## 1.5 Data preparation

Raw data often needs to be manipulated into a useable format before algorithms can be applied. Preprocessing data so that it is suitable for machine analysis is known as **data wrangling** or **data munging**. A related process is **data cleaning**, where missing and anomalous values are removed or replaced.

### 1.5.1 Missing values

In real data sets, we often must cope with data series that have missing values. This is a common source of mistakes and confusion, especially because there is no universal practice. Sometimes zero is used to represent a missing number. It's also common to use an impossible value, such as `-999` to represent weight, to signify missing data.

Formally, the most natural way to represent missing data in Python is as `nan` or `NaN`, and pandas makes it easy to find and manipulate such values. Here is another well-known data set, this time about penguins:

[https://www.dropbox.com/s/wz9vpncyyyw2xex/Section1\\_5\\_a.mp4?raw=1](https://www.dropbox.com/s/wz9vpncyyyw2xex/Section1_5_a.mp4?raw=1)

```

import seaborn as sns
penguins = sns.load_dataset("penguins")
penguins.head()

```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female

Note above that the fourth row of the frame is missing measurements. We can discover how many such rows there are using `isna`:

```
penguins.isna().sum()
```

	0
species	0
island	0
bill_length_mm	2
bill_depth_mm	2
flipper_length_mm	2
body_mass_g	2
sex	11

Sometimes one replaces missing values with average or other representative values, a process called *imputation*. It's often prudent to simply toss them out, as follows:

```

print("original counts:")
print( penguins.count() )
penguins.dropna( inplace=True )
print()
print("after removals:")
print( penguins.count() )

```

```

original counts:
species          344
island           344
bill_length_mm   342
bill_depth_mm    342
flipper_length_mm 342
body_mass_g      342

```

```

sex           333
dtype: int64

after removals:
species        333
island          333
bill_length_mm 333
bill_depth_mm   333
flipper_length_mm 333
body_mass_g    333
sex             333
dtype: int64

```

 Tip

Operations that make changes to or retrieve subsets from a data frame work on copies of the frame. When `inplace=True` is given, though, the operation changes the original frame.

### 1.5.2 Loans example

To demonstrate algorithms in later sections, we will be using a [dataset describing loans](#) made on the crowdfunding site LendingClub. First, we load the raw data from a CSV (comma separated values) file.

 Tip

It's possible to import datasets from the Web directly into pandas. However, web sources and links change and disappear frequently, so if storing the dataset is not a problem, you may want to download your own copy before working on it.

```

import pandas as pd
loans = pd.read_csv("_datasets/loan.csv")
loans.head()

```

[https://www.dropbox.com/s/t56b2blgzn6s9a6/Section1\\_5\\_b.mp4?raw=1](https://www.dropbox.com/s/t56b2blgzn6s9a6/Section1_5_b.mp4?raw=1)

	id	member_id	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment
0	1077501	1296599	5000	5000	4975.0	36 months	10.65%	1
1	1077430	1314167	2500	2500	2500.0	60 months	15.27%	3
2	1077175	1313524	2400	2400	2400.0	36 months	15.96%	3
3	1076863	1277178	10000	10000	10000.0	36 months	13.49%	3
4	1075358	1311748	3000	3000	3000.0	60 months	12.69%	3

The `int_rate` column, which gives the interest rate on the loan, has been interpreted as strings due to the percent sign. We'll strip out those percent signs and convert them to floats.

💡 Tip

As you see below, we often end up with chains of methods separated by dots. Python works from left to right, evaluating a subexpression and then replacing it with the object for the next segment in the chain. We could write these as a sequence of separate lines having intermediate results assigned to variable names, but it's often considered better style to chain them.

```
loans["int_rate"] = loans["int_rate"].str.strip('%').astype(float)  
loans.head()
```

	id	member_id	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment
0	1077501	1296599	5000	5000	4975.0	36 months	10.65	12000
1	1077430	1314167	2500	2500	2500.0	60 months	15.27	36000
2	1077175	1313524	2400	2400	2400.0	36 months	15.96	12000
3	1076863	1277178	10000	10000	10000.0	36 months	13.49	36000
4	1075358	1311748	3000	3000	3000.0	60 months	12.69	12000

Let's add a column for the percentage of the loan request that was eventually funded. This will be a target for some of our learning methods.

```
loans["percent_funded"] = 100 * loans["funded_amnt"] / loans["loan_amnt"]  
target = ["percent_funded"]
```

We will only use a small subset of the numerical columns as features. Let's verify that there are no missing values in those columns:

```
features = [ "loan_amnt", "int_rate", "installment", "annual_inc",  
             "dti", "delinq_2yrs", "delinq_amnt" ]  
loans = loans[features + target]  
loans.isna().sum()
```

	0
loan_amnt	0
int_rate	0
installment	0
annual_inc	0
dti	0
delinq_2yrs	0
delinq_amnt	0
percent_funded	0

### i Note

Given lists of columns names (or any strings), you can use + to concatenate them into a single list.

Finally, we'll output this cleaned data frame to its own CSV file. The index row is an ID number that is meaningless to classification, so we will instruct pandas to exclude it from the new file:

```
loans.to_csv("loan_clean.csv", index=False)
```

### 1.5.3 Diamonds example

We will also be using a seaborn dataset comprising features of diamonds and their prices:

```
import seaborn as sns
diamonds = sns.load_dataset("diamonds")
diamonds.head()
```

[https://www.dropbox.com/s/fjf4gj8bj5hwvwm/Section1\\_5\\_.mp4?raw=1](https://www.dropbox.com/s/fjf4gj8bj5hwvwm/Section1_5_.mp4?raw=1)

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

As you can see above, some of the features (cut, color, clarity) have string designations. However, these do have a specific ordering in this context. So we will replace the strings with the correct ordinal values.

We start with the *cut* column:

```

cuts = ["Fair", "Good", "Very Good", "Premium", "Ideal"]
diamonds["cut"].replace(
    cuts,                      # to be replaced
    range(5),                  # replacements
    inplace=True                # change the original frame, not a copy
)

```

`diamonds.head()`

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	4	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	3	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	1	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	3	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	1	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Above, you see that the *cut* strings have been replaced by integers. Now we do the same for the other categories:

```

diamonds["clarity"].replace(
    ["I1", "SI2", "SI1", "VS2", "VS1", "VVS2", "VVS1", "IF"],
    range(8),
    inplace=True
)

diamonds["color"].replace(
    list("DEFGHIJ"),
    range(7),
    inplace=True
)

diamonds.head()

```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	4	1	1	61.5	55.0	326	3.95	3.98	2.43
1	0.21	3	1	2	59.8	61.0	326	3.89	3.84	2.31
2	0.23	1	1	4	56.9	65.0	327	4.05	4.07	2.31
3	0.29	3	5	3	62.4	58.0	334	4.20	4.23	2.63
4	0.31	1	6	1	63.3	58.0	335	4.34	4.35	2.75

We'll save this modified dataset to its own file for our future use:

```
diamonds.to_csv("diamonds.csv", index=False)
```

## Exercises

**Exercise 1.1.** For each type of data, classify it as discrete quantitative, continuous quantitative, categorical, or other.

- a) How many students are enrolled at a university
- b) Your favorite day of the week
- c) How many inches of rain fall at an airport during one day
- d) Weight of a motor vehicle
- e) Manufacturer of a motor vehicle
- f) Text of all Yelp reviews for a restaurant
- g) Star ratings from all Yelp reviews for a restaurant
- h) Size of the living area of an apartment
- i) DNA nucleotide sequence of a cell

**Exercise 1.2.** Give the length of each vector or series.

- a. Morning waking times every day for a week
- b. Number of siblings (max 12) for each student in a class of 30
- c. Position and momentum of a roller coaster car

**Exercise 1.3.** Describe a scheme for creating dummy variables for the days of the week.  
Use your scheme to encode the vector:

[Tuesday, Sunday, Friday, Tuesday, Monday]

## 2 Descriptive statistics

When confronted with a new dataset, it's crucial to get a sense of its characteristics before attempting to draw conclusions or predictions from it.

One of the fastest ways to become familiar with a data set is to visualize it. Python has many graphics packages with different niches. The most widespread is **Matplotlib**, which is fairly low-level in the sense that you must explicitly specify most aspects of how the plots will look.

We will make extensive use of **seaborn**, which is built on top of Matplotlib. It's meant to be used at a higher level, i.e., letting you describe what you want to see and making it look pretty good. (It is possible to customize seaborn plots using Matplotlib commands, but we won't need much of that.)

```
import seaborn as sns
```

There are three major plot types within seaborn:

**displot** How values of a single variable are distributed.

**catplot** How categorical values are distributed within and across categories.

**relplot** How values of two variables are related to each other.

### 2.1 Summary statistics

We will use data about car fuel efficiency for illustrations.

```
cars = sns.load_dataset("mpg")
```

The **describe** method of a data frame gives summary statistics for each column of quantitative data:

```
cars.describe()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year
count	398.000000	398.000000	398.000000	392.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	104.469388	2970.424623	15.568090	76.010050
std	7.815984	1.701004	104.269838	38.491160	846.841774	2.757689	3.697627
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000
50%	23.000000	4.000000	148.500000	93.500000	2803.500000	15.500000	76.000000
75%	29.000000	8.000000	262.000000	126.000000	3608.000000	17.175000	79.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000

We now discuss the definitions and interpretations of these values.

### 2.1.1 Mean and spread

You may already know the Big Three summary statistics:

**Definition 2.1.** Given data values  $x_1, \dots, x_n$ , their **mean** is

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i, \quad (2.1)$$

their **variance** is

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2, \quad (2.2)$$

and their **standard deviation** (STD) is  $\sigma$ , the square root of the variance.

Mean is a measurement of central tendency. Variance and STD are measures of spread or *dispersion* in the data.

[https://www.  
dropbox.com/  
s/  
enctb5vhk94b1mr/  
Section2\\_1\\_1.mp4?raw=1](https://www.dropbox.com/s/enctb5vhk94b1mr/Section2_1_1.mp4?raw=1)

**Example 2.1.** Suppose that  $x_1 = 0$ ,  $x_2 = t$ , and  $x_3 = -t$ , where  $|t| \leq 6$ . What are the minimum and maximum possible values of the standard deviation?

*Solution.* The mean is  $\mu = 0$ , hence

$$\sigma^2 = \frac{1}{3} [0^2 + t^2 + (-t)^2] = \frac{2}{3}t^2.$$

From this we conclude

$$\sigma = \sqrt{\frac{2}{3}|t|}.$$

Given that  $0 \leq |t| \leq 6$ , we see that the minimum value of  $\sigma$  is 0 and the maximum is  $2\sqrt{6}$ .

[https://www.  
dropbox.com/  
s/  
x9giiq35ix0kbwa/  
Example2\\_1\\_.  
mp4?raw=1](https://www.dropbox.com/s/x9giiq35ix0kbwa/Example2_1_.mp4?raw=1)

### i Note

Variance is in units that are the square of the data, which can be harder to interpret than STD, which has units the same as the data values.

## 2.1.2 z-scores

Given data values  $x_1, \dots, x_n$ , we can define related values known as **standardized scores** or **z-scores**:

$$z_i = \frac{x_i - \mu}{\sigma}, \quad i = 1, \dots, n.$$

The z-scores have mean zero and standard deviation equal to 1; in physical terms, they are dimensionless. That is, the results don't depend on the physical units chosen to express the data. Converting data into z-scores is referred to as **standardization**, and it helps make operations uniform across different datasets.

**Theorem 2.1.** *The z-scores have mean equal to zero and variance equal to 1.*

*Proof.* Direct calculations.

□

**Example 2.2.** Continuing with the values from Example 2.1, we assume without losing generality that  $t \geq 0$ . (Otherwise, we can just swap  $x_2$  and  $x_3$ .) Then we have the z-scores

$$z_1 = \frac{0 - 0}{2t\sqrt{6}} = 0, \quad z_2 = \frac{t - 0}{2t\sqrt{6}} = \frac{1}{2\sqrt{6}}, \quad z_3 = \frac{-t - 0}{2t\sqrt{6}} = \frac{-1}{2\sqrt{6}}.$$

These are independent of  $t$ , which just scales the original values.

We can write a little function to compute z-scores in Python:

```
def standardize(x):
    return (x - x.mean()) / x.std()

cars["mpg_z"] = standardize(cars["mpg"])
cars[["mpg", "mpg_z"]].describe()
```

	mpg	mpg_z
count	398.000000	3.980000e+02
mean	23.514573	1.071170e-16
std	7.815984	1.000000e+00
min	9.000000	-1.857037e+00
25%	17.500000	-7.695221e-01
50%	23.000000	-6.583596e-02
75%	29.000000	7.018217e-01
max	46.600000	2.953617e+00

### 🔥 Caution

Since floating-point values are rounded off, it's unlikely that a value derived from them that is meant to be zero will actually be exactly zero. Above, the mean value of about  $-10^{-15}$  should be seen as reasonable for values that have been rounded off in the 15th digit or so.

[https://www.  
dropbox.com/  
s/  
xm0zkquipg66kcy/  
Section2\\_1\\_  
2.mp4?raw=1](https://www.dropbox.com/s/xm0zkquipg66kcy/Section2_1_2.mp4?raw=1)

### 2.1.3 Populations and samples

In statistics one refers to the **population** as the entire universe of available values. Thus, the ages of adult on Earth at some instant has a particular population mean and standard deviation. However, in order to estimate those values, we can only measure a **sample** of the population directly.

When Equation 2.1 is used to compute the mean of a sample rather than a population, we change the notation a bit as a reminder:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i. \quad (2.3)$$

It can be proved that the sample mean is an accurate way to estimate the population mean, in the following precise sense. If, in a thought experiment, we could average  $\bar{x}$  over all possible samples of size  $n$ , the result would be exactly the population mean  $\mu$ . That is, we say that  $\bar{x}$  is an **unbiased estimator** for  $\mu$ .

The sample mean in turn can be used within Equation 2.2 to compute **sample variance**:

$$s_n^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2.$$

However, sample variance is more subtle than the sample mean. If  $s_n^2$  is averaged over all possible sample sets, we do *not* get the population variance  $\sigma^2$ ; hence,  $s_n^2$  is called a **biased estimator** of the population variance.

An unbiased estimator for  $\sigma^2$  is

$$s_{n-1}^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2. \quad (2.4)$$

**Example 2.3.** The values  $[1, 4, 9, 16, 25]$  have mean  $\bar{x} = 55/5 = 11$ . The sample variance is

$$\begin{aligned} s_n^2 &= \frac{(1-11)^2 + (4-11)^2 + (9-11)^2 + (16-11)^2 + (25-11)^2}{5} \\ &= \frac{374}{5} = 74.8. \end{aligned}$$

By contrast, the unbiased estimate of population variance from this sample is

$$s_{n-1}^2 = \frac{374}{4} = 93.5.$$

As you can see from the formulas and the example, the sample variance is always too large as an estimator, but the difference vanishes as the sample size  $n$  increases.

### ⚠️ Warning

Sources are not always clear about this terminology. Some use *sample variance* to mean  $s_{n-1}^2$ , not  $s_n^2$ , and many even omit the subscripts. You always have to check each source.

### 🔥 Caution

NumPy computes the biased estimator of variance by default, while pandas computes the unbiased version. Whee! Fortunately, most datasets today have large enough  $n$  to make the difference negligible.

For standard deviation, *neither*  $s_n$  nor  $s_{n-1}$  is an unbiased estimator of  $\sigma$ . There is no simple correction that works for all distributions. Our practice is to use  $s_{n-1}$ , which is what `std` computes in pandas. Thus, for instance, a **sample z-score** for  $x_i$  is

$$z_i = \frac{x_i - \bar{x}}{s_{n-1}}. \quad (2.5)$$

## 2.1.4 Median and quantiles

Mean and variance are not the most relevant summary statistics for every dataset. There are important alternatives.

**Definition 2.2.** For any  $0 < p < 1$ , the **100p-percentile** or **quantile** is the value of  $x$  such that  $p$  is the probability of observing a population value less than or equal to  $x$ .

The 50th percentile is known as the **median** of the population.

### i Note

Some sources reserve the term *quantile* for another meaning, but since pandas offers `quantile` to compute percentiles, we don't draw a distinction.

The unbiased sample median of  $x_1, \dots, x_n$  can be computed by sorting the values into  $y_1, \dots, y_n$ . If  $n$  is odd, then  $y_{(n+1)/2}$  is the sample median; otherwise, the average of  $y_{n/2}$  and  $y_{1+(n/2)}$  is the sample median.

**Example 2.4.** If the sorted values are 1, 3, 3, 4, 5, 5, 5, then  $n = 7$  and the sample median is  $y_4 = 4$ . If the sample values are 1, 3, 3, 4, 5, 5, 5, 9, then  $n = 8$  and the sample median is  $(4 + 5)/2 = 4.5$ .

Computing unbiased sample estimates of percentiles other than the median is complicated, and we won't go into the details. For large datasets, the sample values are good estimators in practice.

**Example 2.5.** Let's create a vector of 101 random numbers drawn uniformly from [0, 1]:

```
from numpy.random import default_rng
import pandas as pd

rng = default_rng(19716)
x = rng.uniform( size=(101) )
```

We can turn the vector into a pandas series and then find the 20th percentile:

```
data = pd.Series(x)
data.quantile(0.2)
```

0.16742100040020635

This is equivalent to looking at the 21st element of the sorted values:

```
x.sort()  
x[20]
```

0.16742100040020635

As we increase the length of the random vector, in theory the  $100p$ th percentile should be at  $x = p$ :

```
rng = default_rng(19716)  
data = pd.Series( rng.uniform(size=(200000)) )  
data.quantile( [0.2, 0.5, 0.8] )
```

---

0
0.2 0.201456
0.5 0.499738
0.8 0.798759

---

The 50th percentile is the same thing as the median:

```
data.median()
```

0.49973764014275895

**Definition 2.3.** The 25th, 50th, and 75th percentiles are the first, second, and third **quartiles** of the distribution. The **interquartile range** (IQR) is the difference between the 75th percentile and the 25th percentile.

[https://www.  
dropbox.com/  
s/  
9wov44dazmiyg64/  
Example2\\_5.  
mp4?raw=1](https://www.dropbox.com/s/9wov44dazmiyg64/Example2_5.mp4?raw=1)

Sometimes the definition above is extended to the *zeroth quartile*, which is the minimum sample value, and the *fourth quartile*, which is the maximum sample value.

IQR is an indication of the spread of the values. For some distributions, the median and IQR might be a good substitute for the mean and standard deviation.

**Example 2.6.** The dataframe `describe` method includes mean, standard deviation, and the quartiles:

```
rng = default_rng(19716)  
df = pd.DataFrame( {  
    "normal" : rng.normal( size=(10000,) ),
```

```

    "uniform" : rng.uniform( size=(10000,) )
}
df.describe()

```

	normal	uniform
count	10000.000000	10000.000000
mean	-0.012414	0.503456
std	0.993568	0.285247
min	-3.436924	0.000136
25%	-0.672352	0.258355
50%	-0.008374	0.507225
75%	0.659730	0.746851
max	4.044099	0.999901

It's easy to write a function to compute the IQR of a series:

```

def IQR(x):
    Q25, Q75 = x.quantile( [0.25, 0.75] )
    return Q75 - Q25

IQR(df.normal)

```

1.3320822591671932

## 2.2 Distributions

Mean and STD, or median and IQR, attempt to summarize quantitative data with a couple of numbers. At the other extreme, we can express the distribution of all values precisely using a function.

### 2.2.1 CDF

**Definition 2.4.** The **cumulative distribution function** (CDF) of a population is the function

$$F(t) = \text{fraction of the population that is } \leq t,$$

where  $t$  ranges over all possible values.

Note that by its definition,  $F$  ranges between 0 and 1 (inclusive) and is a nondecreasing function.

**Example 2.7.** If a population is  $x_i = i$  for  $i = 1, \dots, n$ , then  $F(k) = k/n$  at each  $k = 1, \dots, n$ . We could, however, also regard  $F$  as a function of a continuous variable  $t$ , in which case

$$F(t) = \frac{\lfloor t \rfloor}{n},$$

where  $\lfloor \cdot \rfloor$  is the *floor function* that rounds leftward to the nearest integer. This produces a step function that looks like stairs going up from 0 to 1.

Example 2.7 becomes interesting as a template for generalizing to infinite populations. If we take not  $x_i = i$  but  $x_i = i/n$  and then let  $n \rightarrow \infty$ , then the graph of  $F$  converges to

$$F(t) = \begin{cases} 0, & t < 0, \\ t, & 0 \leq t \leq 1, \\ 1, & t > 1. \end{cases} \quad (2.6)$$

[https://www.  
dropbox.com/  
s/  
z8di9mumdeyb4u5/  
Example2\\_7.  
mp4?raw=1](https://www.dropbox.com/s/z8di9mumdeyb4u5/Example2_7.mp4?raw=1)

While it doesn't make sense to think about a fraction of the number of values in the infinite case, we can interpret  $F(t)$  as the \*probability of observing a value less than or equal to the real number  $t$ .

**Definition 2.5.** A **uniform distribution** gives an equal probability to every value. In particular, the uniform distribution over the interval  $[0, 1]$  has the CDF given in Equation 2.6.

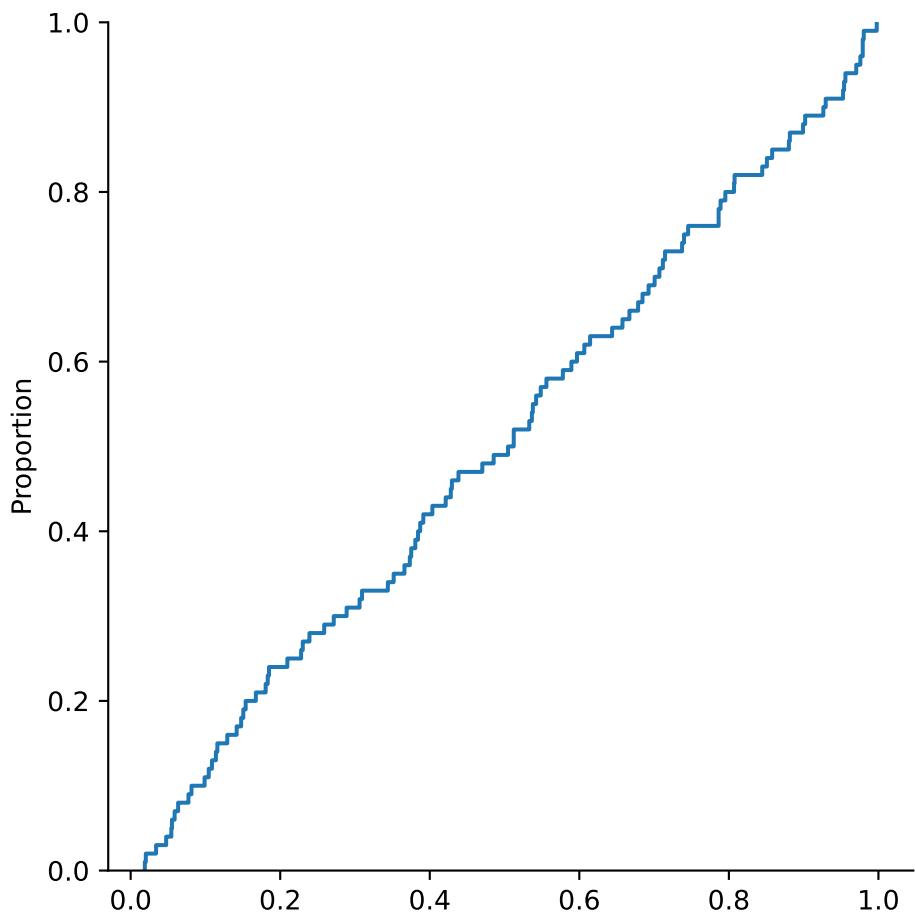
## 2.2.2 Empirical CDF

Given a sample of a population, we can always calculate the analog of a CDF from its values.

**Definition 2.6.** The **empirical cumulative distribution function** or ECDF of a sample is the function  $\hat{F}$  whose value at  $t$  equals the proportion of the sample values that are less than or equal to  $t$ .

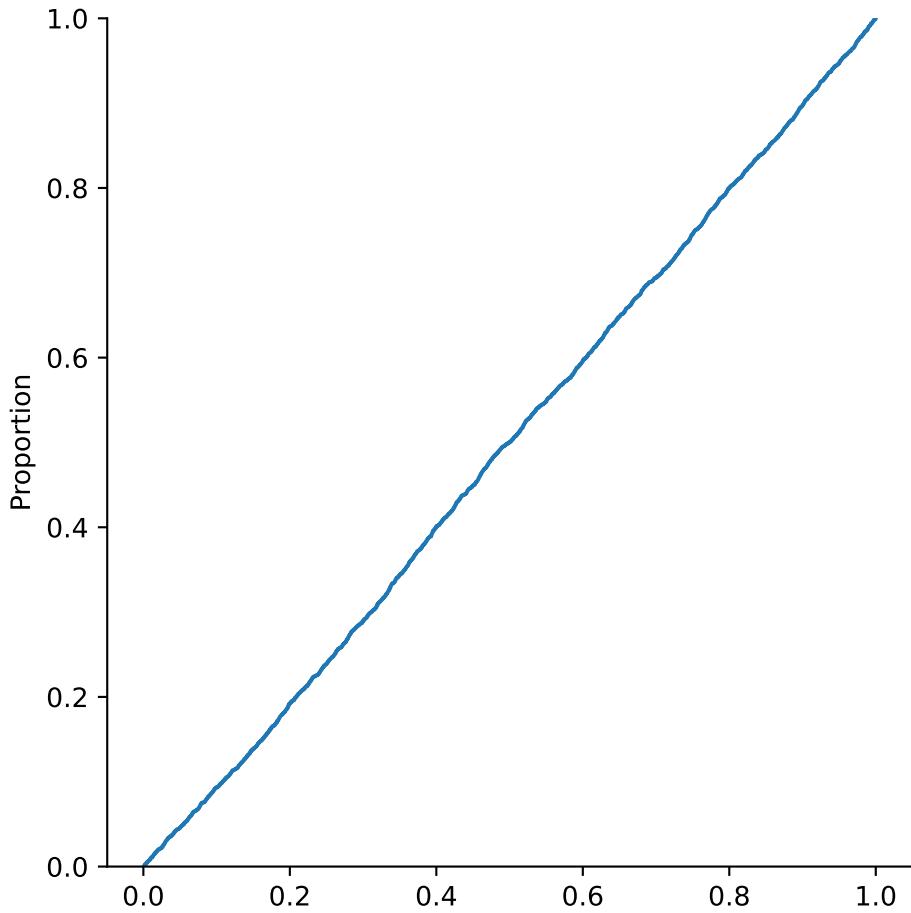
**Example 2.8.** Here is an experiment that producing the ECDF for a sample from the random number generator:

```
from numpy.random import default_rng
rng = default_rng(19716)
x = rng.uniform( size=(100,) )
sns.displot(x, kind="ecdf");
```



If we take more samples, we expect to see a curve closer to the theoretical CDF,  $F(t) = t$ :

```
x = rng.uniform( size=(4000,) )
sns.displot(x, kind="ecdf");
```



### 2.2.3 PDF

By definition, we know that if  $a < b$ ,  $\hat{F}(b) - \hat{F}(a)$  is the number of observations in the half-open interval  $(a, b]$ . This leads into the next definition.

[https://www.  
dropbox.com/  
s/  
3o8rx1k3qozldc3/  
Example2\\_8.  
mp4?raw=1](https://www.dropbox.com/s/3o8rx1k3qozldc3/Example2_8.mp4?raw=1)

**Definition 2.7.** Select the ordered values  $t_1 < t_2 < \dots < t_m$ , called **edges**, and define **bins** as the intervals

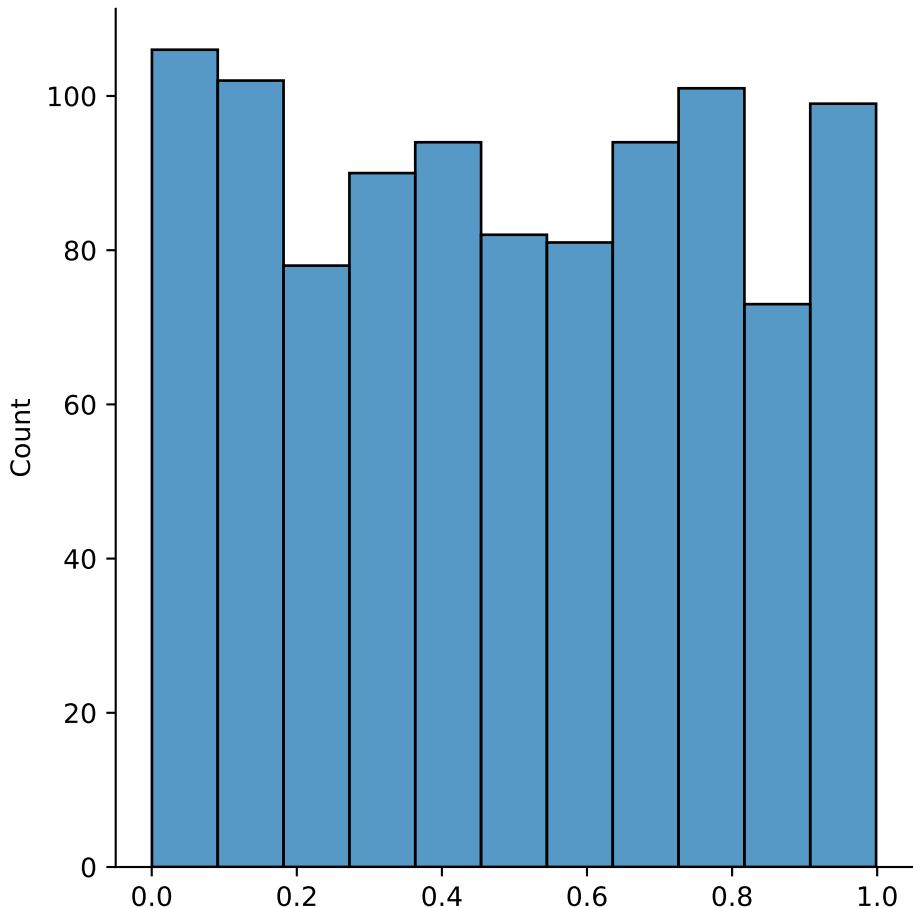
$$B_k = (t_k, t_{k+1}], \quad k = 0, \dots, m,$$

where we adopt the convention that  $t_0 = -\infty$  and  $t_{m+1} = \infty$ . Let  $c_k$  be the number of data values in  $B_k$ . Then a **histogram** relative to the bins is the list of  $(B_0, c_0), \dots, (B_m, c_m)$ .

The default for a seaborn `displot` is to show a histogram.

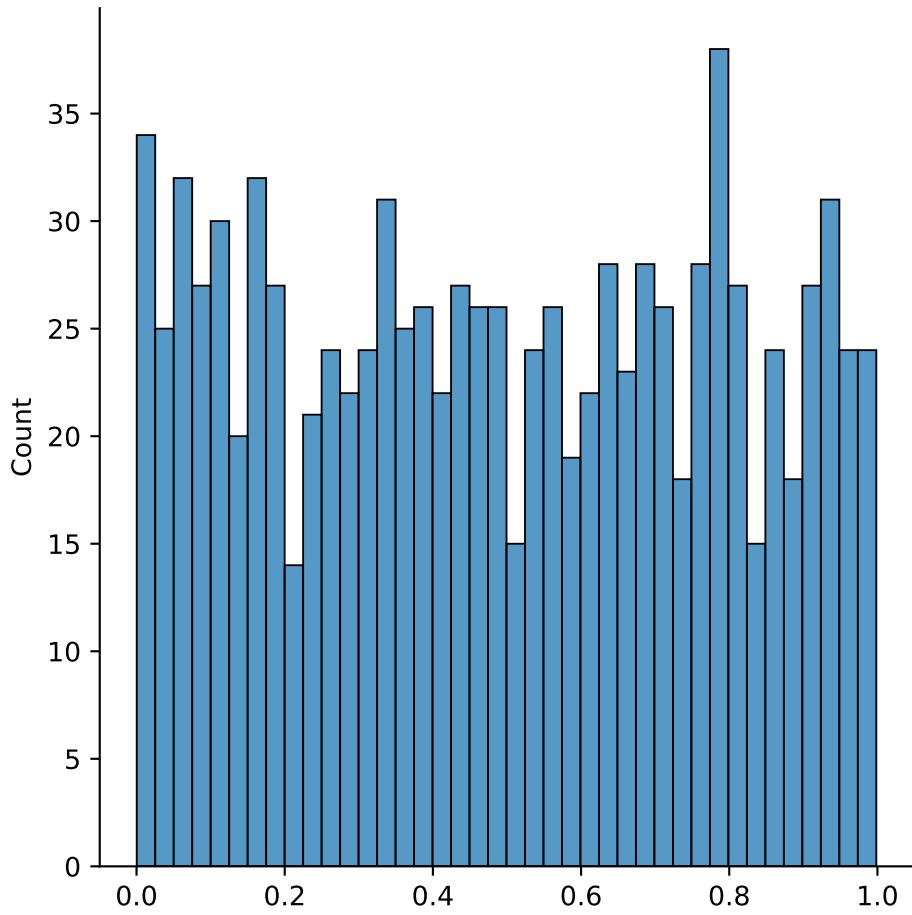
**Example 2.9.** Continuing with the uniform distribution over  $[0, 1]$ :

```
x = rng.uniform( size=(1000,) )
sns.displot(x);
```



We can choose the number of bins to use, or give a vector of their edges:

```
sns.displot(x, bins=40);
```



Again something interesting happens in a limiting case. If we normalize the count in a bin by the length of that bin, we get

$$\frac{c_k}{t_{k+1} - t_k} = \frac{\hat{F}(t_{k+1}) - \hat{F}(t_k)}{t_{k+1} - t_k}. \quad (2.7)$$

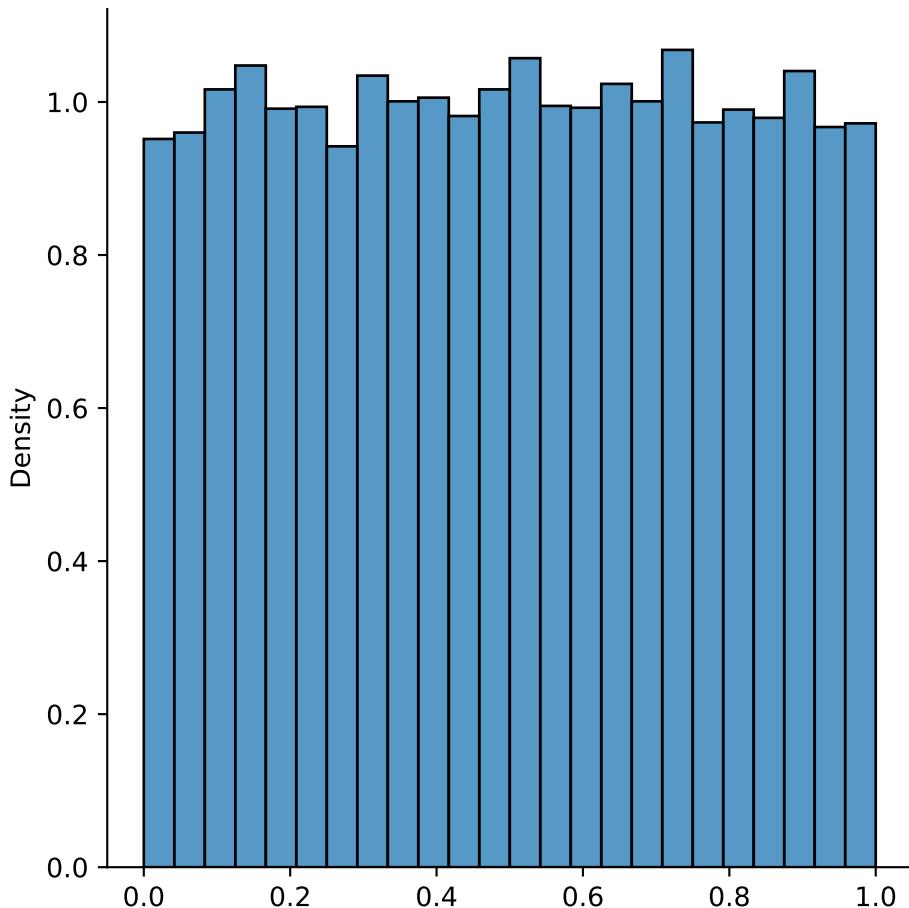
[https://www.dropbox.com/s/jvezl1ko3djwih/Example2\\_9.mp4?raw=1](https://www.dropbox.com/s/jvezl1ko3djwih/Example2_9.mp4?raw=1)

If we let the number of observations tend to infinity, then  $\hat{F}$  will converge to  $F$ , and if we also let the number of bins go to infinity, then the fraction in Equation 2.7 converges to  $F'(t_k)$ .

**Definition 2.8.** The **probability density function** or PDF of a distribution is the derivative of the CDF.

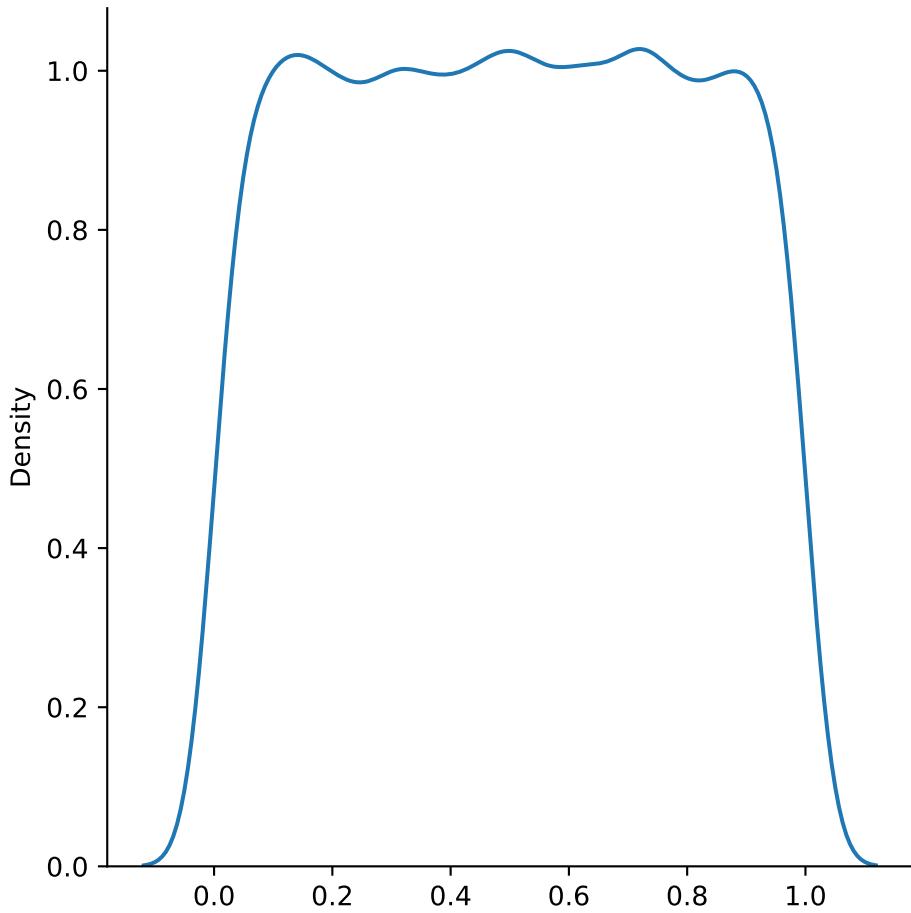
**Example 2.10.** If we have many samples, then we can use a normalized histogram to give an approximation of the PDF:

```
x = rng.uniform( size=(20000,) )
sns.displot(x, bins=24, stat="density");
```



Alternatively, we can use process called *kernel density estimation* to plot a continuous estimate of the PDF:

```
sns.displot(x, kind="kde");
```



In this case we did not obtain a particularly good approximation of the true PDF. In part this is because kernel density estimation assumes that the PDF is continuous, but here it is 1 over  $[0, 1]$  and jumps down to 0 elsewhere.

#### 2.2.4 Mean and variance

It's possible to compute the mean and variance (thus STD) of a distribution from its PDF:

$$\mu = \int x f(x) dx$$

$$\sigma^2 = \int (x - \mu)^2 f(x) dx,$$

where the integrals are taken over the domain of  $f$ .

[https://www.  
dropbox.com/  
s/  
f8mmgnxqe988lul/  
Example2\\_  
10.mp4?raw=  
1](https://www.dropbox.com/s/f8mmgnxqe988lul/Example2_10.mp4?raw=1)

**Example 2.11.** The uniform distribution over  $[0, 1]$  has  $f(x) = 1$  over that interval. Hence,

$$\mu = \int_0^1 x \, dx = \left[ \frac{1}{2}x^2 \right]_0^1 = \frac{1}{2},$$

$$\sigma^2 = \int_0^1 (x - \frac{1}{2})^2 \, dx = \frac{1}{3} - \frac{1}{2} + \frac{1}{4} = \frac{1}{12}.$$

Let's check these results empirically:

```
from numpy.random import default_rng
import numpy as np

rng = default_rng(19716)
x = rng.uniform( size=(2000,) )
print(f"\u03bc = {np.mean(x):.5f}, 12 ^ 2 = {12*np.var(x):.5f}")
```

$\mu = 0.50518, 12^2 = 1.00019$

[https://www.dropbox.com/s/sf9esiixeugop8g/Example2\\_11.mp4?raw=1](https://www.dropbox.com/s/sf9esiixeugop8g/Example2_11.mp4?raw=1)

## 2.2.5 Normal distribution

Next to perhaps the uniform distribution, the following is the most widely used distribution of a random variable.

**Definition 2.9.** The **normal distribution** or *Gaussian distribution* with mean  $\mu$  and variance  $\sigma^2$  is defined by the PDF

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}. \quad (2.8)$$

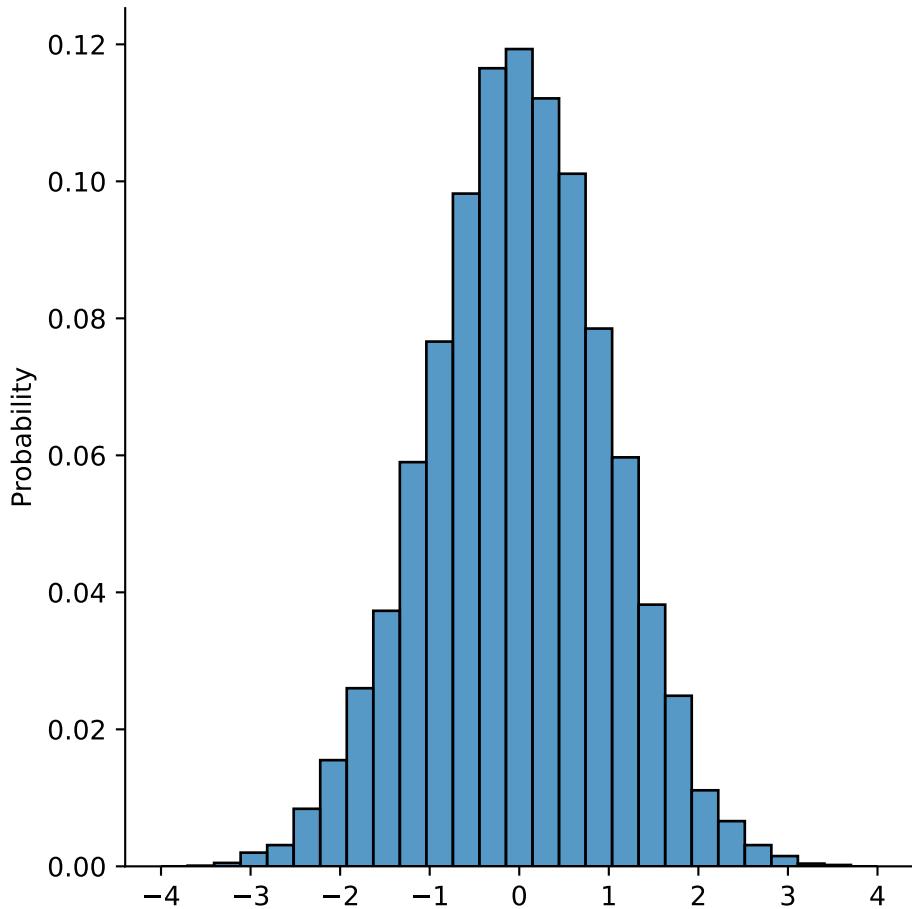
The **standard normal** distribution uses  $\mu = 0$  and  $\sigma = 1$ .

For data that are distributed normally, about 68% of the values lie within one standard deviation of the mean, and 95% lie within two standard deviations.

[https://www.dropbox.com/s/1ir47qpdrycjqm/Section2\\_2\\_5.mp4?raw=1](https://www.dropbox.com/s/1ir47qpdrycjqm/Section2_2_5.mp4?raw=1)

**Example 2.12.** The `normal` method of a NumPy RNG simulates a standard normal distribution.

```
rng = default_rng(19716)
x = rng.normal( size=(10000,) )
sns.displot(x, bins=np.linspace(-4,4,28), stat="probability");
```



We can change the variance by multiplication by  $\sigma$  and change the mean by adding  $\mu$ :

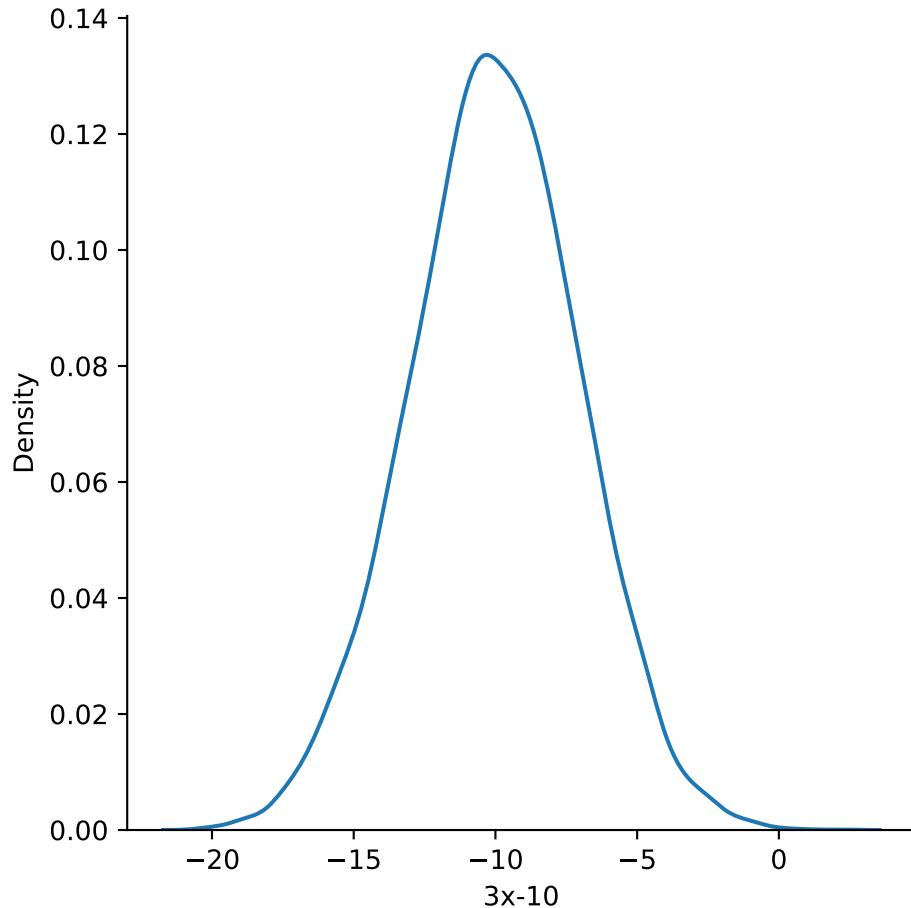
```
df = pd.DataFrame( {"x": x, "3x-10": 3*x-10} )
df.describe()
```

	x	3x-10
count	10000.000000	10000.000000
mean	-0.012414	-10.037242
std	0.993568	2.980704
min	-3.436924	-20.310773
25%	-0.672352	-12.017056
50%	-0.008374	-10.025122
75%	0.659730	-8.020809
max	4.044099	2.132297

The KDE density estimator works pretty well for normally distributed data, except in the

tails where there are few observations:

```
sns.displot(data=df, x="3x-10", kind="kde");
```

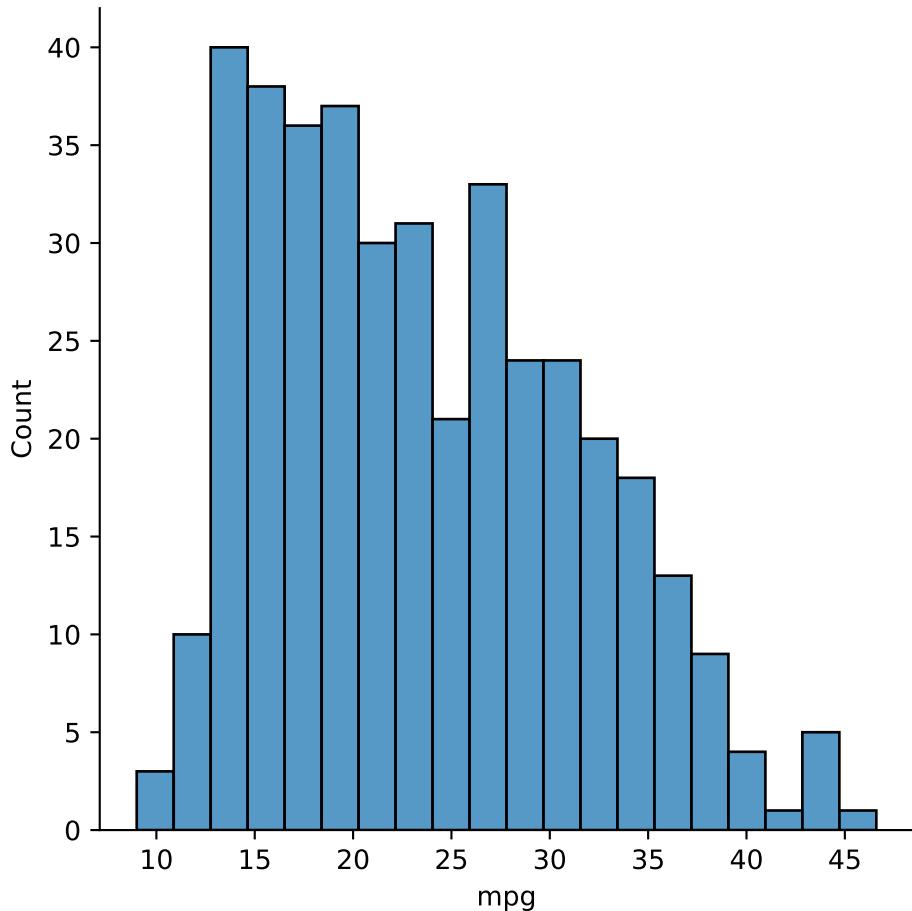


## 2.3 Grouping data

Sometimes we are interested in breaking down data by categorical values or other criteria. Both seaborn and pandas make this relatively straightforward.

Here is the distribution of the *mpg* variable over the entire dataset:

```
sns.displot(data=cars, x="mpg", bins=20);
```



We now look at ways to group the samples within this dataset.

### 2.3.1 Splitting

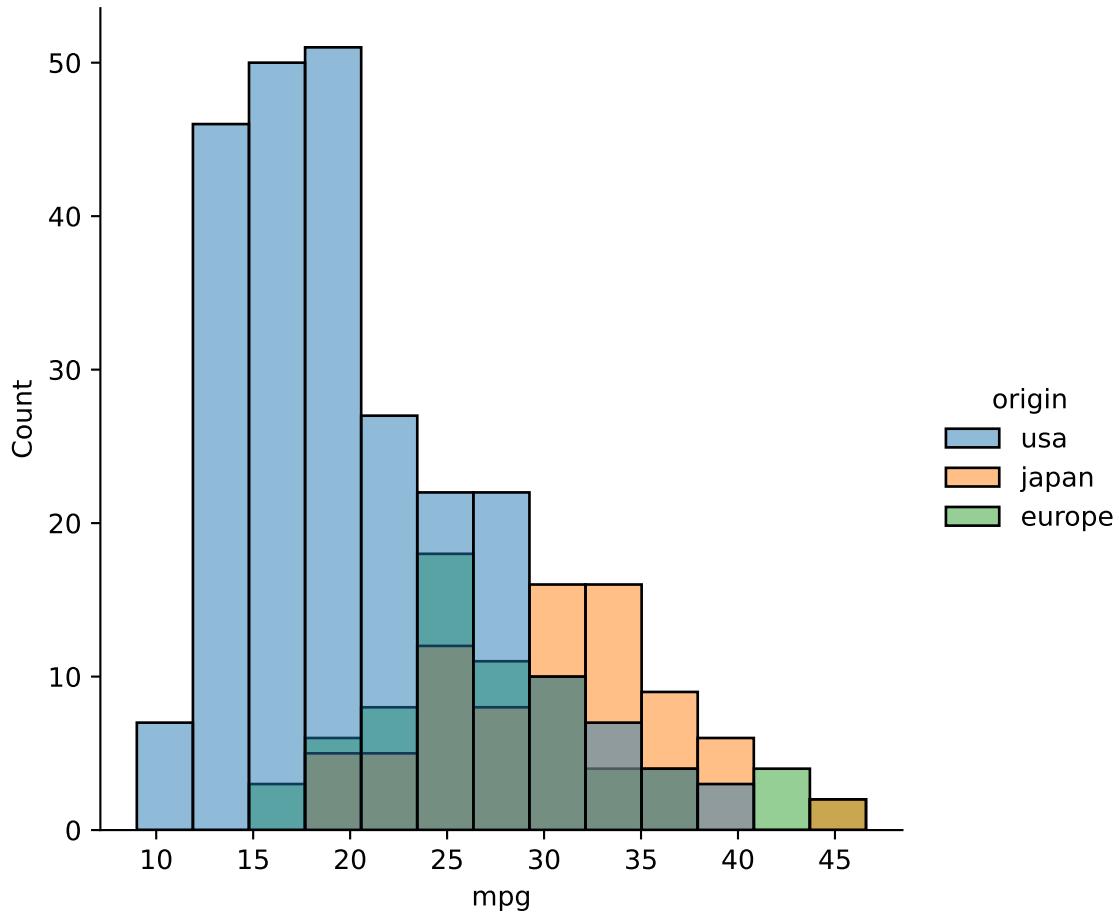
We can use categorical variables to define groups within the data set. Suppose we want to separate by the *origin* column:

```
cars["origin"].value_counts()
```

origin	
usa	249
japan	79
europe	70

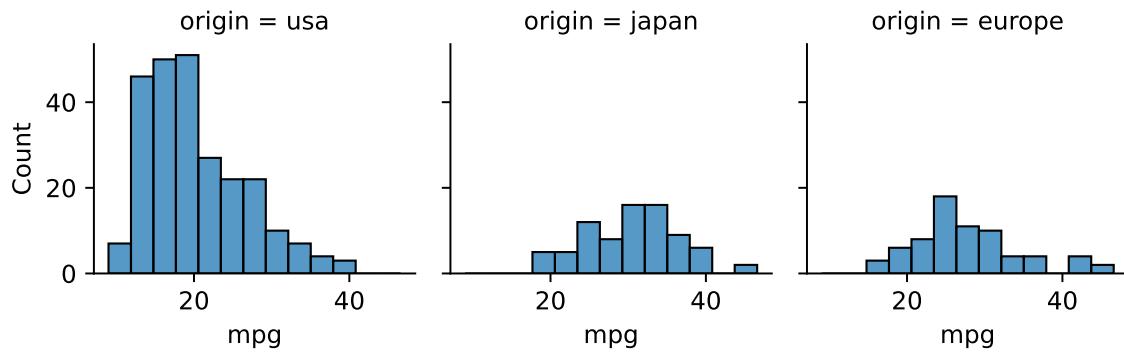
When we plot the distributions in the regions individually using different colors, we see that one is quite different:

```
sns.displot(data=cars, x="mpg", hue="origin");
```



That graph might be hard to read because of the overlaps. We can instead plot the groups in separate columns in what is often called a *facet plot*:

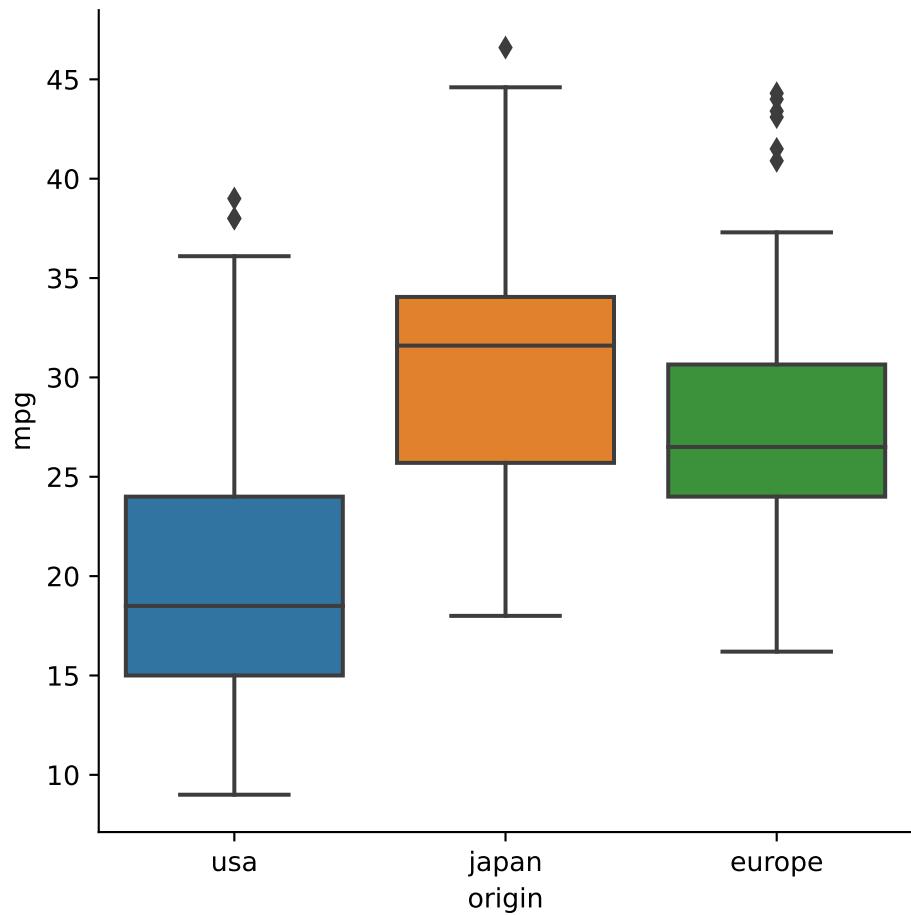
```
sns.displot(data=cars,
            x="mpg",
            col="origin", height=2.2
            );
```



Clearly, the U.S.A. cars are more clustered on the left (smaller MPG) than are the Japanese and European cars.

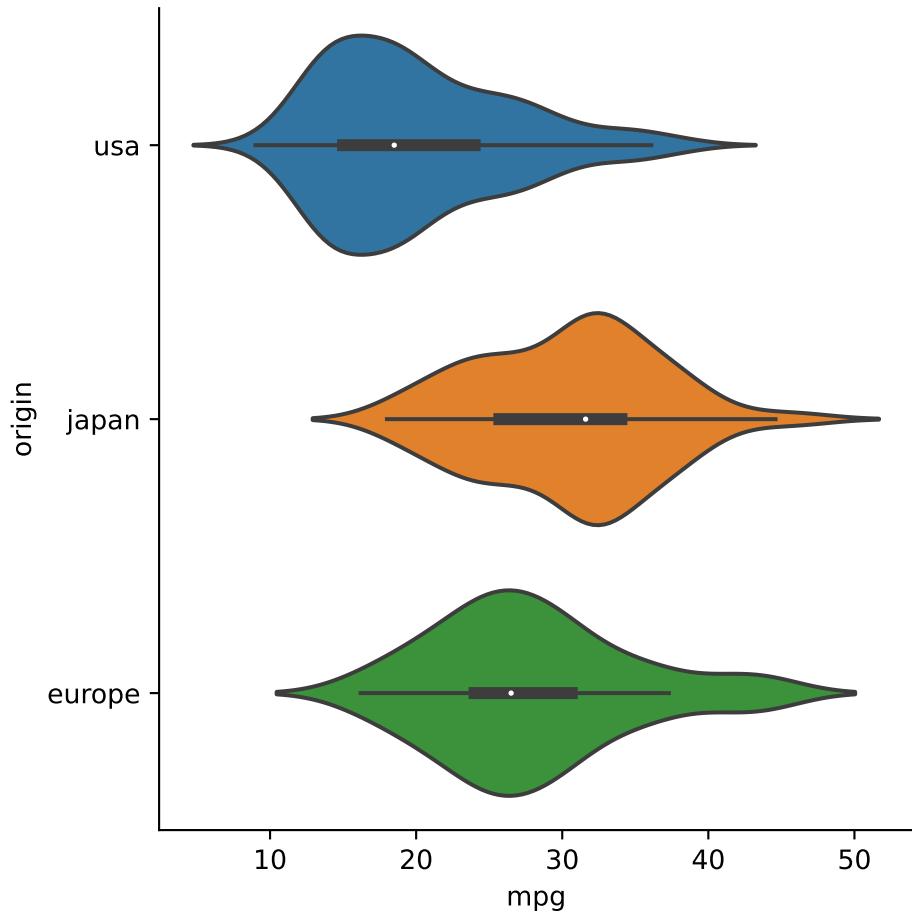
Another way to visualize grouped data is with a **box plot**:

```
sns.catplot(data=cars,
    x="origin", y="mpg",
    kind="box"
);
```



Each colored box shows the interquartile range, with the interior horizontal line showing the median. The whiskers and dots are explained in a later section. A related visualization is a **violin plot**:

```
sns.catplot(data=cars,
             x="mpg", y="origin",
             kind="violin"
            );
```



In a violin plot, the inner lines show the same information as the box plot, with the thick part showing the IQR, while the sides of the “violins” are KDE estimates of the density functions.

In pandas, the `groupby` method splits a data frame into groups based on categorical values in a designated column. So to split based on `origin` and then look at the statistics of the `mpg` column within each group, we say:

```
cars.groupby("origin")["mpg"].describe()
```

origin	count	mean	std	min	25%	50%	75%	max
europe	70.0	27.891429	6.723930	16.2	24.0	26.5	30.65	44.3
japan	79.0	30.450633	6.090048	18.0	25.7	31.6	34.05	46.6
usa	249.0	20.083534	6.402892	9.0	15.0	18.5	24.00	39.0

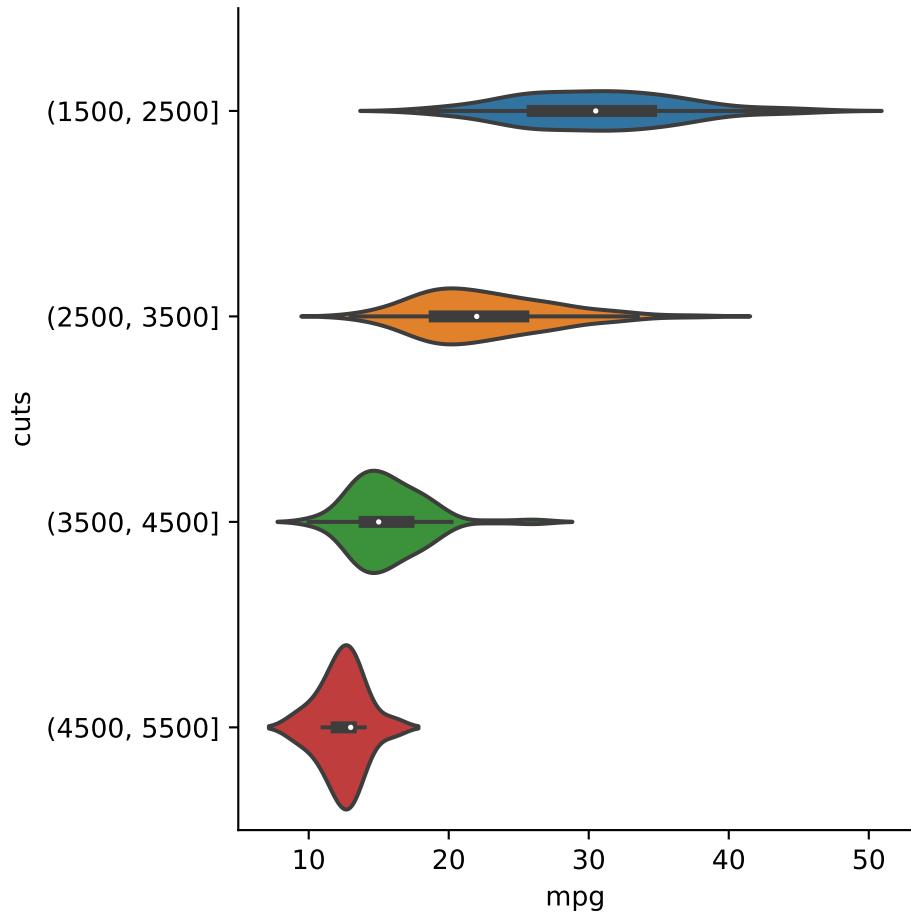
It's also possible to split using a quantitative variable. The `cut` method will put the values into bins that define the groups:

```
cuts = pd.cut(
    cars["weight"],           # series to cut by
    range(1500, 5800, 1000)   # bin edges
)

cars["cuts"] = cuts
cars.head(6)
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chev
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylan
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth s
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel s
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino
5	15.0	8	429.0	198.0	4341	10.0	70	usa	ford galaxie

```
sns.catplot(data=cars,
             x="mpg", y="cuts",
             kind="violin"
            );
```



### 2.3.2 Aggregation

Groups defined by `groupby` can then be passed through *aggregators* that reduce each grouped column to a single value. A list of the most common predefined aggregation functions is given in Table 2.1.

Table 2.1: Aggregation functions. All ignore NaN values.

method	effect
<code>count</code>	Number of values in each group
<code>mean</code>	Mean value in each group
<code>sum</code>	Sum within each group
<code>std, var</code>	Standard deviation/variance within groups
<code>min, max</code>	Min or max within groups
<code>describe</code>	Descriptive statistics

method	effect
<code>first, last</code>	First or last of group values

Here, for example, we group the rows by weight bins and find the max of *mpg* within each bin:

```
by_weight = cars.groupby(cuts)
by_weight["mpg"].max()
```

weight	mpg
(1500, 2500]	46.6
(2500, 3500]	38.0
(3500, 4500]	26.6
(4500, 5500]	16.0

[https://www.  
dropbox.com/  
s/  
p1hv4cojjpoo5a/  
Section2\\_3\\_  
2.mp4?raw=1](https://www.dropbox.com/s/p1hv4cojjpoo5a/Section2_3_2.mp4?raw=1)

If you want a more exotic operation, you can call `agg` with your own function:

```
def iqr(x):
    q1,q3 = x.quantile( [.25, .75] )
    return q3 - q1
```

```
by_weight["mpg"].agg(iqr)
```

weight	mpg
(1500, 2500]	8.450
(2500, 3500]	6.325
(3500, 4500]	3.125
(4500, 5500]	1.000

### 2.3.3 Transformation

A transformation applies a function to each element of a column, producing a result of the same length that can be indexed the same way. This transformation can be applied group by group.

For example, we can standardize to z-scores within each group separately:

[https://www.  
dropbox.com/  
s/  
jf2nkthnf4xygig/  
Section2\\_3\\_  
3.mp4?raw=1](https://www.dropbox.com/s/jf2nkthnf4xygig/Section2_3_3.mp4?raw=1)

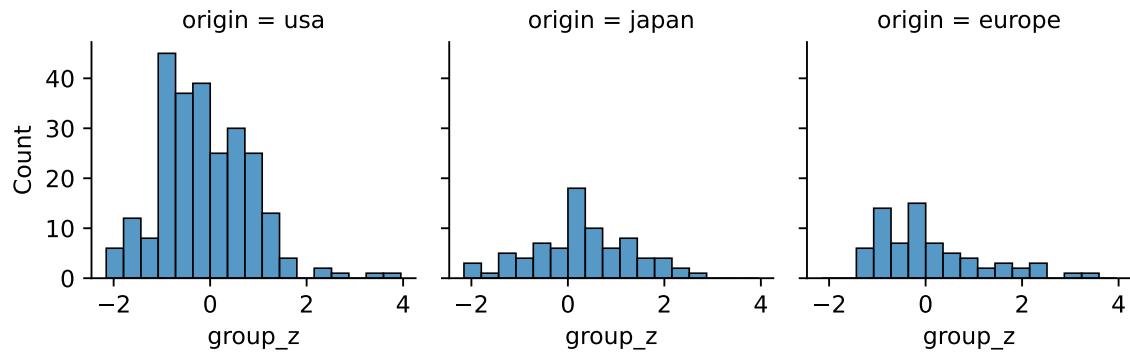
```

def standardize(x):
    return (x - x.mean()) / x.std()

cars["group_z"] = by_weight["mpg"].transform(standardize)

sns.displot(data=cars,
            x="group_z",
            col="origin", height=2.3
            );

```



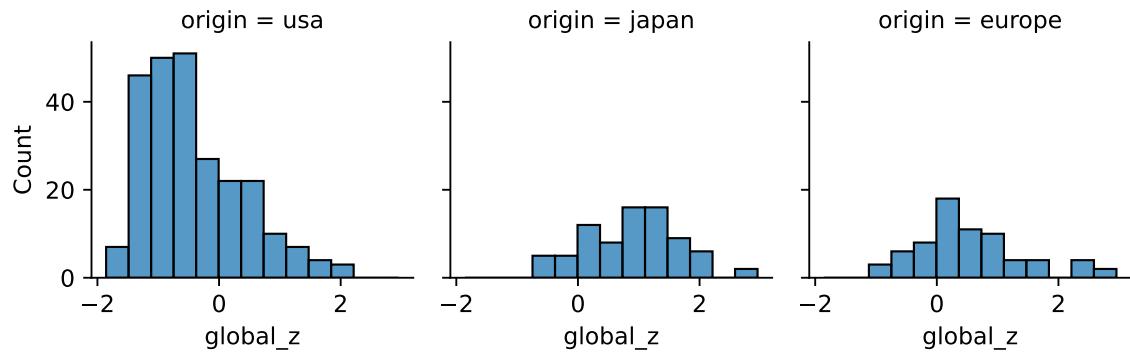
Note how this differs from computing z-scores based on global statistics:

```

cars["global_z"] = standardize( cars["mpg"] )

sns.displot(data=cars,
            x="global_z",
            col="origin", height=2.3
            );

```



### 2.3.4 Filtering

To apply a filter, provide a function that operates on a column and returns either `True`, meaning to keep the column, or `False`, meaning to reject it. This filter is applied groupwise.

For example, suppose we want to group cars by horsepower:

```
cuts = pd.cut(cars["horsepower"], range(40,220,20))
by_hp = cars.groupby(cuts)
by_hp["mpg"].count()
```

[https://www.  
dropbox.com/  
s/  
8htwbb8mj8xxmri/  
Section2\\_3\\_  
4.mp4?raw=1](https://www.dropbox.com/s/8htwbb8mj8xxmri/Section2_3_4.mp4?raw=1)

horsepower	mpg
(40, 60]	20
(60, 80]	99
(80, 100]	123
(100, 120]	48
(120, 140]	25
(140, 160]	40
(160, 180]	20
(180, 200]	7

Say we want to drop the cars belonging to groups having fewer than 30 members:

```
hp_30 = by_hp.filter( lambda x: len(x) > 29 )
hp_30.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel
11	14.0	8	340.0	160.0	3609	8.0	70	usa	plymouth
12	15.0	8	400.0	150.0	3761	9.5	70	usa	chevrolet
14	24.0	4	113.0	95.0	2372	15.0	70	japan	toyota cor

The result is a single frame in which the members of all groups that failed the filter were removed. Thus, the `horsepower` column has no values left less than 60 or greater than 160:

```
hp_30["horsepower"].describe()
```

	horsepower
count	310.000000
mean	95.309677
std	25.298367
min	61.000000
25%	75.000000
50%	90.000000
75%	105.000000
max	160.000000

```
.groupby("color")["Gender"]
```



## 2.4 Outliers

Informally, an **outlier** is a data value that is considered to be far from typical. In some applications, such as detecting earthquakes or cancer, outliers are the cases of real interest. But we will be thinking of them as unwelcome values that might result from equipment failure, confounding effects, mistyping a value, using an extreme value to represent missing data, and so on. In such cases we want to minimize the effect of the outliers on the statistics.

It is well known, for instance, that the mean is more sensitive to outliers than the median is.

**Example 2.13.** The values 1, 2, 3, 4, 5 have a mean and median both equal to 3. If we change the largest value to be a lot larger, say 1, 2, 3, 4, 1000, then the mean changes to 202. But the median is still 3!

If you want to use a method that is vulnerable to outliers, it's typical to remove such values early on. There are various ways of deciding what qualifies as an outlier, with no one-size recommendation for all applications.

### 2.4.1 IQR

Let  $Q_{25}$  and  $Q_{75}$  be the first and third quartiles (i.e., 25% and 75% percentiles), and let  $I = Q_{75} - Q_{25}$  be the interquartile range (IQR). Then  $x$  is an outlier value if

$$x < Q_{25} - 1.5I \text{ or } x > Q_{75} + 1.5I. \quad (2.9)$$

In a box plot, the whiskers growing from a box show the extent of the non-outlier data, and the dots beyond the whiskers represent outliers.

**Example 2.14.** Let's look at another data set, based on an fMRI experiment:

```
fmri = sns.load_dataset("fmri")
fmri.head()
```

	subject	timepoint	event	region	signal
0	s13	18	stim	parietal	-0.017552
1	s5	14	stim	parietal	-0.080883
2	s12	18	stim	parietal	-0.081033
3	s11	18	stim	parietal	-0.046134
4	s10	18	stim	parietal	-0.037970

We want to focus on the *signal* column, splitting according to the *event*.

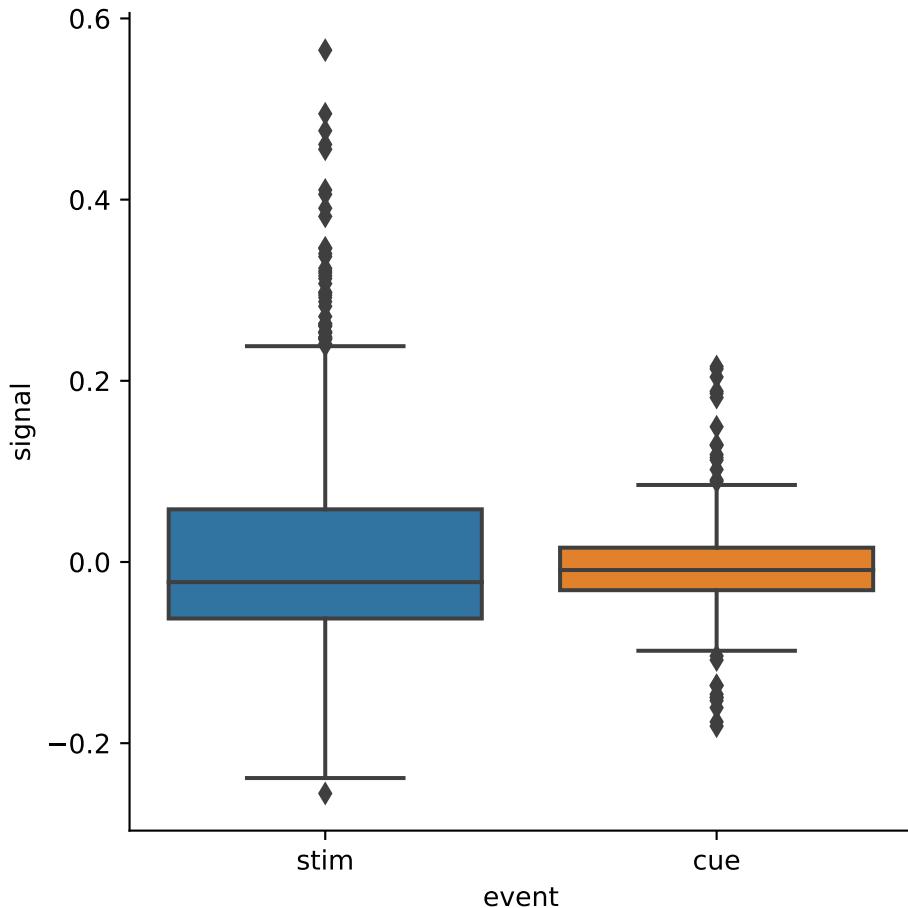
```
by_event = fmri.groupby("event")
by_event["signal"].describe()
```

event	count	mean	std	min	25%	50%	75%	max
cue	532.0	-0.006669	0.047752	-0.181241	-0.031122	-0.008871	0.015825	0.215735
stim	532.0	0.013748	0.123179	-0.255486	-0.062378	-0.022202	0.058143	0.564985

Here is a box plot of the signal for these groups.

```
sns.catplot(data=fmri,
             x="event", y="signal",
             kind="box")
```

) ;



The dots lying outside the whiskers in the plot can be considered outliers satisfying one of the inequalities in Equation 2.9.

Let's now remove the outliers. We start with a function that computes a Boolean-valued series for a given input. This function is applied as a `transform` to the data as grouped by *events*:

```
def is_outlier(x):
    Q25, Q75 = x.quantile([.25,.75])
    I = Q75 - Q25
    return (x < Q25 - 1.5*I) | (x > Q75 + 1.5*I)

outliers = by_event["signal"].transform(is_outlier)
```

```
fmri.loc[outliers,"event"].value_counts()
```

event	
stim	40
cue	26

You can see above that there are 66 outliers. To negate the outlier indicator, we can use `~outs` as a row selector.

```
cleaned = fmri[~outliers]
```

The median values are barely affected by the omission of the outliers:

```
print( "medians with outliers:" )
print( by_event["signal"].median() )
print( "\nmedians without outliers:" )
print( cleaned.groupby("event")["signal"].median() )
```

```
medians with outliers:
event
cue    -0.008871
stim   -0.022202
Name: signal, dtype: float64
```

```
medians without outliers:
event
cue    -0.009006
stim   -0.028068
Name: signal, dtype: float64
```

The means, though, show much greater change:

```
print( "means with outliers:" )
print( by_event["signal"].mean() )
print( "\nmeans without outliers:" )
print( cleaned.groupby("event")["signal"].mean() )
```

```
means with outliers:
event
cue    -0.006669
stim   0.013748
```

```

Name: signal, dtype: float64

means without outliers:
event
cue    -0.008243
stim   -0.010245
Name: signal, dtype: float64

```

For the *stim* case in particular, the mean value changes by almost 200%, including a sign change. (Relative to the standard deviation, it's closer to a 20% change.)

[https://www.  
dropbox.com/  
s/  
t2ndktsuxnmp5pt/  
Example2\\_  
14.mp4?raw=  
1](https://www.dropbox.com/s/t2ndktsuxnmp5pt/Example2_14.mp4?raw=1)

## 2.4.2 Mean and STD

For normal distributions, values more than twice the standard deviation  $\sigma$  from the mean could be considered to be outliers; this would exclude 5% of the values, on average. A less aggressive criterion is to allow a distance of  $3\sigma$ , which excludes only about 0.3% of the values. The IQR criterion above corresponds to about  $2.7\sigma$  in the normal distribution case.

### Note

A criticism of classical statistics is that much of it is conditioned on the assumption of normal distributions. This assumption is often violated by real datasets; quantities that depend on normality should be used judiciously.

**Example 2.15.** The following plot shows the outlier cutoffs for 2000 samples from a normal distribution, using the criteria for 2 (red), 3 (blue), and 1.5 IQR (black).

```

import matplotlib.pyplot as plt
from numpy.random import default_rng
randn = default_rng(1).normal

x = pd.Series(randn(size=2000))
sns.displot(data=x,bins=30);
m,s = x.mean(),x.std()
plt.axvline(m-2*s,color='r')
plt.axvline(m+2*s,color='r')
plt.axvline(m-3*s,color='b')
plt.axvline(m+3*s,color='b')

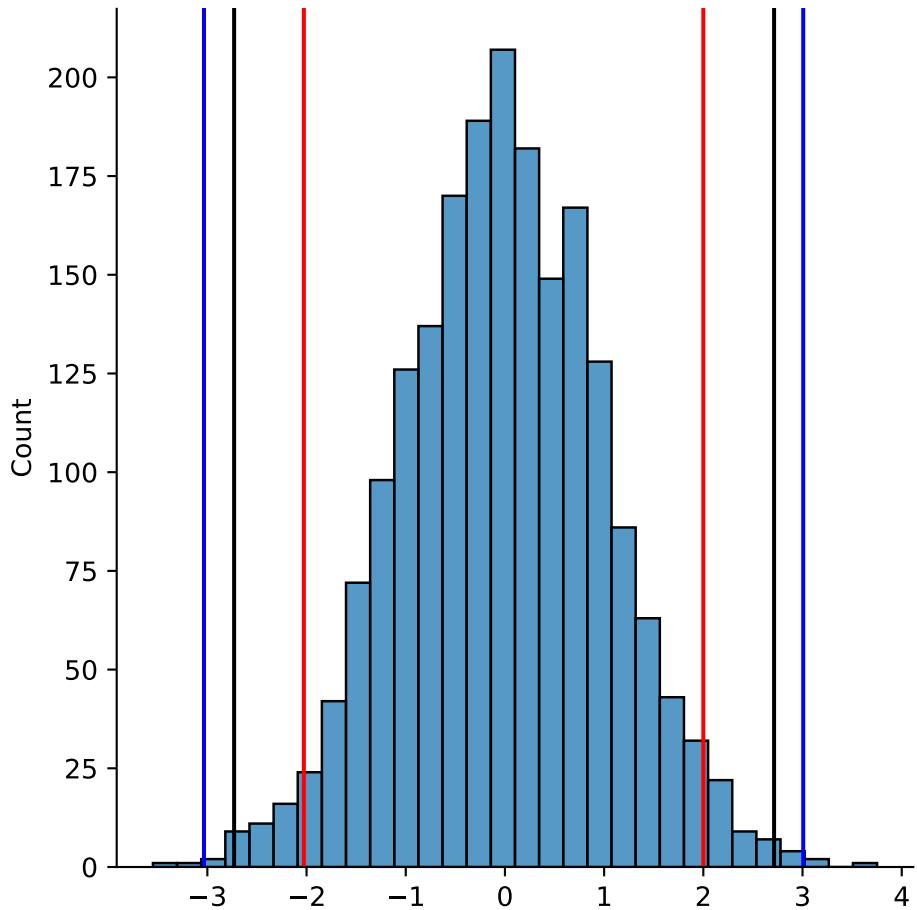
q1,q3 = x.quantile([.25,.75])

```

```

plt.axvline(q3+1.5*(q3-q1),color='k')
plt.axvline(q1-1.5*(q3-q1),color='k');

```



For asymmetric distributions, or those with a heavy tail, these criteria might show greater differences.

[https://www.dropbox.com/s/4j52cut7dyomqaf/Example2\\_15.mp4?raw=1](https://www.dropbox.com/s/4j52cut7dyomqaf/Example2_15.mp4?raw=1)



## 2.5 Correlation

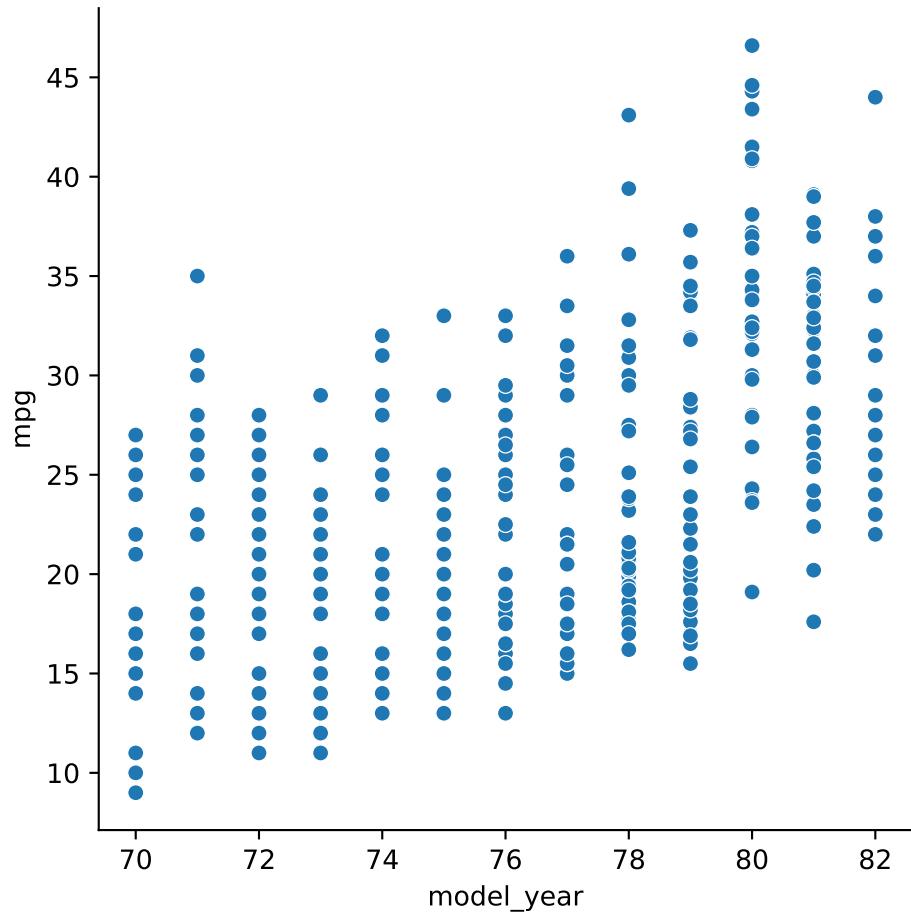
There are often variables that we believe to be linked, either because one influences the other, or because both are influenced by some other factor. In either case, we say the quantities are *correlated*.

There are several ways to measure correlation. It's good practice to look at the data first, though, before jumping to the numbers.

### 2.5.1 Relational plots

We can use `relplot` to make a scatter plot of two different columns in a frame:

```
sns.relplot(data=cars,  
            x="model_year", y="mpg"  
            );
```

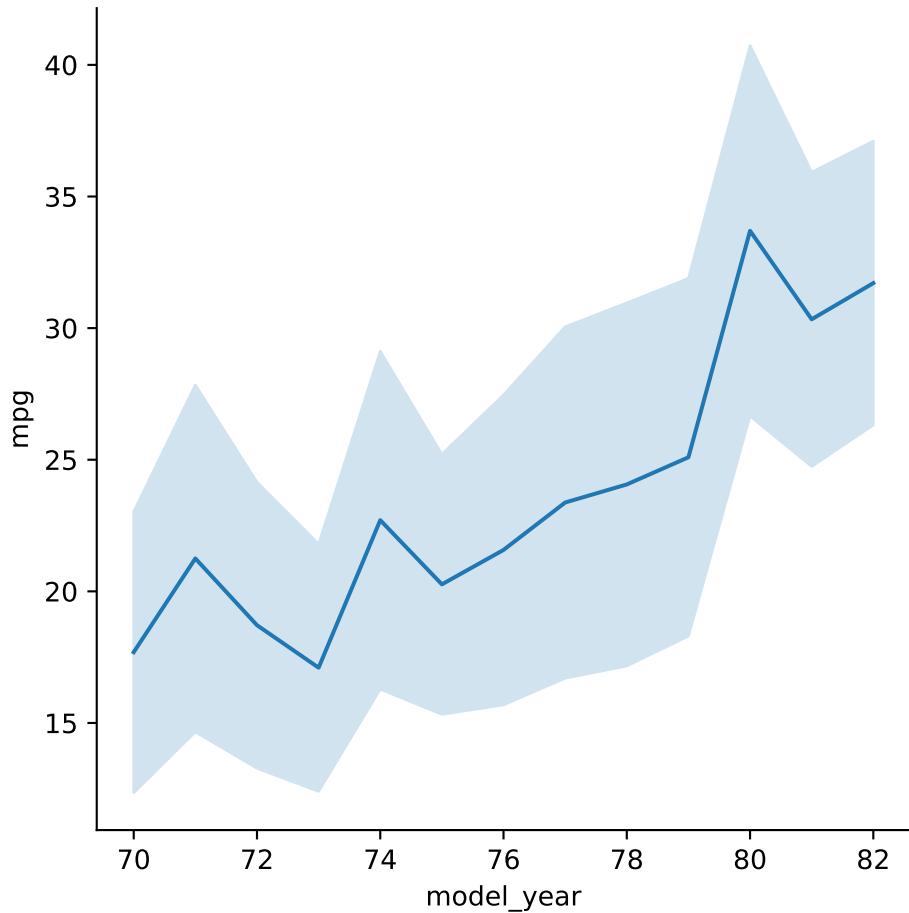


Like the other plot types above, we can use color, column, marker size, etc. to separate the dots into groups.

If we want to emphasize a trend, we can instead plot the average value at each  $x$  with error bars:

```
sns.relplot(data=cars,
            x="model_year", y="mpg",
            kind="line", errorbar="sd"
            );
```

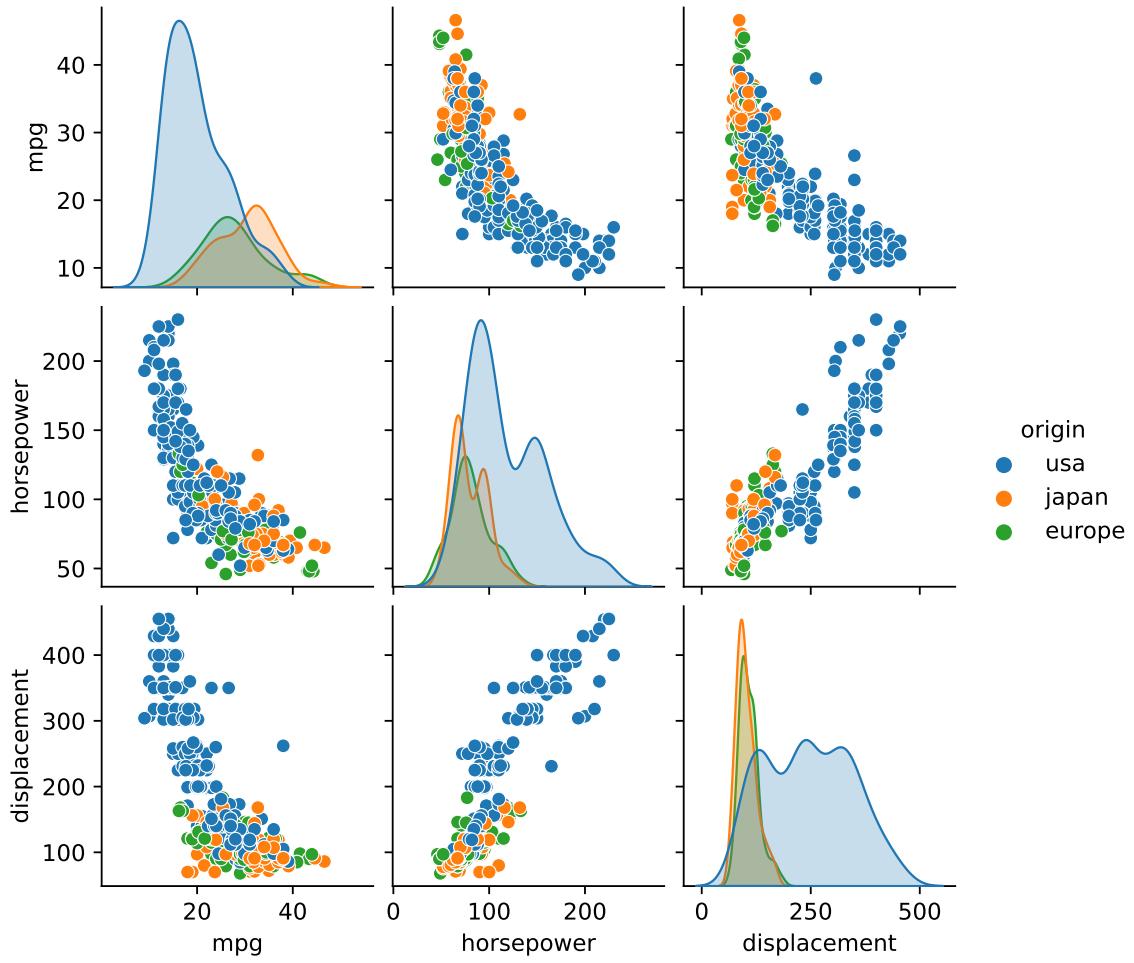
[https://www.  
dropbox.com/  
s/  
htm7sso6p51n9pd/  
Section2\\_5\\_  
1.mp4?raw=1](https://www.dropbox.com/s/htm7sso6p51n9pd/Section2_5_1.mp4?raw=1)



The error ribbon above is drawn at one standard deviation around the mean.

In order to see multiple pairwise scatter plots, we can use `pairplot` in seaborn:

```
columns = [ "mpg", "horsepower",
            "displacement", "origin" ]  
  
sns.pairplot(data=cars[columns],
              hue="origin", height=2
            );
```



The panels along the diagonal show each quantitative variable's PDF. The other panels show scatter plots putting one pair at a time of the variables on the coordinate axes.

### 2.5.2 Covariance

**Definition 2.10.** Suppose we have two series of observations,  $[x_i]$  and  $[y_i]$ , representing observations of random quantities  $X$  and  $Y$  having means  $\mu_X$  and  $\mu_Y$ . Their **covariance** is defined as

$$\text{Cov}(X, Y) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_X)(y_i - \mu_Y).$$

Note that the values  $x_i - \mu_X$  and  $y_i - \mu_Y$  are deviations from the means. It follows from

the definitions that

$$\begin{aligned}\text{Cov}(X, X) &= \sigma_X^2, \\ \text{Cov}(Y, Y) &= \sigma_Y^2,\end{aligned}$$

i.e., self-covariance is simply variance.

Covariance is not easy to interpret. Its units are the products of the units of the two variables, and it is sensitive to rescaling the variables (e.g., grams versus kilograms).

### 2.5.3 Pearson coefficient

We can remove the dependence on units and scale by applying the covariance to standardized scores for both variables. The following is the best-known measure of correlation.

**Definition 2.11.** For the populations of  $X$  and  $Y$ , the **Pearson correlation coefficient** is

$$\begin{aligned}\rho(X, Y) &= \frac{1}{n} \sum_{i=1}^n \left( \frac{x_i - \mu_X}{\sigma_X} \right) \left( \frac{y_i - \mu_Y}{\sigma_Y} \right) \\ &= \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y},\end{aligned}\tag{2.10}$$

where  $\sigma_X^2$  and  $\sigma_Y^2$  are the population variances of  $X$  and  $Y$ .

For samples from the two populations, we use

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}},\tag{2.11}$$

where  $\bar{x}$  and  $\bar{y}$  are sample means.

Both  $\rho_{XY}$  and  $r_{xy}$  are between  $-1$  and  $1$ , with the endpoints indicating perfect correlation (inverse or direct).

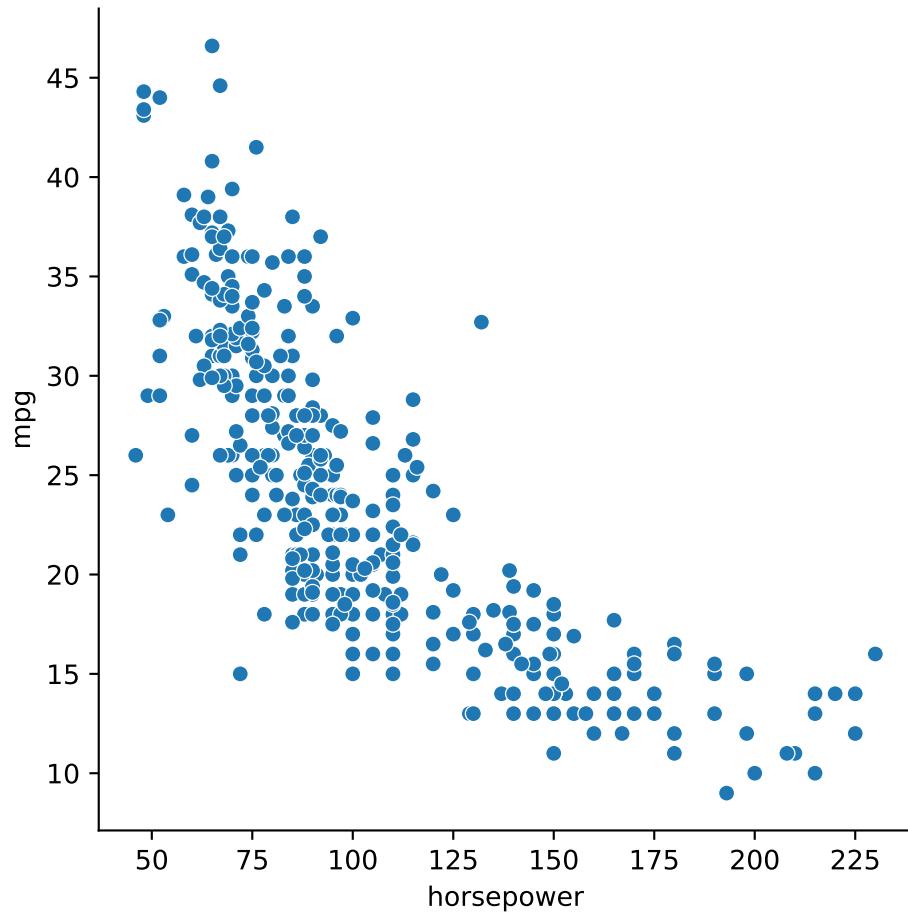
An equivalent formula for  $r_{xy}$  is

$$r_{xy} = \frac{1}{n-1} \sum_{i=1}^n \left( \frac{x_i - \bar{x}}{s_x} \right) \left( \frac{y_i - \bar{y}}{s_y} \right),\tag{2.12}$$

where the quantities in parentheses are z-scores.

**Example 2.16.** We might reasonably expect horsepower and miles per gallon to be inversely correlated:

```
sns.relplot( data=cars,  
             x="horsepower", y="mpg"  
           );
```



Covariance allows us to confirm the relationship:

```
cars[ ["horsepower", "mpg"] ].cov()
```

	horsepower	mpg
horsepower	1481.569393	-233.857926
mpg	-233.857926	61.089611

But should these numbers considered big? The Pearson coefficient is more helpful:

```
cars[ ["horsepower", "mpg"] ].corr()
```

	horsepower	mpg
horsepower	1.000000	-0.778427
mpg	-0.778427	1.000000

The value of about  $-0.79$  suggests that knowing one of the values would allow us to predict the other one rather well using a best-fit straight line (more on that in a future chapter).

As usual when dealing with means, however, the Pearson coefficient can be sensitive to outlier values.

**Example 2.17.** The Pearson coefficient of any variable with itself is 1. But let's correlate two series that differ in only one element:  $0, 1, 2, \dots, 19$ , and the same sequence but with the fifth value replaced by  $-100$ :

[https://www.  
dropbox.com/  
s/  
0jpbrcfk2lu25zd/  
Example2\\_  
16.mp4?raw=  
1](https://www.dropbox.com/s/0jpbrcfk2lu25zd/Example2_16.mp4?raw=1)

```
x = pd.Series( range(20) )
y = x.copy()
y[4] = -100
x.corr(y)
```

0.43636501543147005

Despite the change being in a single value, over half of the predictive power was lost.

## 2.5.4 Spearman coefficient

The Spearman coefficient is one way to lessen the impact of outliers when measuring correlation. The idea is that the values are used only in their orderings.

**Definition 2.12.** If  $x_1, \dots, x_n$  is a series of observations, let their sorted ordering be

$$x_{s_1}, x_{s_2}, \dots, x_{s_n}.$$

Then  $s_1, s_2, \dots, s_n$  is the **rank series** of  $x$ .

**Definition 2.13.** The **Spearman coefficient** of two series of equal length is the Pearson coefficient of their rank series.

**Example 2.18.** Returning to Example 2.17, we find the Spearman coefficient is barely affected by the single outlier:

```
x = pd.Series( range(20) )
y = x.copy()
y[4] = -100
x.corr(y, "spearman")
```

0.9849624060150375

It's trivial in this case to produce the two rank series by hand:

```
s = pd.Series( range(1,21) )      # already sorted
t = s.copy()
t[:5] = [2,3,4,5,1]      # modified sort ordering

t.corr(s)
```

0.9849624060150375

As long as  $y[4]$  is negative, it doesn't matter what its particular value is:

```
y[4] = -1000000
x.corr(y, "spearman")
```

0.9849624060150375

Since real data almost always features outlying or anomalous values, it's important to think about the robustness of the statistics you choose.

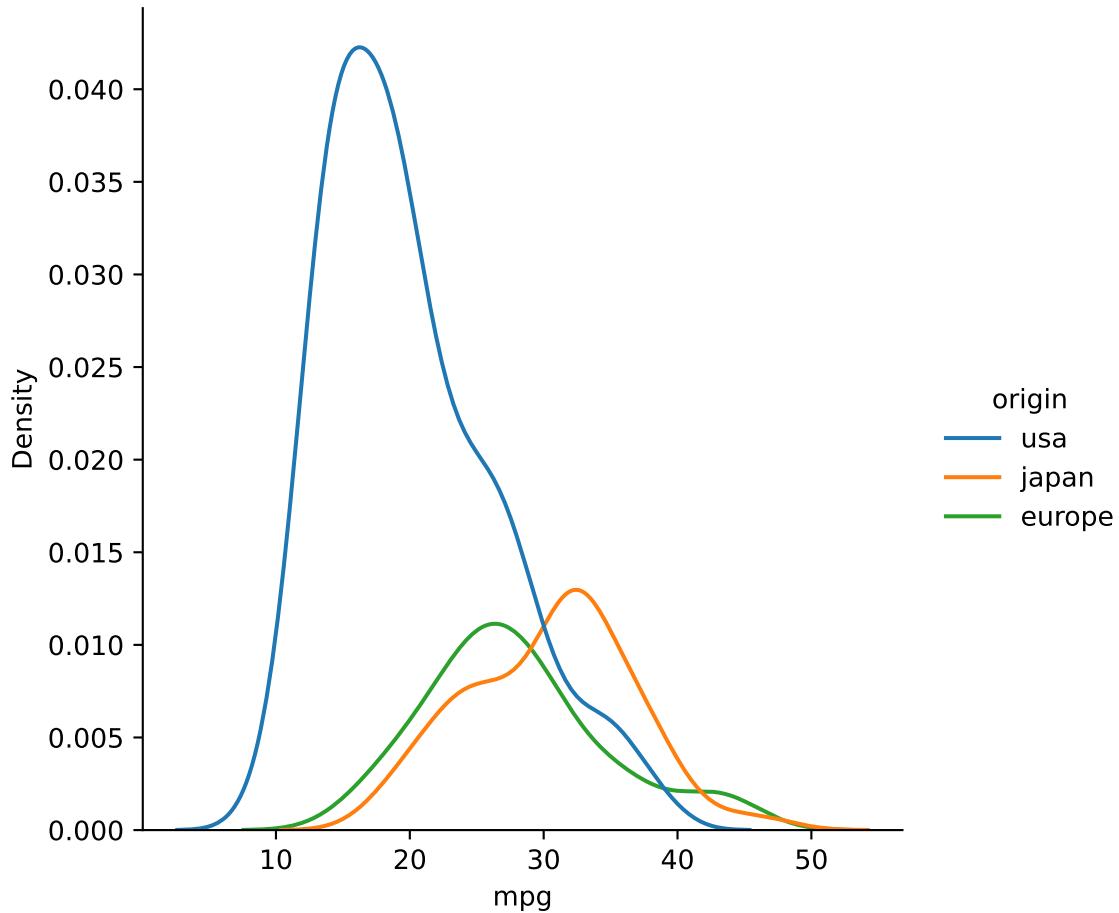
[https://www.  
dropbox.com/  
s/  
xzq46nce6vv12oc/  
Section2\\_5\\_  
4.mp4?raw=1](https://www.dropbox.com/s/xzq46nce6vv12oc/Section2_5_4.mp4?raw=1)

## 2.5.5 Categorical correlation

An ordinal variable, such as the days of the week, is often straightforward to quantify as integers. But a nominal variable poses a different challenge.

**Example 2.19.** Grouped histograms suggest an association between country of origin and MPG:

```
sns.displot(data=cars, kind="kde",
             x="mpg", hue="origin");
```



How can we quantify the association? The first step is to convert the *origin* column into dummy variables:

```
dum = pd.get_dummies(cars, columns=["origin"])
dum.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	name
0	18.0	8	307.0	130.0	3504	12.0	70	chevrolet chevelle m
1	15.0	8	350.0	165.0	3693	11.5	70	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	ford torino

The original *origin* column has been replaced by three binary indicator columns. Now we can look for correlations between them and *mpg*:

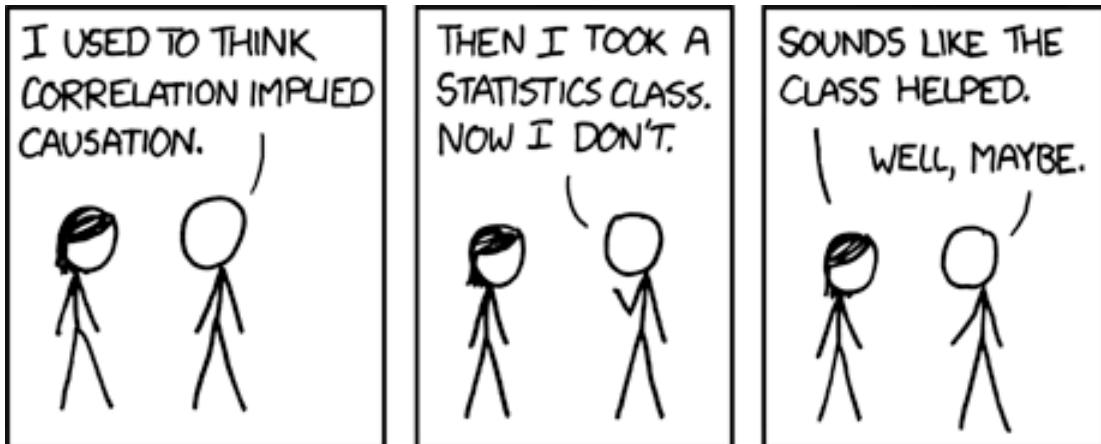
```

columns = [
    "mpg",
    "origin_europe",
    "origin_japan",
    "origin_usa"
]
dum[columns].corr()

```

	mpg	origin_europe	origin_japan	origin_usa
mpg	1.000000	0.259022	0.442174	-0.568192
origin_europe	0.259022	1.000000	-0.229895	-0.597198
origin_japan	0.442174	-0.229895	1.000000	-0.643317
origin_usa	-0.568192	-0.597198	-0.643317	1.000000

As you can see from the above, `europe` and `japan` are positively associated with `mpg`, while `usa` is inversely associated with `mpg`.



## 2.6 Cautionary tales

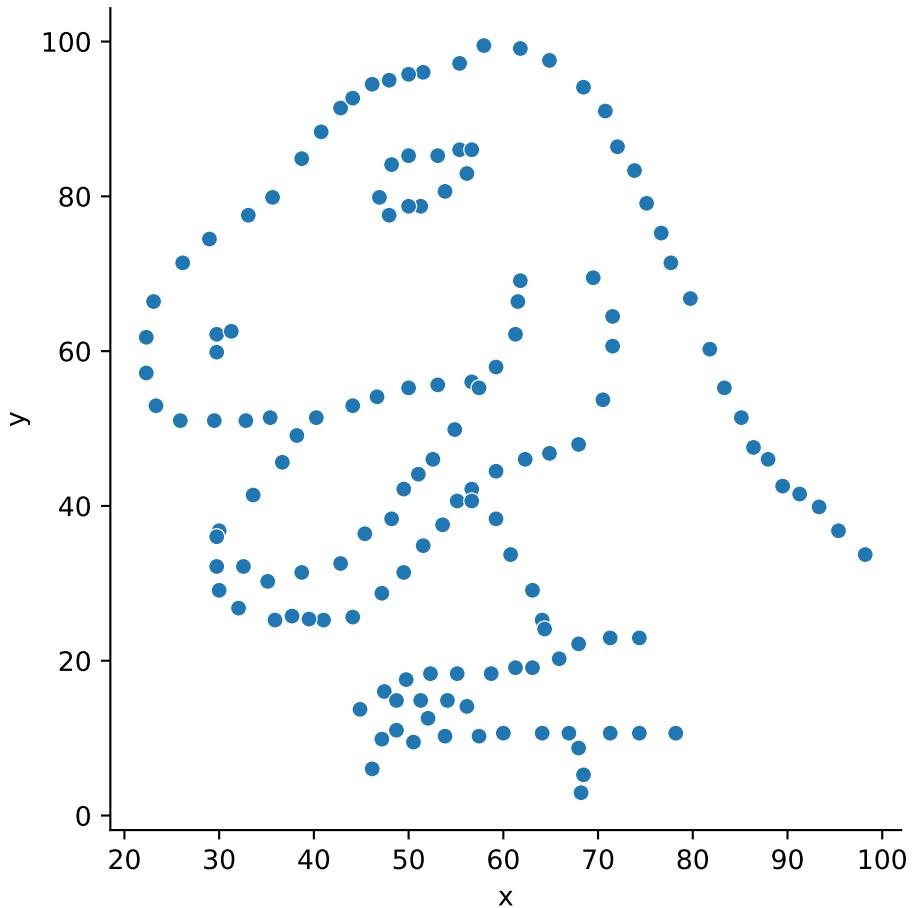
Attaching theorem-supported numbers to real data feels precise and infallible. The theorems do what they say, of course—they’re theorems—but our intuition can be a little too ready to attach significance to the numbers, causing misconceptions or mistakes. Proper visualizations can help us see through such issues.

### 2.6.1 The datasaurus

The Datasaurus Dozen is a collection of datasets that highlights the perils of putting blind trust into summary statistics. The Datasaurus is a set of 142 points making a handsome

portrait:

```
dozen = pd.read_csv("_datasets/DatasaurusDozen.tsv", delimiter="\t")
sns.relplot(data=dozen[dozen["dataset"]=="dino"], x="x", y="y");
```



However, there are 12 other datasets that all have roughly the same mean and variance for  $x$  and  $y$ , and the same correlations between them:

```
by_set = dozen.groupby("dataset")
by_set.mean()
```

[https://www.  
dropbox.com/  
s/  
qmnh7s2w58oim7n/  
Section2\\_6\\_  
1.mp4?raw=1](https://www.dropbox.com/s/qmnh7s2w58oim7n/Section2_6_1.mp4?raw=1)

dataset	x	y
away	54.266100	47.834721
bullseye	54.268730	47.830823
circle	54.267320	47.837717
dino	54.263273	47.832253
dots	54.260303	47.839829
h_lines	54.261442	47.830252
high_lines	54.268805	47.835450
slant_down	54.267849	47.835896
slant_up	54.265882	47.831496
star	54.267341	47.839545
v_lines	54.269927	47.836988
wide_lines	54.266916	47.831602
x_shape	54.260150	47.839717

`by_set.std()`

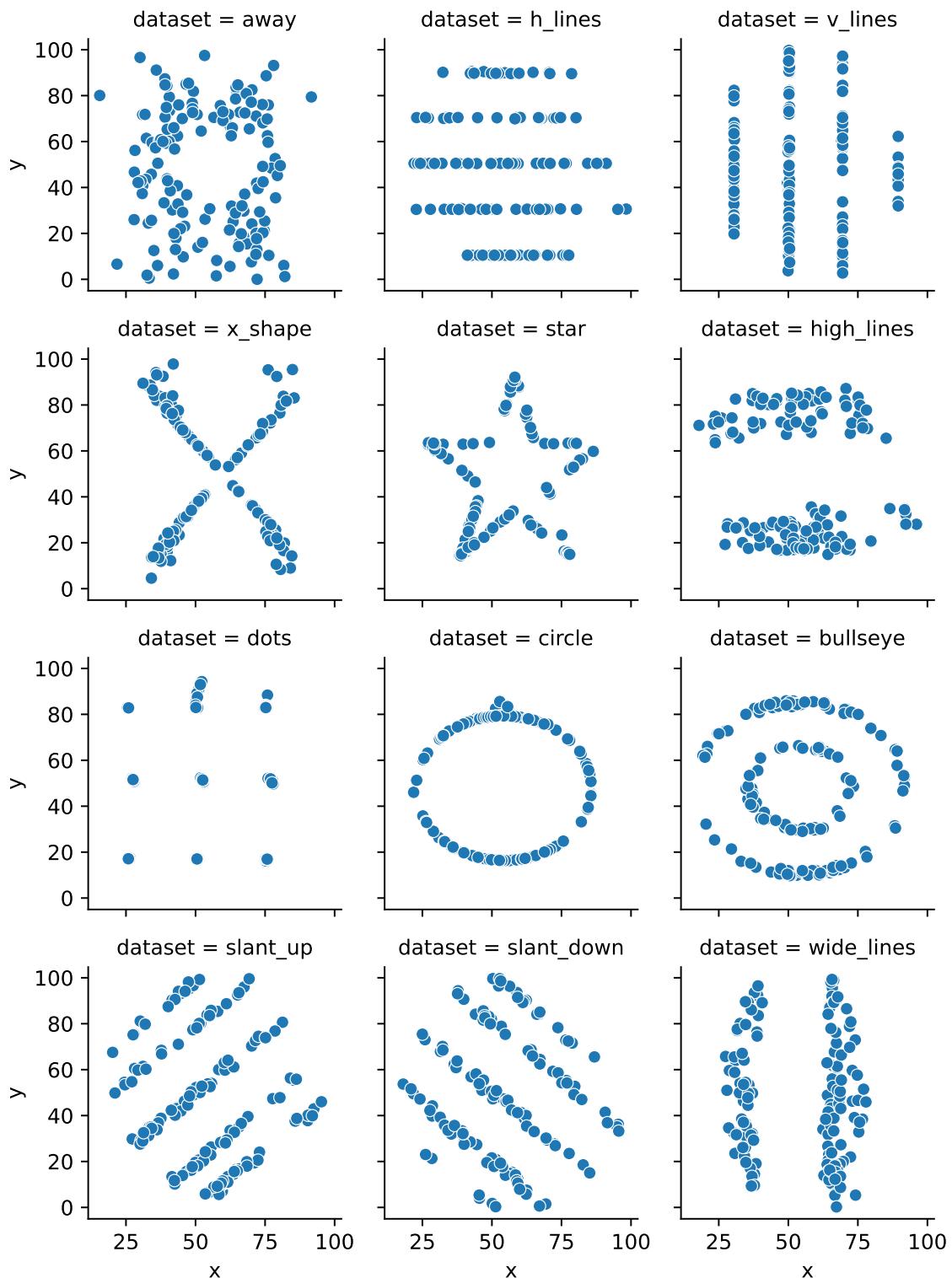
dataset	x	y
away	16.769825	26.939743
bullseye	16.769239	26.935727
circle	16.760013	26.930036
dino	16.765142	26.935403
dots	16.767735	26.930192
h_lines	16.765898	26.939876
high_lines	16.766704	26.939998
slant_down	16.766759	26.936105
slant_up	16.768853	26.938608
star	16.768959	26.930275
v_lines	16.769959	26.937684
wide_lines	16.770000	26.937902
x_shape	16.769958	26.930002

`by_set.corr()`

dataset		x	y
away	x	1.000000	-0.064128
	y	-0.064128	1.000000
bullseye	x	1.000000	-0.068586
	y	-0.068586	1.000000
circle	x	1.000000	-0.068343
	y	-0.068343	1.000000
dino	x	1.000000	-0.064472
	y	-0.064472	1.000000
dots	x	1.000000	-0.060341
	y	-0.060341	1.000000
h_lines	x	1.000000	-0.061715
	y	-0.061715	1.000000
high_lines	x	1.000000	-0.068504
	y	-0.068504	1.000000
slant_down	x	1.000000	-0.068980
	y	-0.068980	1.000000
slant_up	x	1.000000	-0.068609
	y	-0.068609	1.000000
star	x	1.000000	-0.062961
	y	-0.062961	1.000000
v_lines	x	1.000000	-0.069446
	y	-0.069446	1.000000
wide_lines	x	1.000000	-0.066575
	y	-0.066575	1.000000
x_shape	x	1.000000	-0.065583
	y	-0.065583	1.000000

However, a plot reveals that these sets are, to put it mildly, quite distinct:

```
others = dozen[ dozen["dataset"] != "dino" ]
sns.relplot(data=others,
             x="x", y="y",
             col="dataset", col_wrap=3, height=2.2
            );
```



### ! Important

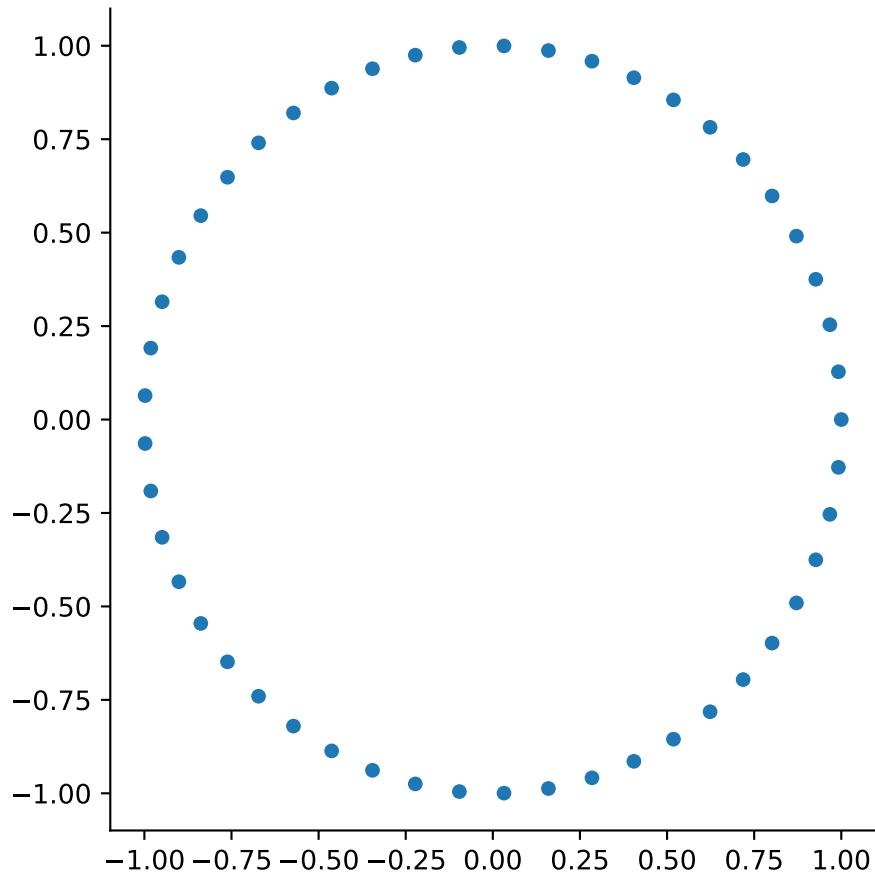
Always plot your data.

Always. Plot. Your. Data!

### 2.6.2 Correlation vs. dependence

In casual speech, you might expect two variables that are uncorrelated to be unrelated. But that is not at all how the mathematical definitions play out. For example, here are  $x$  and  $y$  variables that both depend on a hidden variable  $\theta$ :

```
theta = np.linspace(0,2*np.pi,50)
x = pd.Series(np.cos(theta))
y = pd.Series(np.sin(theta))
sns.relplot(x=x, y=y);
```



Neither informally nor mathematically would we say that  $x$  and  $y$  are independent! For example, if  $y = 0$ , then there is only one possible value for  $x$ . Yet the correlation between  $x$  and  $y$  is zero:

```
x.corr(y)
```

```
8.276095507101827e-18
```

Correlation can only measure the extent of the relationship that is *linear*;  $x$  and  $y$  lying on a straight line means perfect correlation. In this case, every line you can draw that passes through  $(0, 0)$  is essentially equally bad at representing the data, and correlation cannot express the relationship.

### 2.6.3 Simpson's paradox

The penguin dataset contains a common paradox—or a counterintuitive phenomenon, at least. Two of the variables show a fairly strong negative correlation:

```
penguins = sns.load_dataset("penguins")
columns = [ "body_mass_g", "bill_depth_mm" ]
penguins[columns].corr()
```

	body_mass_g	bill_depth_mm
body_mass_g	1.000000	-0.471916
bill_depth_mm	-0.471916	1.000000

But something surprising happens if we compute correlations *after* grouping by species.

```
penguins.groupby("species")[columns].corr()
```

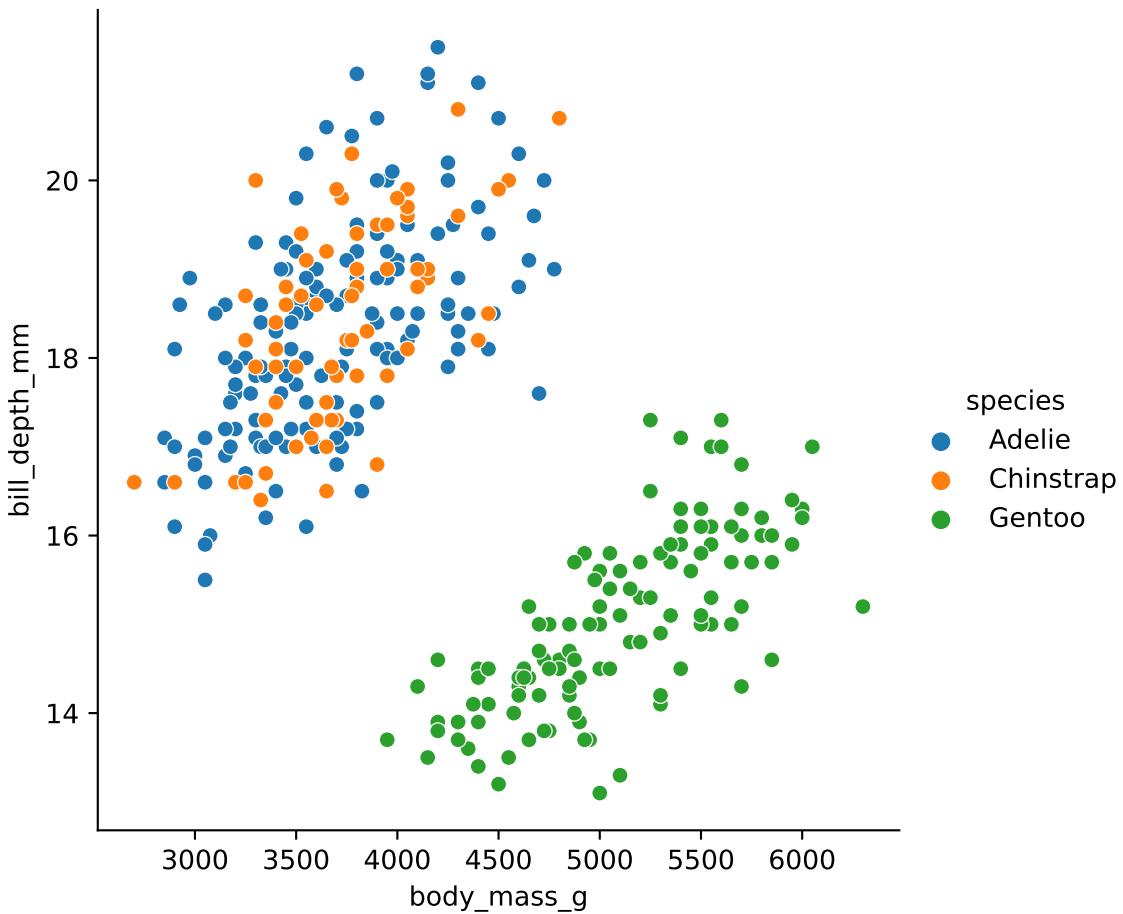
species		body_mass_g	bill_depth_mm
	body_mass_g	1.000000	0.576138
Chinstrap	bill_depth_mm	0.576138	1.000000
	body_mass_g	1.000000	0.604498
Gentoo	bill_depth_mm	0.604498	1.000000
	body_mass_g	1.000000	0.719085
	bill_depth_mm	0.719085	1.000000

[https://www.dropbox.com/s/1zlkjz3642u3tyo/Sectioon2\\_6\\_2.mp4?raw=1](https://www.dropbox.com/s/1zlkjz3642u3tyo/Sectioon2_6_2.mp4?raw=1)

Within each individual species, the correlation between the variables is strongly positive!

This is an example of **Simpson's paradox**. The reason for it can be seen from a scatter plot:

```
sns.relplot(data=penguins,
x=columns[0], y=columns[1],
hue="species"
);
```



Within each color, the positive association is clear. But what dominates the combination of all three species is the large difference between Gentoo and the others. Because the Gentoo are both larger and have shallower bills, the dominant relationship is negative.

As often happens in statistics, the precise framing of the question can strongly affect its answer. This can lead to honest mistakes by the naive as well as outright deception by the unscrupulous.

## Exercises

For these exercises, you may of course use computer help to work on a problem, but your answer should be self-contained without reference to computer output (unless stated otherwise).

**Exercise 2.1.** The following parts are about the sample set of  $n$  values  $x_1, \dots, x_n$ , where  $n > 2$ ,  $x_i = 0$  for  $i = 1, \dots, n - 1$ , and  $x_n = M$  for a positive number  $M$ .

- (a) Show that the sample mean is  $M/n$ .
- (b) Find the sample median. (You will have to describe multiple cases for  $n$ , even if they give the same result.)
- (c) Show that the corrected sample variance  $s_{n-1}^2$  is  $M^2/n$ .
- (d) Find the sample z-scores of all the  $x_i$  in terms of  $M$  and  $n$ .

**Exercise 2.2.** Suppose the samples  $x_1, \dots, x_n$  have the z-scores  $z_1, \dots, z_n$ .

- (a) Show that the sample z-scores satisfy  $\sum_{i=1}^n z_i = 0$ .
- (b) Show that the sample z-scores satisfy  $\sum_{i=1}^n z_i^2 = n - 1$ .

**Exercise 2.3.** For this exercise, apply the 2 outlier criterion to the sample set given in Exercise 2.1 using the sample mean and sample variance. (Again assume that  $n > 2$  and  $M > 0$ .)

- (a) Show that regardless of  $n$  and  $M$ , the value 0 is never an outlier.
- (b) Show that the value  $M$  is an outlier if  $n \geq 6$ , regardless of  $M$ .

**Exercise 2.4.** Define a population by

$$x_i = \begin{cases} 1, & 1 \leq i \leq 11, \\ 2, & 12 \leq i \leq 14, \\ 4, & 15 \leq i \leq 22, \\ 6, & 23 \leq i \leq 32. \end{cases}$$

- (a) Find the median of the population.
- (b) Find the smallest interval containing all non-outlier values according to the 1.5 IQR criterion.

**(b)** (old version) Which of the following values are outliers ?

$$-10, -5, 0, 5, 10, 15, 20$$

**Exercise 2.5.** Suppose that a population has values  $x_1, x_2, \dots, x_n$ . Define the function

$$r_2(x) = \sum_{i=1}^n (x_i - x)^2.$$

Show using calculus that  $r_2$  is minimized at  $x = \mu$ , the population mean.

**Exercise 2.6.** Suppose that  $n = 2k + 1$  and a population has values  $x_1, x_2, \dots, x_n$  in sorted order, so that the median is equal to  $x_k$ . Define the function

$$r_1(x) = \sum_{i=1}^n |x_i - x|.$$

(This function is called the *total absolute deviation* of  $x$  from the population.) Show that  $r_1$  has a global minimum at  $x = x_k$  by way of the following steps.

- (a)** Explain why the derivative of  $r_1$  is undefined at every  $x_i$ . Consequently, all of the  $x_i$  are critical points of  $r_1$ .
- (b)** Determine  $r'_1$  within each interval  $(-\infty, x_1), (x_1, x_2), (x_2, x_3)$ , and so on. Explain why this shows that there cannot be any additional critical points to consider. (Note: you can replace the absolute values with a piecewise definition of  $r_1$ , where the formula for the pieces changes as you cross over each  $x_i$ .)
- (c)** By considering the  $r'_1$  values between the  $x_i$ , explain why it must be that

$$r_1(x_1) > r_1(x_2) > \dots > r_1(x_k) < r_1(x_{k+1}) < \dots < r_1(x_n).$$

**Exercise 2.7.** Prove that two sample sets have a Pearson correlation coefficient equal to 1 if they have identical z-scores. (Hint: See Exercise 2.2.)

**Exercise 2.8.** Suppose that two sample sets satisfy  $y_i = -x_i$  for all  $i$ . Prove that the Pearson correlation coefficient between  $x$  and  $y$  equals  $-1$ .

# 3 Classification

Machine learning is the use of data to tune algorithms for making decisions or predictions. Unlike deduction based on reasoning from principles governing the application, machine learning is a “black box” that just adapts via training.

We divide machine learning into three major forms:

**Supervised learning** The training data only examples that include the answer (or **label**) we expect to get. The goals are to find important effects and/or to predict labels for previously unseen examples.

**Unsupervised learning** The data is unlabeled, and the goal is to discover structure and relationships inherent to the data set.

**Reinforcement learning** The data is unlabeled, but there are known rules and goals that can be encouraged through penalties and rewards.

We start with supervised learning, which can be subdivided into two major areas:

- **Classification**, in which the algorithm is expected to choose from among a finite set of options.
- **Regression**, in which the algorithm should predict the value of a quantitative variable.

Most algorithms for one of these problems have counterparts in the other.

## 3.1 Classification basics

A single training example or sample is characterized by a **feature vector**  $\mathbf{x}$  of  $d$  real numbers and a **label**  $y$  drawn from a finite set  $L$ . If  $L$  has only two members (say, “true” and “false”), we have a **binary classification** problem; otherwise, we have a **multiclass** problem.

When we have  $n$  training samples, it’s natural to collect them into columns of a **feature matrix**  $\mathbf{X}$  with  $n$  rows and  $d$  columns. Using subscripts to represent the indexes of the matrix, we can write

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1d} \\ X_{21} & X_{22} & \cdots & X_{2d} \\ \vdots & \vdots & & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{nd} \end{bmatrix}.$$

**!** Important

A 2D array or matrix has elements that are addressed by two subscripts. These are always given in order *row, column*.

In math, we usually start the row and column indexes at 1, but Python starts them at 0.

[https://www.dropbox.com/s/lqeka4m8p5q44k2/Section3\\_1.mp4?raw=1](https://www.dropbox.com/s/lqeka4m8p5q44k2/Section3_1.mp4?raw=1)

Each row of the feature matrix is a single feature vector, while each column is the value for a single feature over the entire training set.

**Example 3.1.** Suppose we want to train an algorithm to predict whether a basketball shot will score. For one shot, we might collect three coordinates to represent the launch point, three to represent the launch velocity, and three to represent the initial angular rotation (axis and magnitude). Thus each shot will require a feature vector of length 9. A collection of 200 sample shots would be encoded as a  $200 \times 9$  feature matrix.

We can also collect the associated training labels into the **label vector**

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

**!** Important

In linear algebra, the default shape for a vector is usually as a single column. In Python, a vector doesn't exactly have a row or column orientation, though when it matters, a row shape is usually preferred.

Each component  $y_i$  of the label vector is drawn from the label set  $L$ .

### 3.1.1 Encoding qualitative data

We have defined the features as numerical values. What should we do with qualitative data? We could arbitrarily assign numbers to possible values, such as “0=red”, “1=blue”,

“2=yellow,” and so on. But this is not ideal: most of the time, we would not want to say that yellow is twice as far from red as it is from blue!

A better strategy is to use the one-hot or dummy encoding. If a particular feature can take on  $k$  unique values, then we introduce  $k$  new features indicating which value is present. (We can use  $k - 1$  dummy features if we interpret all-zeros to mean the  $k$ th possibility.)

### 3.1.2 Walkthrough

The *scikit-learn* package `sklearn` is a collection of well-known machine learning algorithms and tools. Scikit-learn offers a uniform framework across all classifier types:

1. Define features and labels as numpy arrays or pandas frames.
2. Create a learner object, specifying any values that specialize the behavior.
3. Train the learner on the data by calling the `fit` method.
4. Apply the learner to a feature matrix/frame using the `predict` method.

The package also includes a few classic example datasets. We load one derived from automatic recognition of handwritten digits:

```
from sklearn import datasets
ds = datasets.load_digits()          # loads a well-known dataset
X, digits = ds["data"], ds["target"]    # assign feature matrix and label vector
print("The feature matrix has shape", X.shape)
print("The label vector has shape", digits.shape)
n, d = X.shape
print("there are", d, "features and", n, "samples")
```

[https://www.  
dropbox.com/  
s/  
zxel4113v15y5/  
Section3\\_1\\_  
2.mp4?raw=1](https://www.dropbox.com/s/zxel4113v15y5/Section3_1_2.mp4?raw=1)

```
The feature matrix has shape (1797, 64)
The label vector has shape (1797,)
there are 64 features and 1797 samples
```

The entries of `digits` are integer values 0 through 9, indicating the true value of the corresponding handwritten digit. Let’s consider the binary problem, “is this digit a 6?” That implies the following Boolean label vector:

```
y = (digits == 6)
print("Number of sixes in dataset:", sum(y))
```

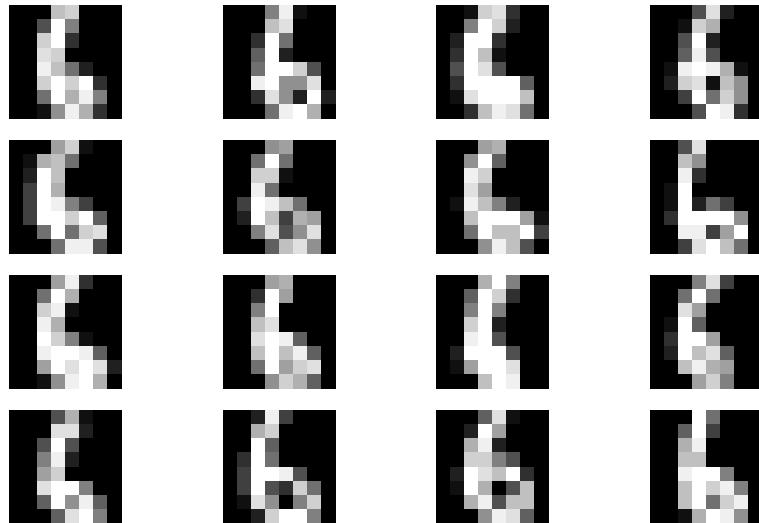
```
Number of sixes in dataset: 181
```

It so happens that the 64 features in the dataset are the pixel grayscale values from an  $8 \times 8$  bitmap of a handwritten digit. We can visualize the raw data. Here are some of the 6s, for example:

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

def plot_digits(X):
    fig, axes = plt.subplots(4,4)
    for i in range(4):
        for j in range(4):
            row = j + 4*i
            A = np.reshape(np.array(X[row,:]),(8,8))
            sns.heatmap(A,ax=axes[i,j],square=True,cmap="gray",cbar=False)
            axes[i,j].axis(False)
    return None

plot_digits(X[y])
```



The process of training a classifier is called **fitting**. We first have to import a particular classifier type, then create an instance of that type. Here, we choose one that we will study in a future section:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=20)      # specification of the model
```

Now we can fit the learner to the training data:

```
knn.fit(X, y) # training of the model  
  
KNeighborsClassifier(n_neighbors=20)
```

At this point, the classifier object `knn` has figured out what it needs from the training data. It has methods we can now call to make predictions and evaluate the quality of the results.

Each new prediction is for a **query vector** with 64 features. In practice, we can use a list in place of a vector for the query.

```
query = [20]*d # list with d copies of 20
```

The `predict` method of the classifier allows specifying multiple query vectors as rows of an array. In fact, it expects a 2D array in all cases, even if there is just one row.

```
Xq = [ query ] # 2D array with a single row
```

The result of the prediction will be a vector of labels, one per row of the query.

```
# Get vector of predictions:  
knn.predict(Xq)
```

```
array([False])
```

### ! Important

The `predict` method requires a vector or list of query vectors or lists and it outputs a vector of classes. This is true even if there is just a single query.

At the moment, we don't have any realistic query data at hand other than the training data. But we can investigate the question of how well the classifier does on that data, simply by using the feature matrix as the query:

```
# Get vector of predictions for the training set:  
yhat = knn.predict(X)
```

Now we simply count up the number of correctly predicted labels and divide by the total number of samples to get the **accuracy** of the classifier.

```
acc = sum(yhat == y) / n      # fraction of correct predictions
print(f"accuracy is {acc:.1%}")
```

```
accuracy is 99.9%
```

Not surprisingly, sklearn has functions for doing this measurement in fewer steps. The `metrics` module has functions that can compare true labels with predictions. In addition, each classifier object has a `score` method that allows you to skip finding the predictions vector yourself.

```
from sklearn.metrics import accuracy_score

# Compare original labels to predictions:
acc = accuracy_score(y, yhat)
print(f"accuracy score is {acc:.1%}")

# Same result, if we don't want to keep the predicted values around:
acc = knn.score(X, y)
print(f"knn score is {acc:.1%}")
```

```
accuracy score is 99.9%
knn score is 99.9%
```

Does this mean that the classifier is a good one? The raw number looks great, but that question is more subtle than you would expect.

## 3.2 Classifier performance

Let's return to the (previously cleaned) loan applications dataset.

```
import pandas as pd
loans = pd.read_csv("_datasets/loan_clean.csv")
loans.head()
```

	loan_amnt	int_rate	installment	annual_inc	dti	delinq_2yrs	delinq_amnt	percent_funded
0	5000	10.65	162.87	24000.0	27.65	0	0	100.0
1	2500	15.27	59.83	30000.0	1.00	0	0	100.0
2	2400	15.96	84.33	12252.0	8.72	0	0	100.0
3	10000	13.49	339.31	49200.0	20.00	0	0	100.0
4	3000	12.69	67.79	80000.0	17.94	0	0	100.0

We create a binary classification problem by labelling whether each loan was at least 95% funded. The other columns will form the features for the predictions.

```
X = loans.drop("percent_funded", axis=1)
y = loans["percent_funded"] > 95
```

[https://www.dropbox.com/s/6midrbwhjta9tjm/Section3\\_2.mp4?raw=1](https://www.dropbox.com/s/6midrbwhjta9tjm/Section3_2.mp4?raw=1)

### 3.2.1 Train–test paradigm

It seems desirable for a classifier to work well on the samples it was trained on. But we probably want to do more than that.

**Definition 3.1.** The performance of a predictor on previously unseen data is known as the **generalization** of the predictor.

In order to gauge generalization, we hold back some of the labeled data from training and use it only to test the performance. An `sklearn` helper function called `train_test_split` allows us to split off 20% of the data to use for testing. It's usually recommended to shuffle the order of the samples before the split, and in order to make the results reproducible, we give a specific random seed to the RNG used for the shuffle.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2,
    shuffle=True,
    random_state=3
)

print("There are", X_train.shape[0], "training samples.")
print("There are", X_test.shape[0], "testing samples.")
```

[https://www.dropbox.com/s/ofj6kirff2q1gkh/Section3\\_2\\_1.mp4?raw=1](https://www.dropbox.com/s/ofj6kirff2q1gkh/Section3_2_1.mp4?raw=1)

There are 31773 training samples.  
There are 7944 testing samples.

We can check that the test and train labels have similar characteristics:

```
import pandas as pd
print("labels in the training set:")
print( pd.Series(y_train).describe() )
```

```
print("\nlabels in the testing set:")
print( pd.Series(y_test).describe() )

labels in the training set:
count      31773
unique       2
top     True
freq      30351
Name: percent_funded, dtype: object

labels in the testing set:
count      7944
unique       2
top     True
freq      7575
Name: percent_funded, dtype: object
```

Now we train on the training data...

```
knn = KNeighborsClassifier(n_neighbors=9)
knn.fit(X_train, y_train)    # fit only to train set
```

```
KNeighborsClassifier(n_neighbors=9)
```

...and test on the rest.

```
acc = knn.score(X_test, y_test)    # score only on test set
print(f"test accuracy is {acc:.1%}")
```

```
test accuracy is 95.6%
```

This seems like a high accuracy, perhaps. But consider that the vast majority of loans were funded:

```
funded = sum(y)
print(f"{funded/len(y):.1%} were funded")
```

```
95.5% were funded
```

Therefore, an algorithm that simply “predicts” funding every single loan could do as well as the trained classifier!

```
from sklearn.metrics import accuracy_score
generous = [True]*len(y_test)
acc = accuracy_score(y_test, generous)
print(f"fund-them-all accuracy is {acc:.1%}")
```

```
fund-them-all accuracy is 95.4%
```

In this context, our trained classifier is not impressive at all. We need a metric other than accuracy.

### 3.2.2 Binary classifiers

A binary classifier is one that produces just two unique labels, which we call “yes” and “no” here. To fully understand the performance of a binary classifier, we have to account for four cases:

- True positives (TP): Predicts “yes”, actually is “yes”
- False positives (FP): Predicts “yes”, actually is “no”
- True negatives (TN): Predicts “no”, actually is “no”
- False negatives (FN): Predicts “no”, actually is “yes”

[https://www.  
dropbox.com/  
s/  
hxu66k8g2gqa06t/  
Section3\\_2\\_2.mp4?raw=1](https://www.dropbox.com/s/hxu66k8g2gqa06t/Section3_2_2.mp4?raw=1)

We often display these in a  $2 \times 2$  table according to the states of the prediction and *ground truth* (i.e., the given label in the dataset). The table can be filled with counts or percentages of tested instances to create a **confusion matrix**, as illustrated in Figure 3.1.

**Definition 3.2.** The following primary metrics are defined for a binary classifier:

accuracy	$\frac{TP + TN}{TP + FP + TN + FN}$
recall	$\frac{TP}{TP + FN}$
specificity	$\frac{TN}{TN + FP}$
precision	$\frac{TP}{TP + FP}$
negative predictive value (NPV)	$\frac{TN}{TN + FN}$

		prediction		
		yes	no	
ground truth	yes	<b>TP</b>	<b>FN</b>	recall
	no	<b>FP</b>	<b>TN</b>	specificity
		precision      NPV		

Figure 3.1: A confusion matrix. Correct predictions are on the diagonal.

**i** Note

Recall is also known as *sensitivity* or the *true positive rate*. Specificity is the *true negative rate*.

All of the primary metrics vary between 0 (worst) and 1 (best). Accuracy is self-explanatory; the other metrics answer the following questions:

- **recall** How often are actual “yes” cases predicted correctly?
- **specificity** How often are actual “no” cases predicted correctly?
- **precision** How often are the “yes” predictions correct?
- **NPV** How often are the “no” predictions correct?

**Example 3.2.** Here is a confusion matrix for a hypothetical test for COVID-19 antigens applied to 100 samples:

	Predicted +	Predicted -
Actually +	22	4
Actually -	12	62

From this we see that the accuracy is  $84/100$ , or 84%. Out of 26 samples that had the antigen, the test identified 22, for a recall of  $20/26 = 84.6\%$ . Out of 74 samples that did not have the antigen, 62 were predicted correctly, for a specificity of  $62/74 = 83.8\%$ . Finally, the precision is  $22/34 = 64.7\%$  and the NPV is  $62/66 = 93.9\%$ .

The metrics to pay attention to depend on the context and application. For a pregnancy test, for example, the health consequences of a false negative might be substantial, so the

manufacturer might aim mainly for a high recall rate. But a risk-averse loan officer would be most concerned about making loans to those who end up defaulting, i.e. a low false positive rate, and seek a high precision.

There are ways of combining the primary metrics in order to account for two at once.

**Definition 3.3.** The  **$F$  score** is the harmonic mean of the precision and the recall, i.e.,

$$F_1 = \left[ \frac{1}{2} \left( \frac{1}{\text{precision}} + \frac{1}{\text{recall}} \right) \right]^{-1} = \frac{2\text{TP}}{2\text{TP} + \text{FN} + \text{FP}}.$$

The **balanced accuracy** is the arithmetic mean of recall and specificity.

Like the primary metrics,  $F_1$  and balanced accuracy range between 0 (worst) and 1 (best). The harmonic mean is small if either of its terms is small, so a high  $F_1$  means both precision and recall are good.

**Example 3.3.** Continuing with the loan classifier trained earlier in this section, we can find the confusion matrix:

```
from sklearn.metrics import confusion_matrix
yhat = knn.predict(X_test)
C = confusion_matrix(y_test, yhat, labels=[True, False])
C

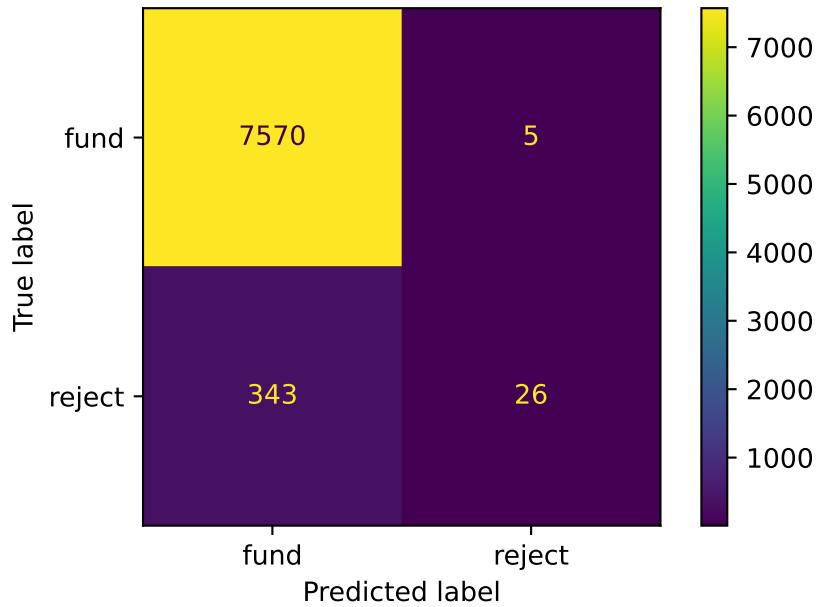
array([[7570,      5],
       [ 343,    26]])
```

### 🔥 Caution

It's advisable to always call `confusion_matrix` with the `labels` argument, even though it is optional, in order to control the ordering within the matrix. In particular, `False < True`, so the default for Boolean labels is to count the upper left corner of the matrix as "true negatives," assuming that `False` represents a negative result.

In order order to help keep track of what the entries mean, we can also make a picture of the confusion matrix:

```
from sklearn.metrics import ConfusionMatrixDisplay
lbl = ["fund", "reject"]
ConfusionMatrixDisplay(C, display_labels=lbl).plot();
```



Hence, there are 7570 true positives. The accuracy is

```
print( f"accuracy = {(7570 + 26) / np.sum(C):.1%}" )
```

```
accuracy = 95.6%
```

(In practice, of course, we can use the `accuracy` function we imported earlier.) Here are the other primary scores:

```
TP, FN, FP, TN = C.ravel()      # grab the 4 values in the confusion matrix
print( f"recall = {TP/(TP+FN):.1%}" )
print( f"specificity = {TN/(TN+FP):.1%}" )
print( f"precision = {TP/(TP+FP):.1%}" )
print( f"NPV = {TN/(TN+FN):.1%}" )

recall = 99.9%
specificity = 7.0%
precision = 95.7%
NPV = 83.9%
```

As noted above, few who ought to get a loan will go away disappointed, a loan officer might get nervous about the low specificity score that indicates bad loans.

In `sklearn.metrics` there are functions to compute recall and precision without reference to the confusion matrix. You must put the ground-truth labels before the predicted labels, and you should also specify which label value corresponds to a “positive” result. Swapping the “positive” role effectively swaps recall with specificity and precision with NPV.

```
from sklearn.metrics import precision_score, recall_score

for pos in [True, False]:
    print("With", pos, "as positive:")
    s = recall_score(y_test, yhat, pos_label=pos)
    print(f"    recall is {s:.3f}")
    s = precision_score(y_test, yhat, pos_label=pos)
    print(f"    precision is {s:.3f}")
    print()
```

With True as positive:

```
recall is 0.999
precision is 0.957
```

With False as positive:

```
recall is 0.070
precision is 0.839
```

There are also functions for the composite scores defined in Definition 3.3:

```
from sklearn.metrics import f1_score, balanced_accuracy_score

print( f"F1 = {f1_score(y_test, yhat):.1%}" )
print( f"balanced accuracy = {balanced_accuracy_score(y_test, yhat):.1%}" )
```

```
F1 = 97.8%
balanced accuracy = 53.5%
```

The loan classifier has excellent recall, respectable precision, and terrible specificity, resulting in a good  $F$  score and a low balanced accuracy score.

**Example 3.4.** If  $k$  of the  $n$  testing samples were funded loans, then the fund-them-all loan classifier has

$$TP = k, TN = 0, FP = n - k, FN = 0.$$

[https://www.  
dropbox.com/  
s/  
napiitmlezyqrq/  
Example3\\_3.  
mp4?raw=1](https://www.dropbox.com/s/napiitmlezyqrq/Example3_3.mp4?raw=1)

Its  $F$  score is thus

$$\frac{2\text{TP}}{2\text{TP} + \text{FN} + \text{FP}} = \frac{2k}{2k + n - k} = \frac{2k}{k + n}.$$

If the fraction of funded samples in the test set is  $k/n = a$ , then the accuracy of this classifier is  $a$ . Its  $F$  score is  $2a/(1+a)$ , which is larger than  $a$  unless  $a = 1$ . That's because the true positives greatly outweigh the other confusion matrix values.

The balanced accuracy is

$$\frac{1}{2} \left( \frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{TN} + \text{FP}} \right) = \frac{1}{2},$$

independently of  $a$ . This quantity is sensitive to the low specificity.

### 3.2.3 Multiclass classifiers

When there are more than two unique possible labels, these metrics can be applied using the **one-vs-rest** paradigm. For  $K$  unique labels, this paradigm poses  $K$  binary questions: “Is it in class 1, or not?”, “Is it in class 2, or not?”, etc. The confusion matrix becomes  $K \times K$ .

**Example 3.5.** We load a dataset on the characteristics of cars and use quantitative factors to predict the region of origin:

```
import seaborn as sns
cars = sns.load_dataset("mpg").dropna()
cars.head()
```

[https://www.  
dropbox.com/  
s/  
sk7gua9paulqmn6/  
Section3\\_2\\_  
3.mp4?raw=1](https://www.dropbox.com/s/sk7gua9paulqmn6/Section3_2_3.mp4?raw=1)

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chev
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylan
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth s
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel s
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino

Now we extract the quantitative features and labels:

```
features = ["cylinders", "horsepower", "weight", "acceleration", "mpg"]
X = cars[features]
y = pd.Categorical(cars["origin"])
```

```
print(X.shape[0], "samples and", X.shape[1], "features")
```

392 samples and 5 features

### 💡 Tip

It's not necessary to convert a vector of strings into a `Categorical` vector of labels, but it does make a few things handy in post-processing.

Next, we split into training and testing subsets:

```
X_train, X_test, y_train, y_test = train_test_split(  
    X, y,  
    test_size=0.2,  
    shuffle=True,  
    random_state=1  
)
```

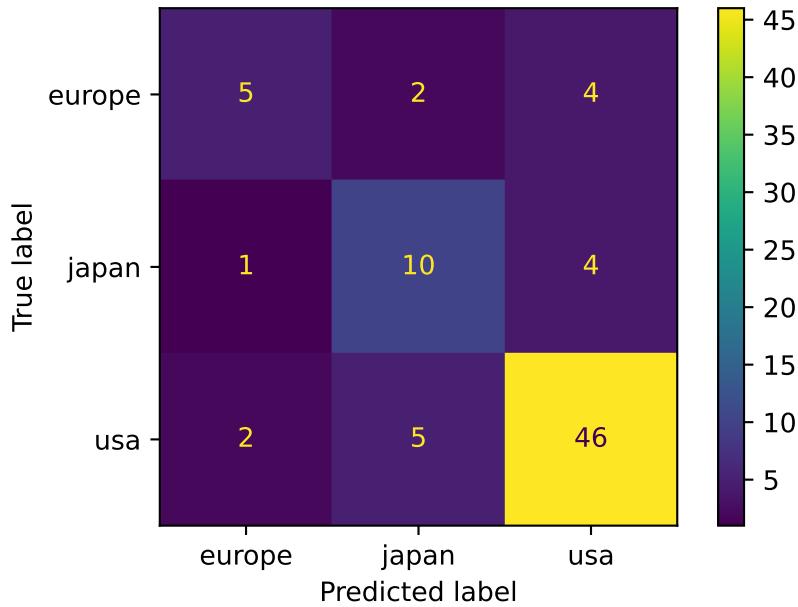
Now we perform the fit and measure the accuracy:

```
knn = KNeighborsClassifier(n_neighbors=8)  
knn.fit(X_train, y_train)  
yhat = knn.predict(X_test)  
print(f"accuracy is {accuracy_score(y_test, yhat):.1%}")
```

accuracy is 77.2%

Here is the confusion matrix:

```
labels = y.categories  
C = confusion_matrix(y_test, yhat, labels=labels)  
ConfusionMatrixDisplay(C, display_labels=labels).plot();
```



From the confusion matrix above we can see that, for example, out of 54 predictions of “usa” on the test set, there are 8 total false positives, in the sense that the actual labels were otherwise.

We also get  $K$  versions of the metrics like accuracy, recall,  $F$  score, and so on. We can get all the individual precision scores, say, automatically:

```
precisions = precision_score(y_test, yhat, average=None)
for (label,p) in zip(labels,precisions):
    print(f"{label}: {p:.1%}")
```

```
europe: 62.5%
japan: 58.8%
usa: 85.2%
```

To get a composite precision score, we have to specify an averaging method. The “macro” option simply takes the mean of the vector above.

```
mac = precision_score(y_test, yhat, average="macro")
print(mac)
```

```
0.6883623819898329
```

There are other ways to average performance scores over the classes, depending on whether poorly represented classes should be weighted more weakly than others.

[https://www.  
dropbox.com/  
s/  
p0s4kshrlnfcp0z/  
Example3\\_5.  
mp4?raw=1](https://www.dropbox.com/s/p0s4kshrlnfcp0z/Example3_5.mp4?raw=1)

### 3.3 Decision trees

A decision tree is much like playing Twenty Questions. A question is asked, and the answer reduces the possible results, leading to a new question. **CART** (Classification And Regression Tree), which we present here, is a popular method for systematizing the idea.

Given feature vectors  $\mathbf{x}_1, \dots, \mathbf{x}_n$  with labels  $y_1, \dots, y_n$ , the immediate goal is to partition the samples into two subsets, each of which has as uniform as set of labels as possible. The process is then repeated recursively: the subsets are bisected to make four subsets, and so on. These splits form a binary tree. When a prediction is required, we apply the same criteria as the splits in our tree, and when we reach a *leaf* of the tree (i.e., no further subdivisions), we take a vote of all the samples in the leaf subset to determine the label.

There are two details that need to be specified: what kind of subset splits to allow, and how to determine the uniformity of labels within each subset. We start with the latter.

#### 3.3.1 Gini impurity

**Definition 3.4.** Let  $k$  be an integer. The **indicator function**  $\mathbb{1}_k$  is the function on integers defined as

$$\mathbb{1}_k(t) = \begin{cases} 1, & \text{if } t = k, \\ 0, & \text{otherwise.} \end{cases}$$

Let  $S$  be any subset of samples, given as a list of indices into the original set. Suppose there are  $K$  unique labels, which we denote  $1, 2, \dots, K$ . Define the values

$$p_k = \frac{1}{|S|} \sum_{i \in S} \mathbb{1}_k(y_i), \quad k = 1, \dots, K, \tag{3.1}$$

where  $|S|$  is the number of elements in  $S$ . In words,  $p_k$  is the proportion of samples in  $S$  that have label  $k$ . Note that the sum over all the  $p_k$  equals 1.

**Definition 3.5.** The **Gini impurity** of set  $S$  is defined as

$$H(S) = \sum_{k=1}^K p_k(1 - p_k),$$

where  $p_k$  is defined in Equation 3.1.

[https://www.  
dropbox.com/  
s/  
giztxxn7q669ug0/  
Section3\\_3\\_  
1.mp4?raw=1](https://www.dropbox.com/s/giztxxn7q669ug0/Section3_3_1.mp4?raw=1)

If one of the  $p_k$  is 1, then the others are all zero and  $H(S) = 0$ . This is considered optimal—it indicates that all the labels in the set  $S$  are identical.

At the other extreme, if  $p_k = 1/K$  for all  $k$ , then

$$H(S) = \sum_{k=1}^K \frac{1}{K} \left(1 - \frac{1}{K}\right) = K \cdot \frac{1}{K} \cdot \frac{K-1}{K} = \frac{K-1}{K} < 1.$$

In general,  $H$  is always nonnegative and less than 1, and it only approaches 1 in the limit of a large number of equally distributed classes.

**Example 3.6.** Suppose a set  $S$  has  $n$  members with label A, 1 member with label B, and 1 member with label C. What is the Gini impurity of  $S$ ?

*Solution.* We have  $p_A = n/(n+2)$ ,  $p_B = p_C = 1/(n+2)$ . Hence

$$\begin{aligned} H(S) &= \frac{n}{n+2} \left(1 - \frac{n}{n+2}\right) + 2 \frac{1}{n+2} \left(1 - \frac{1}{n+2}\right) \\ &= \frac{n}{n+2} \frac{2}{n+2} + \frac{2}{n+2} \frac{n+1}{n+2} \\ &= \frac{4n+2}{(n+2)^2}. \end{aligned}$$

This value is 1/2 for  $n = 0$  and approaches zero as  $n \rightarrow \infty$ .

### 3.3.2 Partitioning

Suppose we start with a sample set  $S$  that we partition into disjoint (i.e., nonoverlapping) subsets  $S_L$  and  $S_R$ . We want to assign a total impurity to the partitioning so that we can compare different scenarios. One pitfall we should avoid is to put just one sample into  $S_L$  and all the rest into  $S_R$ , which would make  $H(S_L) = 0$  automatically and give an advantage. So we will choose a formula that rewards more evenly divided subsets.

**Definition 3.6.** The **total impurity** of the partition  $(S_L, S_R)$  is

$$Q(S_L, S_R) = |S_L| H(S_L) + |S_R| H(S_R).$$

If  $S_L$  and  $S_R$  are both pure subsets, then  $Q(S_L, S_R) = 0$ . Otherwise,  $Q$  tends to be larger in response to less purity and greater size in the subsets.

[https://www.  
dropbox.com/  
s/  
vaz7ymxfwjg79kt/  
Section3\\_3\\_  
2.mp4?raw=1](https://www.dropbox.com/s/vaz7ymxfwjg79kt/Section3_3_2.mp4?raw=1)

Our goal is to choose a partition that minimizes  $Q$ . However, we constrain ourselves to allow only certain kinds of partitions. If feature space is  $d$ -dimensional, we select a dimension  $1 \leq j \leq d$  and a real threshold value  $\theta$ . Then each feature vector  $\mathbf{x}$  is placed in  $S_L$  if  $x_j \leq \theta$

and in  $S_R$  if  $x_j > \theta$ . Geometrically, we are dividing feature space by an axis-aligned line if  $d = 2$  or an axis-aligned plane if  $d = 3$ . This splitting criterion can be evaluated extremely quickly, and we can find the best possible such partitioning in the sense of minimizing  $Q(S_L, S_R)$  in a reasonable amount of time.

**Example 3.7.** Suppose we have  $d = 1$  feature and are given the four samples  $(X_i, y_i) = \{(0, A), (1, B), (2, A), (3, B)\}$ . What is the optimal partition?

*Solution.* There are three allowable ways to partition the samples:

- $S_L = \{0\}, S_R = \{1, 2, 3\}$ . We have

$$H(S_L) = 0, \quad H(S_R) = (2/3)(1/3) + (1/3)(2/3) = 4/9.$$

Hence the total impurity for this partition is  $Q = (1)(0) + (3)(4/9) = 4/3$ .

- $S_L = \{0, 1\}, S_R = \{2, 3\}$ . Then

$$H(S_L) = H(S_R) = 2(1/2)(1/2) = 1/2,$$

and the total impurity is  $Q = (2)(1/2) + (2)(1/2) = 2$ .

- $S_L = \{0, 1, 2\}, S_R = \{3\}$ . This arrangement is the same as the first case with A and B swapped, so  $Q = 4/3$ .

Both  $x \leq 0$  and  $x \leq 2$  are equally good, and better than the alternative.

Finally, once we have found the optimal way to split  $S$  into  $(S_L, S_R)$ , we need to check whether it is an improvement—that is, whether

$$Q(S_L, S_R) + \alpha < Q(S, \emptyset) = |S|H(S),$$

where  $\alpha$  is an optional nonnegative number that requires a certain minimum amount of decrease. If the condition is satisfied, then  $S$  is not split and becomes a leaf of the decision tree. If so, then we add this split to the binary tree and recursively check both  $S_L$  and  $S_R$  for partitioning.

The above strategy is fast but can fail to get the best possible decision tree. That is, the optimal partition into 4 or 8 subsets might require you to look ahead in order to avoid a good-looking first partition that leads you astray. Because the optimization algorithm considers only one level at a time, we say it is a *greedy* approach.

### 3.3.3 Usage

In naive form, the decision tree construction continues to find partitions until every leaf in the tree represents a pure subset. In practice, we usually set a limit on the depth of the tree, which is the maximum number of partitions it takes to start from the root and reach any leaf. This obviously puts an upper limit on the computational time, but it is also desirable for other reasons we will explore in the next chapter.

**Example 3.8.** We create a toy dataset with 20 random points in the plane, with two subsets of 10 that are shifted left/right a bit. (The details are not important.) Here is how the set looks:

```
import numpy as np
import pandas as pd
from numpy.random import default_rng

rng = default_rng(1)
gp1 = rng.random((10,2))
gp1[:,0] -= 0.25
gp2 = rng.random((10,2))
gp2[:,0] += 0.25
X = np.vstack((gp1,gp2))
y = np.hstack(([1]*10,[2]*10))
print("feature matrix X:")
print(X)
print("\nlabel vector y:")
print(y)
```

feature matrix X:

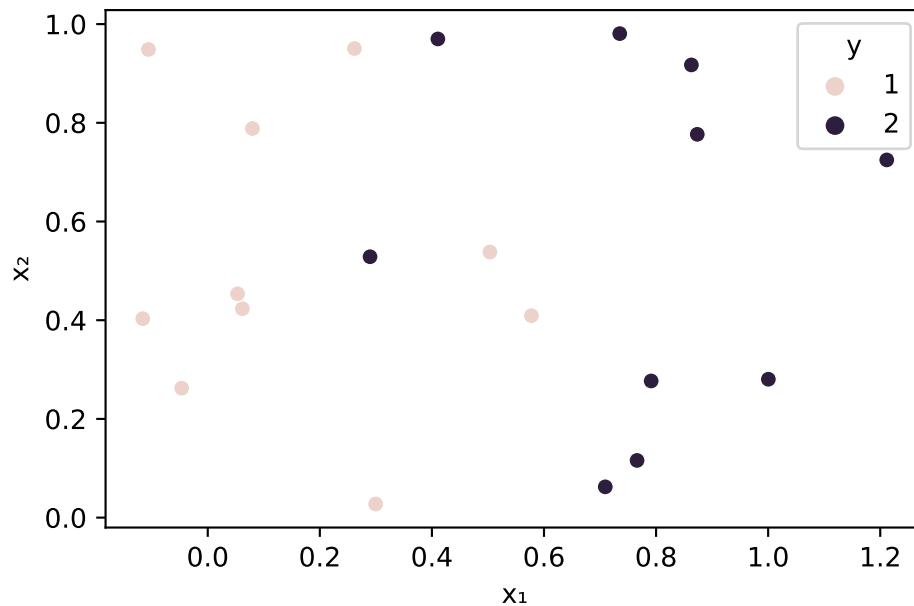
[ 0.26182162	0.9504637 ]
[-0.10584039	0.94864945]
[ 0.06183145	0.42332645]
[ 0.57770259	0.40919914]
[ 0.29959369	0.02755911]
[ 0.50351311	0.53814331]
[ 0.07973172	0.7884287 ]
[ 0.05319483	0.45349789]
[-0.1159583	0.40311299]
[-0.04654476	0.26231334]
[ 1.00036467	0.28040876]
[ 0.73519097	0.9807372 ]
[ 1.21165719	0.72478994]
[ 0.79122686	0.2768912 ]
[ 0.41065201	0.96992541]
[ 0.76606859	0.11586561]
[ 0.87348976	0.77668311]
[ 0.8630033	0.9172977 ]
[ 0.28959288	0.52858926]
[ 0.70933588	0.06234958]]

label vector y:

```
[1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2]
```

The features are the two point coordinates, giving a 20-by-2 feature matrix  $X$ , and the labels are a vector  $y$  of length 20. We will create a data frame for aiding in visualization:

```
import seaborn as sns
df = pd.DataFrame( {"x":X[:,0], "x":X[:,1], "y":y} )
sns.scatterplot(data=df, x="x", y="x", hue="y");
```



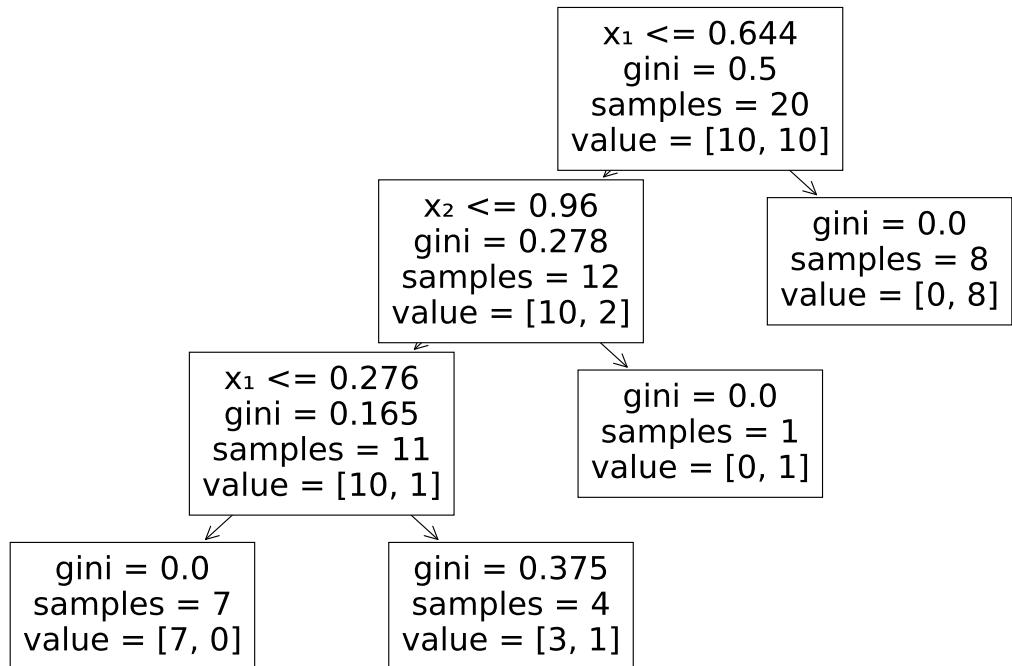
Now we create and fit a decision tree for these samples:

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
tree = DecisionTreeClassifier(max_depth=3)
tree.fit(X,y)
```

```
DecisionTreeClassifier(max_depth=3)
```

At this point, the `tree` object has created and stored all the information derived from the dataset that defines the decision tree. Since this is a small tree, we can easily look under the hood:

```
from matplotlib.pyplot import figure
figure(figsize=(18,11), dpi=160)
plot_tree(tree, feature_names=["x ", "x "]);
```



The root of the tree at the top shows that the best initial split was found at the vertical line  $x_1 = 0.644$ . To the right of that line is a Gini value of zero: 8 samples, all with label 2. That is, any future prediction by this tree will immediately return label 2 if its value for  $x_1$  exceeds 0.644. Otherwise, it moves to the left child node and tests whether  $x_2$  is greater than 0.96. As you can see from the scatterplot above, that horizontal line has a single sample with label 2 above it. And so on.

Notice that the bottom right node has a nonzero final Gini impurity. This node could be partitioned, but the classifier was constrained to stop at a depth of 3. If a prediction ends up here, then the classifier returns label 1, which is the most likely outcome.

Because we can follow a decision tree's logic step by step, we say it is highly **interpretable**. The transparency of the prediction algorithm is an attractive aspect of decision trees, although this advantage can weaken as the power of the tree is increased to handle difficult datasets.

**Example 3.9.** We return to the penguins and fit a decision tree to the quantitative features:

[https://www.dropbox.com/s/7piwuswj773unku/Example3\\_8.mp4?raw=1](https://www.dropbox.com/s/7piwuswj773unku/Example3_8.mp4?raw=1)

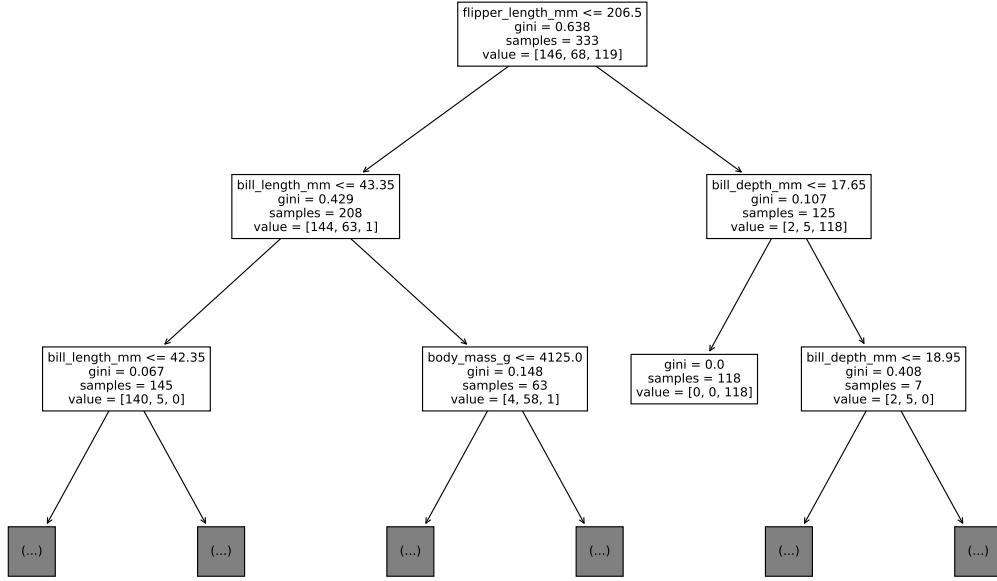
```
import pandas as pd
pen = sns.load_dataset("penguins")
pen = pen.dropna()
features = [
    "bill_length_mm",
    "bill_depth_mm",
    "flipper_length_mm",
    "body_mass_g"
]
X = pen[features]
y = pen["species"]

dt = DecisionTreeClassifier(max_depth=4)
dt.fit(X, y)
```

```
DecisionTreeClassifier(max_depth=4)
```

We get some interesting information from looking at the top levels of a decision tree trained on the full dataset:

```
from matplotlib.pyplot import figure
figure(figsize=(18,11), dpi=160)
plot_tree(dt, max_depth=2, feature_names=features);
```



The most determinative feature for identifying the species is apparently the flipper length. If it exceeds 206.5 mm, then the penguin is rather likely to be a Gentoo.

We can measure the relative importance of each feature by comparing their total contributions to reducing the Gini index:

```
pd.Series(dt.feature_importances_, index=features)
```

	0
bill_length_mm	0.367310
bill_depth_mm	0.065186
flipper_length_mm	0.553866
body_mass_g	0.013638

This ranking is known as **Gini importance**.

Flipper length alone accounts for about half of the resolving power of the tree, followed in importance by the bill length. The other measurements apparently have little discriminative value.

In order to assess the effectiveness of the tree, we use the train–test paradigm:

```

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True,
    random_state=0
)
dt.fit(X_train, y_train)

yhat = dt.predict(X_test)
print("Confusion matrix:")
print( confusion_matrix(
    y_test, yhat,
    labels=["Adelie", "Chinstrap", "Gentoo"]
) )
print("\nPerformance metrics:")
print( classification_report(y_test, yhat) )

```

Confusion matrix:

```

[[39  0  0]
 [ 2  8  0]
 [ 0  0 18]]

```

Performance metrics:

	precision	recall	f1-score	support
Adelie	0.95	1.00	0.97	39
Chinstrap	1.00	0.80	0.89	10
Gentoo	1.00	1.00	1.00	18
accuracy			0.97	67
macro avg	0.98	0.93	0.95	67
weighted avg	0.97	0.97	0.97	67

The performance is quite good, although the Chinstrap case is hindered by the relatively low number of training examples:

```
y_train.value_counts()
```

species	
Adelie	107
Gentoo	101
Chinstrap	58

Decision trees can depend sensitively on the sample locations; a small change might completely rewrite large parts of the tree, in which case interpretability becomes less clear.

[https://www.  
dropbox.com/  
s/  
mrdr38t2tzpv6kq/  
Example3\\_9.  
mp4?raw=1](https://www.dropbox.com/s/mrdr38t2tzpv6kq/Example3_9.mp4?raw=1)

## 3.4 Nearest neighbors

The next learner type is conceptually simple: given a point in feature space to classify, survey the nearest known examples and choose the most frequently occurring class. This is called the *k nearest neighbors* (kNN or *k*-NN) algorithm, where *k* is the number of neighboring examples to survey.

### 3.4.1 Distances and norms

The existence of “closest” examples means that we need to define a notion of distance in feature spaces of any dimension. Let  $\mathbb{R}^d$  be the space of vectors with *d* real components, and let  $\mathbf{0}$  be the vector of all zeros.

**Definition 3.7.** A **distance metric** is a function *dist* on pairs of vectors that satisfies the following properties for all vectors  $\mathbf{u}$ ,  $\mathbf{v}$ , and  $\mathbf{z}$ :

1.  $\text{dist}(\mathbf{u}, \mathbf{v}) \geq 0$ ,
2.  $\text{dist}(\mathbf{u}, \mathbf{v}) = 0$  if and only if  $\mathbf{u} = \mathbf{v}$ ,
3.  $\text{dist}(\mathbf{u}, \mathbf{v}) = \text{dist}(\mathbf{v}, \mathbf{u})$ , and
4.  $\text{dist}(\mathbf{u}, \mathbf{v}) \leq \text{dist}(\mathbf{u}, \mathbf{z}) + \text{dist}(\mathbf{z}, \mathbf{v})$ , known as the triangle inequality.

These are considered the essential properties desired of a distance metric. We will define a distance metric by using a function on vectors known as a *norm*.

**Definition 3.8.** For any vector  $\mathbf{u} \in \mathbb{R}^d$ , we define the following norms.

- The **2-norm** or *Euclidean norm*:

$$\|\mathbf{u}\|_2 = (u_1^2 + u_2^2 + \cdots + u_d^2)^{1/2}.$$

- The **1-norm** or *Manhattan norm*:

$$\|\mathbf{u}\|_1 = |u_1| + |u_2| + \cdots + |u_d|.$$

- The  **$\infty$ -norm**, *max norm*, or *Chebyshev norm*:

$$\|\mathbf{u}\|_\infty = \max_{1 \leq i \leq d} |u_i|.$$

Given a norm, the distance between vectors  $\mathbf{u}$  and  $\mathbf{v}$  is defined by

$$\text{dist}(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|.$$

[https://www.  
dropbox.com/  
s/  
rscyxjwazq4ol9v/  
Section3\\_4\\_  
1.mp4?raw=1](https://www.dropbox.com/s/rscyxjwazq4ol9v/Section3_4_1.mp4?raw=1)

**Example 3.10.** Given  $\mathbf{u} = [-1, 1, 0, 4]$  and  $\mathbf{v} = [-2, 1, 2, 2]$ , find the distance between them using all three common norms.

*Solution.* We first calculate  $\mathbf{u} - \mathbf{v} = [-3, 0, -2, 2]$ . Then

$$\begin{aligned}\|\mathbf{u} - \mathbf{v}\|_2 &= (3^2 + 0^2 + 2^2 + 2^2)^{1/2} = \sqrt{17}, \\ \|\mathbf{u} - \mathbf{v}\|_1 &= 3 + 0 + 2 + 2 = 7, \\ \|\mathbf{u} - \mathbf{v}\|_\infty &= \max\{3, 0, 2, 2\} = 3.\end{aligned}$$

The Euclidean norm generalizes ordinary geometric distance in  $\mathbb{R}^2$  and  $\mathbb{R}^3$  and is usually considered the default. One of its most important features is that  $\|\mathbf{x}\|_2^2$  is a differentiable function of the components of  $\mathbf{x}$ .

#### i Note

When  $\|\cdot\|$  is used with no subscript, it's usually meant to be the 2-norm, but it can also mean a generic, unspecified norm.

### 3.4.2 Algorithm

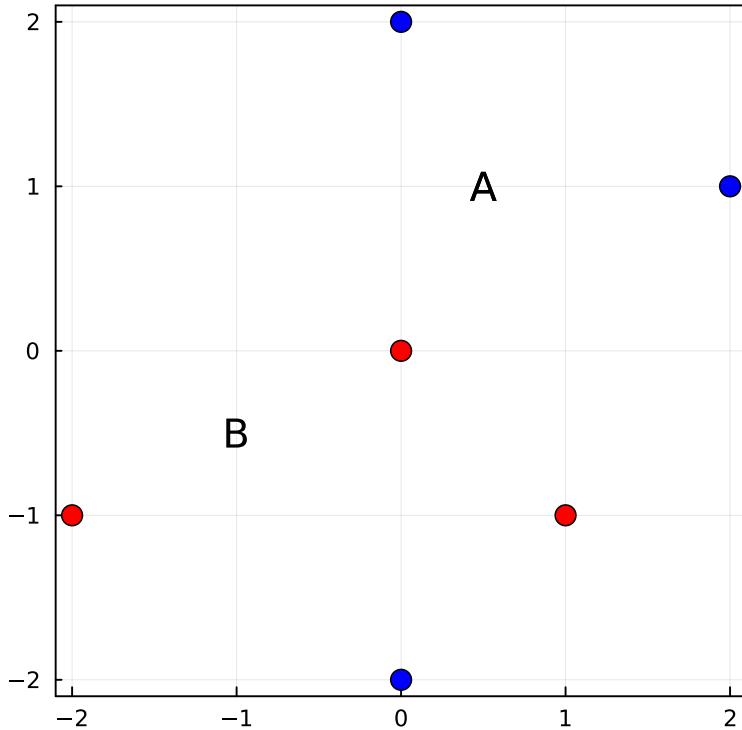
As data, we are given labeled samples  $\mathbf{x}_1, \dots, \mathbf{x}_n$  in  $\mathbb{R}^d$ .

**Definition 3.9.** Given a new query vector  $\mathbf{x}$ , the **kNN algorithm** finds the  $k$  labeled samples closest to  $\mathbf{x}$  and chooses the most frequently occurring label among them. Ties are broken randomly.

#### i Note

The *nearest neighbor search* is a well-studied problem in computer science, and there are specialized data structures used for its efficient solution.

**Example 3.11.** Here are 6 sample points, labeled blue and red, in 2-dimensional feature space:



Using inf-norm distance and  $k = 3$ , find the kNN labels at the locations marked A and B.

*Solution.* At point A, the nearest samples are at  $(0, 2)$  with distance 1,  $(0, 0)$  with distance 1, and  $(2, 1)$  with distance 1.5. By a 2-1 vote, the point should be labeled blue.

At point B, the nearest samples are at  $(0, 0)$  with distance 1,  $(-2, -1)$  with distance 1, and  $(0, -2)$  with distance 1.5. By a 2-1 vote, the point should be labeled red.

Note that the blue sample point at  $(0, -2)$  is its own closest neighbor, but the next two nearest neighbors are red. Therefore, in kNN with  $k = 3$ , the red points will outvote it and predict red! As always, we should not expect perfect performance on the training set.

kNN effectively divides up the feature space into domains that are dominated by nearby instances. The boundaries between those domains, called **decision boundaries**, are usually fairly complicated, as shown in the animation below for 2-norm distance.

[\\_media/knn\\_demo.mp4](#)

At  $k = 1$  neighbor, each sample point defines its own local domain of influence that gives way when reaching a point equally close to a differently-labeled sample. This typically produces the most complicated decision boundaries. At the other extreme, with  $k = n$

neighbors, all the samples vote every time, so all of feature space is given the same label (pending tiebreakers).

**Example 3.12.** Back to the penguins! We use `dropna` to drop any rows with missing values.

```
import seaborn as sns
import pandas as pd
penguins = sns.load_dataset("penguins")
penguins = penguins.dropna()
penguins.head(6)
```

[https://www.  
dropbox.com/  
s/  
l153u6gi1i42qes/  
Section3\\_4\\_  
2.mp4?raw=1](https://www.dropbox.com/s/l153u6gi1i42qes/Section3_4_2.mp4?raw=1)

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female
5	Adelie	Torgersen	39.3	20.6	190.0	3650.0	Male
6	Adelie	Torgersen	38.9	17.8	181.0	3625.0	Female

The data set has four quantitative columns that we use as features, and the species name is the label.

```
features = [
    "bill_length_mm",
    "bill_depth_mm",
    "flipper_length_mm",
    "body_mass_g"
]
X = penguins[features]
y = penguins["species"]
```

Each type of classifier has to be imported before its first use in a session. (Importing more than once does no harm.)

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X, y)
```

```
KNeighborsClassifier()
```

We can manually find the neighbors of a new vector. However, we have to make the query in the form of a data frame, since that is how the training data was provided. Here we make a query frame for values very close to the ones in the first row of the data.

```
vals = [39, 19, 180, 3750]
query = pd.DataFrame([vals], columns=features)
dist, idx = knn.kneighbors(query)
idx[0]

array([ 0, 143,  53, 100, 153])
```

The result above indicates that the first sample (index 0) was the closest, followed by four others. We can look up the labels of these points:

```
y[idx[0]]
```

species	
0	Adelie
143	Adelie
53	Adelie
100	Adelie
153	Chinstrap

By a vote of 4–1, then, the classifier should choose Adelie as the result at this location.

```
knn.predict(query)

array(['Adelie'], dtype=object)
```

Note that points can be outvoted by their neighbors. In other words, the classifier won't necessarily be correct on every training sample. For example:

```
print("Predicted:")
print( knn.predict(X.iloc[:5,:]) )
print()
print("Data:")
print( y.iloc[:5].values )
```

```
Predicted:
['Adelie' 'Adelie' 'Chinstrap' 'Adelie' 'Chinstrap']
```

Data:

```
['Adelie' 'Adelie' 'Adelie' 'Adelie' 'Adelie']
```

Next, we split into training and test sets to gauge the performance of the classifier. The `classification_report` function creates a summary of some of the important metrics.

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report,confusion_matrix

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=302
)
knn.fit(X_train,y_train)

yhat = knn.predict(X_test)
print("Confusion matrix:")
print( confusion_matrix(
    y_test, yhat,
    labels=["Adelie", "Chinstrap", "Gentoo"]
) )
print("\nPerformance metrics:")
print( classification_report(y_test, yhat) )
```

Confusion matrix:

```
[[34  2  2]
 [ 8  6  1]
 [ 1  0 13]]
```

Performance metrics:

	precision	recall	f1-score	support
Adelie	0.79	0.89	0.84	38
Chinstrap	0.75	0.40	0.52	15
Gentoo	0.81	0.93	0.87	14
accuracy			0.79	67
macro avg	0.78	0.74	0.74	67
weighted avg	0.79	0.79	0.77	67

[https://www.  
dropbox.com/  
s/  
yj9d51w4e67l5bw/  
Example3\\_  
12.mp4?raw=  
1](https://www.dropbox.com/s/yj9d51w4e67l5bw/Example3_12.mp4?raw=1)

### Tip

The default norm in the kNN learner is the 2-norm. To use the 1-norm instead, add `metric="manhattan"` to the classifier construction call.

### 3.4.3 Standardization

The values in the columns of the penguin frame in Example 3.12 are scaled quite differently. In particular, the values in the body mass column are more than 20x larger than the other columns on average:

```
X.mean()
```

	0
bill_length_mm	43.992793
bill_depth_mm	17.164865
flipper_length_mm	200.966967
body_mass_g	4207.057057

Consequently, the mass feature will dominate the distance calculations. To remedy this issue, we could transform the data into z-scores:

```
Z = X.transform( lambda x: (x - x.mean()) / x.std() )
```

We could then retrain the classifier using `Z` in place of `X`. A nuisance of doing so is that the standardization transformation must also be performed on every new query vector that comes along, requiring us to keep track of the mean and std of the original dataset. Scikit-learn allows us to automate this process by creating a **pipeline**, which makes it easy to chain together a data transformation followed by a learner.

**Example 3.13.** Once created, a pipeline object can mostly be treated the same as any other learner:

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler    # converts to z-scores

knn = KNeighborsClassifier(n_neighbors=5)
pipe = make_pipeline(StandardScaler(), knn)
pipe.fit(X_train, y_train)

Pipeline(steps=[('standardscaler', StandardScaler()),
               ('kneighborsclassifier', KNeighborsClassifier())])
```

In this case, standardization allows us to perform perfectly!

```
yhat = pipe.predict(X_test)

print("Confusion matrix:")
print( confusion_matrix(
    y_test, yhat,
    labels=["Adelie", "Chinstrap", "Gentoo"]
) )
print("\nPerformance metrics:")
print( classification_report(y_test, yhat) )
```

Confusion matrix:

```
[[38  0  0]
 [ 0 15  0]
 [ 0  0 14]]
```

Performance metrics:

	precision	recall	f1-score	support
Adelie	1.00	1.00	1.00	38
Chinstrap	1.00	1.00	1.00	15
Gentoo	1.00	1.00	1.00	14
accuracy			1.00	67
macro avg	1.00	1.00	1.00	67
weighted avg	1.00	1.00	1.00	67

We can look under the hood of the pipeline. For example, we can see that the mean and variance of each of the original data columns is stored in the first part of the pipeline:

```
print( pipe[0].mean_ )
print( pipe[0].var_ )

[ 44.18759398  17.01503759 202.04135338 4266.72932331]
[3.00582295e+01 3.98263101e+00 1.94114831e+02 6.53423137e+05]
```

However, we don't need to access that data just to use the pipeline. That's taken care of when we use `pipe.score` or `pipe.predict`.

[https://www.  
dropbox.com/  
s/  
x7br8qha5gcfsz4/  
Example3\\_  
13.mp4?raw=  
1](https://www.dropbox.com/s/x7br8qha5gcfsz4/Example3_13.mp4?raw=1)

## 3.5 Probabilistic interpretation

Both kNN and decision trees base classification on a voting procedure—for kNN, the  $k$  nearest neighbors cast votes, and for a decision tree, the values at a leaf cast votes. So far, we have interpreted the voting results in a winner-takes-all sense, i.e., the class with the most votes wins. But that interpretation discards a lot of potentially valuable information.

**Definition 3.10.** Let  $\mathbf{x}$  be a query vector in a vote-based classification method. The **probability vector**  $\hat{p}(\mathbf{x})$  is the vector of vote fractions received by each class.

**Example 3.14.** Suppose we have trained a kNN classifier with  $k = 10$  for data with three classes, called A, B, and C, and that the votes at the testing points are as follows:

	A	B	C
0	9	0	1
1	5	3	2
2	6	1	3
3	2	0	8
4	4	5	1

The values of  $\hat{p}$  over the test set form a  $5 \times 3$  matrix:

```
p_hat = np.array( [
    [0.9, 0, 0.1],
    [0.5, 0.3, 0.2],
    [0.6, 0.1, 0.3],
    [0.2, 0, 0.8],
    [0.4, 0.5, 0.1]
] )
```

[https://www.  
dropbox.com/  
s/  
oxil4vxd891ve8v/  
Section3\\_5.  
mp4?raw=1](https://www.dropbox.com/s/oxil4vxd891ve8v/Section3_5.mp4?raw=1)

It's natural to interpret  $\hat{p}$  as predicting the probability of each label at any query point, since the values are nonnegative and sum to 100%. Given  $\hat{p}$ , we can still output a predicted class; it's just that we also get additional information about how the prediction was made.

**Example 3.15.** Consider the penguin species classification problem:

```
penguins = sns.load_dataset("penguins").dropna()
# Select only numeric columns for features:
X = penguins.loc[:, penguins.dtypes=="float64"]
y = penguins["species"].astype("category")
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=5
)
```

We can train a kNN classifier and then retrieve the probabilities via `predict_proba`:

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
p_hat = knn.predict_proba(X_test)
p_hat[:6,:]

array([[0.8, 0.2, 0. ],
       [0.8, 0.2, 0. ],
       [0., 0., 1. ],
       [0., 0., 1. ],
       [0.8, 0.2, 0. ],
       [0.6, 0.4, 0.]])
```

From the output above we see that, for example, while the third and fourth test cases led to unanimous votes for *Gentoo*, the sixth case is deemed *Adelie* in a 3–2 squeaker (or is it a squawker?):

```
yhat = knn.predict(X_test)
yhat[:6]

array(['Adelie', 'Adelie', 'Gentoo', 'Gentoo', 'Adelie', 'Adelie'],
      dtype=object)
```

### 3.5.1 ROC curve

In the binary label case, our assumption so far has been that a simple majority vote determines a positive outcome. But we could choose a different threshold—a supermajority, for example, if we want to reduce false positives. This way of thinking leads us to a new way to fine-tune classification methods.

**Definition 3.11.** Let  $\theta$  be a number in the interval  $[0, 1]$ . We say that a class  $T$  **hits** at level  $\theta$  at a query point if the fraction of votes that  $T$  receives at that point is at least  $\theta$ .

**Example 3.16.** Continuing with the data in Example 3.14, we find that at  $\theta = 0$ , everything always hits:

	A	B	C
0	1	1	1
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1

At  $\theta = 0.05$ , say, we lose all the cases where no votes were received:

	A	B	C
0	1	0	1
1	1	1	1
2	1	1	1
3	1	0	1
4	1	1	1

At  $\theta = 0.15$ , we have also lost all those receiving 1 out of 10 votes:

	A	B	C
0	1	0	0
1	1	1	1
2	1	0	1
3	1	0	1
4	1	1	0

By the time we get to  $\theta = 0.7$ , there are only two hits left:

	A	B	C
0	1	0	0
1	0	0	0
2	0	0	0
3	0	0	1
4	0	0	0

The probability vector  $\hat{p}(\mathbf{x})$  holds the largest possible  $\theta$  values for which each class hits at  $\mathbf{x}$ . Looking at it another way,  $\theta = 0$  represents maximum credulity—everybody's a winner!—while  $\theta = 1$  represents maximum skepticism—unanimous winners only, please.

The **ROC curve** is a way to visualize the hits as a function of  $\theta$  over a fixed testing set. The idea is to tally, at each value of  $\theta$ , all the hits within each class that represent true positives and false positives, and present the results visually.

[https://www.  
dropbox.com/  
s/  
km6h1w1c7hprzww/  
Section3\\_5\\_  
1.mp4?raw=1](https://www.dropbox.com/s/km6h1w1c7hprzww/Section3_5_1.mp4?raw=1)

**i Note**

The name of the ROC curve is a throwback to the early days of radar, when the idea was first developed.

**Example 3.17.** We continue with the data from Example 3.14, but now we add ground truth to the queries:

	A	B	C	truth
0	9	0	1	A
1	5	3	2	B
2	6	1	3	A
3	2	0	8	C
4	4	5	1	A

Let's look at class A. At  $\theta = 0.05$ , class A hits in every case, giving TP=3 and FP=2. At  $\theta = 0.25$ , the fourth query drops out; we still have TP=3, but now FP=1. Here is the table of all the unique values of TP and FP that we can achieve as  $\theta$  varies between 0 and 1:

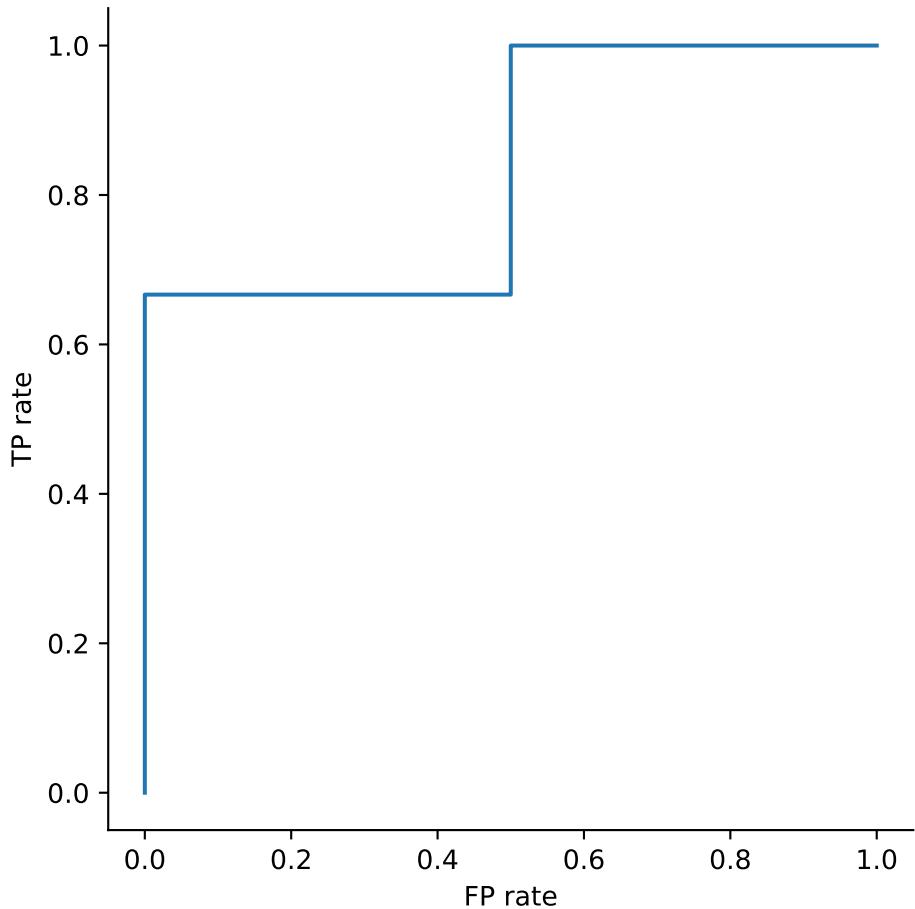
	theta	FP	TP
0	0.05	2	3
1	0.25	1	3
2	0.45	1	2
3	0.55	0	2
4	0.65	0	1
5	0.95	0	0

In order to make a graph, we convert the raw TP and FP numbers to rates. Since there are 2 positive and 3 negative over the entire test set, we can represent the rows above as the points

$$\left(\frac{2}{3}, \frac{3}{3}\right), \left(\frac{1}{2}, \frac{3}{3}\right), \left(\frac{1}{2}, \frac{2}{3}\right), \left(\frac{0}{2}, \frac{2}{3}\right), \left(\frac{0}{2}, \frac{1}{3}\right) \left(\frac{0}{2}, \frac{0}{3}\right).$$

The ROC curve for class A is just connect-the-dots for these points:

```
data = pd.DataFrame({"FP rate": [1, 1/2, 1/2, 0, 0, 0], "TP rate": [1, 1, 2/3, 2/3, 1/3, 0]})  
sns.relplot(data=data,  
            x="FP rate", y="TP rate",  
            kind="line", estimator=None  
);
```



As we're about to see, the step-by-step process above is just for illustration and is completely automated in practice.

Unsurprisingly, `sklearn` can compute the points defining the ROC curve automatically, which greatly simplifies drawing them. In a multiclass problem with  $K$  classes, there are  $K$  implied binary classification versions of one-vs-rest, so there are  $K$  curves to draw, one for the identification of each class.

**Example 3.18.** Continuing Example 3.15, we will plot ROC curves for the three species in the penguin data:

```
from sklearn.metrics import roc_curve

p_hat = knn.predict_proba(X_test)
results = []
for i, label in enumerate(knn.classes_):
```

```

actual = (y_test==label)
fp, tp, theta = roc_curve(actual,p_hat[:,i])
results.extend( [(label,fp,tp) for fp,tp in zip(fp,tp)] )
roc = pd.DataFrame( results, columns=["label","FP rate","TP rate"] )
roc

```

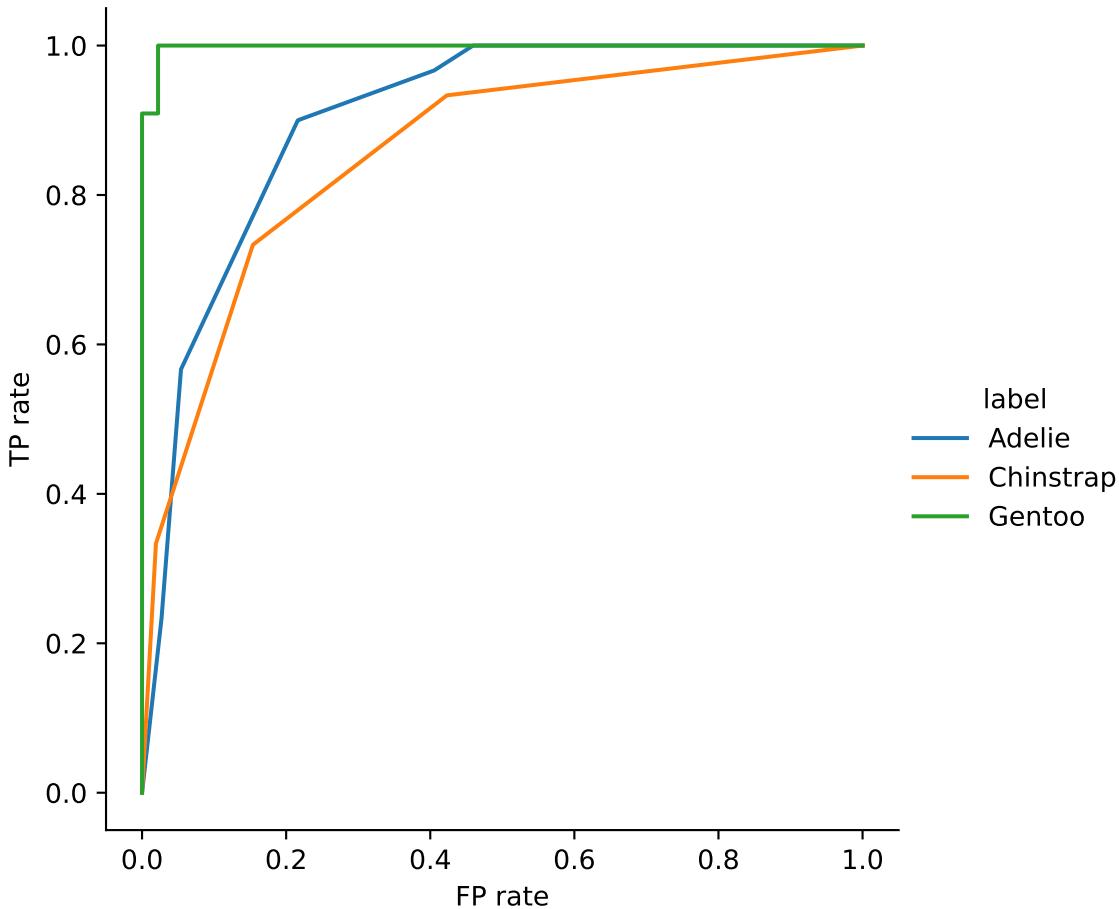
	label	FP rate	TP rate
0	Adelie	0.000000	0.000000
1	Adelie	0.027027	0.233333
2	Adelie	0.054054	0.566667
3	Adelie	0.216216	0.900000
4	Adelie	0.405405	0.966667
5	Adelie	0.459459	1.000000
6	Adelie	1.000000	1.000000
7	Chinstrap	0.000000	0.000000
8	Chinstrap	0.019231	0.333333
9	Chinstrap	0.153846	0.733333
10	Chinstrap	0.423077	0.933333
11	Chinstrap	1.000000	1.000000
12	Gentoo	0.000000	0.000000
13	Gentoo	0.000000	0.909091
14	Gentoo	0.022222	0.909091
15	Gentoo	0.022222	1.000000
16	Gentoo	0.088889	1.000000
17	Gentoo	0.200000	1.000000
18	Gentoo	1.000000	1.000000

The table above holds all of the key points on the ROC curves:

```

sns.relplot(data=roc,
             x="FP rate", y="TP rate",
             hue="label", kind="line", estimator=None
            );

```



Each curve starts in the lower left corner and ends at the upper right corner. The ideal situation is in the top left corner of the plot, corresponding to perfect recall and specificity. All of the curves explicitly show the tradeoff between recall and specificity as the decision threshold is varied. The *Gentoo* curve comes closest to the ideal.

If we weight neighbors' votes inversely to their distances from the query point, then the thresholds aren't restricted to multiples of  $\frac{1}{5}$ :

```

knnw = KNeighborsClassifier(n_neighbors=5, weights="distance")
knnw.fit(X_train, y_train)
p_hat = knnw.predict_proba(X_test)

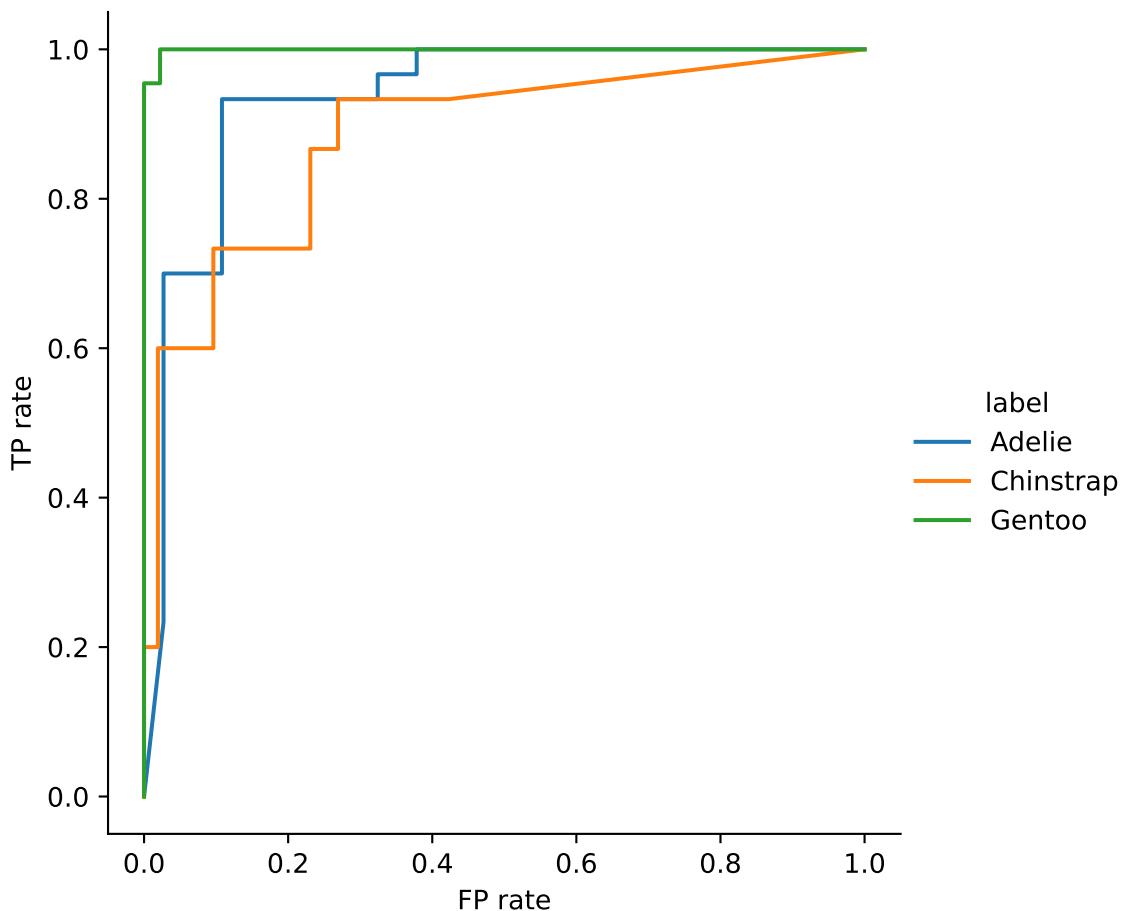
results = []
for i, label in enumerate(knn.classes_):
    actual = (y_test==label)
    fp, tp, theta = roc_curve(actual,p_hat[:,i])

```

```

results.extend( [(label,fp,tp) for fp,tp in zip(fp,tp)] )
roc = pd.DataFrame( results, columns=["label","FP rate","TP rate"] )
sns.relplot(data=roc,
             x="FP rate", y="TP rate",
             hue="label", kind="line", estimator=None
            );

```



### 3.5.2 AUC

ROC curves lead to another classification performance metric known as **area under ROC curve (AUC)**. Its name tells you exactly what it is, and it ranges between 0 (bad) and 1 (ideal). Unlike the other classification metrics we have encountered, AUC tries to account

[https://www.dropbox.com/s/82dsuclyo76r0ut/Example3\\_18.mp4?raw=1](https://www.dropbox.com/s/82dsuclyo76r0ut/Example3_18.mp4?raw=1)

not just for the result of the classification at a single threshold, but over the full range from credulous to skeptical. You might think of it as grading with partial credit.

**Example 3.19.** The AUC metric allows us to compare the standard and weighted kNN classifiers from Example 3.18. Note that the function for computing them, `roc_auc_score`, requires a keyword argument when there are more than two classes, to specify “one vs. rest” (our usual) or “one vs. one” matchups.

```
from sklearn.metrics import roc_auc_score
s = roc_auc_score(
    y_test, knn.predict_proba(X_test),
    multi_class="ovr", average=None
)

sw = roc_auc_score(
    y_test, knnw.predict_proba(X_test),
    multi_class="ovr", average=None
)

pd.DataFrame(
    {"standard": s, "weighted": sw},
    index=knn.classes_
)
```

	standard	weighted
Adelie	0.903153	0.935586
Chinstrap	0.857051	0.883333
Gentoo	0.997980	0.998990

Based on the above scores, the weighted classifier seems to be better at identifying all three species.

## Exercises

For these exercises, you may use computer help to work on a problem, but your answer should be self-contained without reference to computer output (unless stated otherwise).

**Exercise 3.1.** Here is a confusion matrix for a classifier of meme dankness.

<i>True label</i>	dank	648	78
	not dank	45	1004
		dank	not dank
		<i>prediction</i>	

Considering *dank* to be the positive outcome, calculate the (a) recall, (b) precision, (c) specificity, (d) accuracy, and (e)  $F_1$  score of the classifier.

**Exercise 3.2.** Here is a confusion matrix for a classifier of ice cream flavors.

<i>true flavor</i>	vanilla	75	10	4
	chocolate	8	163	6
	strawberry	11	22	24
		vanilla	chocolate	strawberry
		<i>prediction</i>		

(a) Calculate the recall rate for *chocolate*.

(b) Find the precision for *vanilla*.

(c) Find the accuracy for *strawberry*.

**Exercise 3.3.** Find the Gini impurity of this set:

$$\{A, B, B, C, C, C\}.$$

**Exercise 3.4.** Use the definition of Gini impurity to prove that it is never negative and always less than 1.

**Exercise 3.5.** Given  $x_i = i$  for  $i = 0, \dots, 5$ , with labels

$$y_1 = y_5 = y_6 = A, \quad y_2 = y_3 = y_4 = B,$$

find an optimal partition threshold using Gini impurity.

**Exercise 3.6.** Using 1-norm, 2-norm, and  $\infty$ -norm, find the distance between the given vectors.

(a)  $\mathbf{u} = [2, 3, 0]$ ,  $\mathbf{v} = [-2, 2, 1]$

(b)  $\mathbf{u} = [0, 1, 0, 1, 0]$ ,  $\mathbf{v} = [1, 1, 1, 1, 1]$

**Exercise 3.7.** (a) Prove that for any  $\mathbf{u} \in \mathbb{R}^d$ ,  $\|\mathbf{u}\|_\infty \leq \|\mathbf{u}\|_2$ .

(b) Prove that for any  $\mathbf{u} \in \mathbb{R}^d$ ,  $\|\mathbf{u}\|_2 \leq \sqrt{d} \|\mathbf{u}\|_\infty$ .

**Exercise 3.8.** Carefully sketch the set of all points in  $\mathbb{R}^2$  whose 1-norm distance from the origin equals 1. This is a *Manhattan unit circle*.

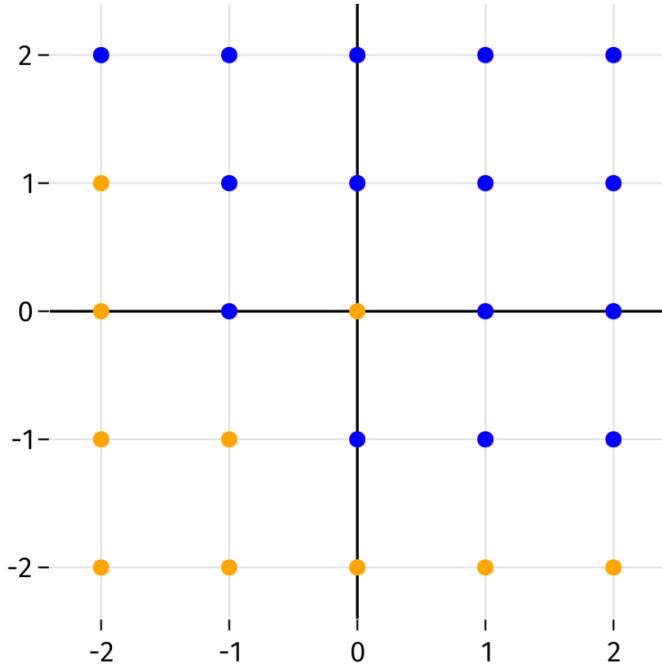
**Exercise 3.9.** Three points in the plane lie at the vertices of an equilateral triangle. One is labeled A and the other two are B. Carefully sketch the decision boundary for  $k$ -nearest neighbors with  $k = 1$ , using (a) the 2-norm and (b) the infinity norm.

**Exercise 3.10.** Define 8 points on an ellipse by  $x_k = a \cos(\theta_k)$  and  $y_k = b \sin(\theta_k)$ , where  $a$  and  $b$  are positive and

$$\theta_1 = \frac{\pi}{4}, \theta_2 = \frac{\pi}{2}, \theta_3 = \frac{3\pi}{4}, \dots, \theta_8 = 2\pi.$$

Let  $u_1, \dots, u_8$  and  $v_1, \dots, v_8$  be the z-scores of the  $x_k$  and the  $y_k$ , respectively. Show that the points  $(u_k, v_k)$  all lie on a circle centered at the origin for all  $k = 1, \dots, 8$ . (By extension, standardizing points into z-scores is sometimes called *sphereing* them.)

**Exercise 3.11.** Here are blue/orange labels on an integer lattice.



Let  $\hat{f}(x_1, x_2)$  be the kNN probabilistic classifier with  $k = 4$ , Euclidean metric, and mean averaging that returns the probability of a blue label. In each case below, a function  $g(t)$  is

defined from values of  $\hat{f}$  along a vertical or horizontal line. Carefully sketch a plot of  $g(t)$  for  $2 \leq t \leq 2$ .

- (a)  $g(t) = \hat{f}(1.2, t)$
- (b)  $g(t) = \hat{f}(t, -0.75)$
- (c)  $g(t) = \hat{f}(t, 1.6)$
- (d)  $g(t) = \hat{f}(-0.25, t)$

## 4 Model selection

```
import numpy as np
from numpy.random import default_rng
import pandas as pd
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix, f1_score, balanced_accuracy_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_curve, roc_auc_score
```

We have barely scratched the surface of the universe of classification algorithms. Even just the two types we have seen, nearest neighbors and decision trees, have multiple variations and options available through *hyperparameters*.

**Definition 4.1.** A **hyperparameter** of a learning algorithm is a value or setting affecting the algorithm that remains fixed throughout training.

**i** Note

In ML, a *parameter* is a value that is adjusted during training; i.e., it is learned from the training data. In most of mathematics, we would refer to these as *variables*, but in ML that term is often understood to be synonymous with *feature*.

[https://www.dropbox.com/s/zzly25dlcm5dg53/Section4\\_0.mp4?raw=1](https://www.dropbox.com/s/zzly25dlcm5dg53/Section4_0.mp4?raw=1)

Some hyperparameters, such as the choice of norm in the nearest-neighbors algorithm, have an influence that is not easy to characterize. But others clearly affect the potential expressive power of the algorithm.

**Example 4.1.** The maximum depth  $r$  of a decision tree limits the complexity that the tree can attain. When  $r = 1$ , the tree can divide the data only once and assign different values to the different sides. In general, though, a tree can assign up to  $2^r$  unique values, which

grows exponentially with  $r$ ; in fact, any training set of that size or smaller can be modeled with 100% training accuracy.

For a kNN classifier, when  $k$  is as large as the number of samples, the classifier can only take one value on the entire set—all the samples have a vote everywhere. The other extreme is  $k = 1$ , where each sample rules within its own neighborhood, and again we achieve 100% training accuracy.

Options provide flexibility but also demand rationales for their use. How can we choose the best hyperparameters for a given problem? And how do we choose the best algorithm overall? In order to answer these questions, we must first understand what to expect from the results of a learner in general terms.

## 4.1 Bias–variance tradeoff

When we train a classifier, we use a particular set of training data. In a different parallel universe, we might have been handed a different training set drawn from the same overall population. While we might be optimistic and hope for receiving the best-case training set, it's more prudent to consider what happens in the average case.

### 4.1.1 Learner bias

Suppose that  $f(x)$  is a perfect labeller, i.e., a function with 100% accuracy over an entire population. For simplicity, we can imagine that  $f$  is a binary classifier, i.e.,  $f(x) \in \{0, 1\}$ , although this assumption is not essential.

Let  $\hat{f}(x)$  denote a probabilistic classification function obtained after training. It depends on the particular training set we used. Suppose there are  $N$  total possible training sets, leading to labelling functions

$$\hat{f}_1(x), \hat{f}_2(x), \dots, \hat{f}_N(x).$$

Then we define the **expected value** of the classifier as the average over all training sets:

$$\mathbb{E} [\hat{f}(x)] = \frac{1}{N} \sum_{i=1}^N \hat{f}_i(x).$$

#### Note

Except on toy problems, we don't know how to calculate this average. This is more of a thought experiment. But we will simulate the idea later on.

The term *expected* doesn't mean that we anticipate getting this answer for our particular  $\hat{f}$ . It's just what we would get if we could average over all parallel universes receiving unique training sets.

We can apply the expectation operator  $\mathbb{E}$  to any function of  $x$ . In particular, the expected error in our own universe's prediction is

$$\begin{aligned}\mathbb{E} [f(x) - \hat{f}(x)] &= \frac{1}{N} \sum_{i=1}^N (f(x) - \hat{f}_i(x)) \\ &= \frac{1}{N} \left( \sum_{i=1}^N f(x) \right) - \frac{1}{N} \left( \sum_{i=1}^N \hat{f}_i(x) \right) \\ &= f(x) - \mathbb{E} [\hat{f}(x)].\end{aligned}$$

We will set  $y = f(x)$  as the true label and  $\hat{y} = \mathbb{E} [\hat{f}(x)]$  as the expected prediction. The quantity above,  $y - \hat{y}$ , is called the **bias** of the classifier. Bias depends on the particular algorithm and its hyperparameters, but not on the training set. Among other things, bias accounts for the fact that any particular finite algorithm can represent only some labelling functions perfectly.

### 4.1.2 Variance

It might seem as though the only important goal is to minimize the bias. To see why this is not the case, imagine that you are playing a hole of golf where the green lies on an island at the end of the fairway. You're capable of landing the ball on the green in one swing, but it's near the upper end of your range, and the penalty for landing in the water instead is severe. You might be better off playing it safe by just approaching the water's edge, which is a shot you can make much more reliably. On average over many attempts, you may well get a better score from the aggressive strategy, but the safe strategy gives you a more reliable result and better odds of doing pretty well, though not optimally.

In essence, a good chance of a mediocre result can outweigh a small chance of a better result. To express this tradeoff mathematically, we can compute the variance of the predicted label at any  $x$ :

$$\begin{aligned}\mathbb{E} [(y - \hat{f}(x))^2] &= \frac{1}{N} \sum_{i=1}^N (y - \hat{f}_i(x))^2 \\ &= \frac{1}{N} \sum_{i=1}^N (y - \hat{y} + \hat{y} - \hat{f}_i(x))^2 \\ &= \frac{1}{N} \sum_{i=1}^N (y - \hat{y})^2 + \frac{1}{N} \sum_{i=1}^N (\hat{y} - \hat{f}_i(x))^2 \\ &\quad + 2(y - \hat{y}) \cdot \frac{1}{N} \sum_{i=1}^N (\hat{y} - \hat{f}_i(x)).\end{aligned}$$

Now we find something interesting:

$$\frac{1}{N} \sum_{i=1}^N (\hat{y} - \hat{f}_i(x)) = \hat{y} - \frac{1}{N} \sum_{i=1}^N \hat{f}_i(x) = 0,$$

by the definition of  $\hat{y}$ . So overall,

$$\begin{aligned} \mathbb{E}[(y - \hat{f}(x))^2] &= \frac{1}{N} \sum_{i=1}^N (y - \hat{y})^2 + \frac{1}{N} \sum_{i=1}^N (\hat{y} - \hat{f}_i(x))^2 \\ &= (y - \hat{y})^2 + \mathbb{E}[(\hat{y} - \hat{f}(x))^2] \end{aligned} \quad (4.1)$$

[https://www.dropbox.com/s/h7siuimkbyio4fj/Section4\\_1\\_2.mp4?raw=1](https://www.dropbox.com/s/h7siuimkbyio4fj/Section4_1_2.mp4?raw=1)

The first term is the squared bias. The second is the **variance** of the learning method. In words, the variance of the learning process has two contributions:

**Bias** How close is the average prediction to the ground truth?

**Variance** How close to the average prediction is any one prediction likely to be?

```

rng = default_rng(302)
x, y, bias, var = [], [], [], []
x.extend(rng.normal(0.04, 0.08, 40))
y.extend(rng.normal(-0.03, 0.06, 40))
bias.extend(["low"]*40)
var.extend(["low"]*40)
x.extend(rng.normal(0.55, 0.11, 40))
y.extend(rng.normal(-0.35, 0.05, 40))
bias.extend(["high"]*40)
var.extend(["low"]*40)
x.extend(rng.normal(-0.02, 0.34, 40))
y.extend(rng.normal(0.03, 0.33, 40))
bias.extend(["low"]*40)
var.extend(["high"]*40)
x.extend(rng.normal(-0.25, 0.33, 40))
y.extend(rng.normal(-0.35, 0.33, 40))
bias.extend(["high"]*40)
var.extend(["high"]*40)
points = pd.DataFrame({"bias": bias, "variance": var, "x": x, "y": y})
fig = sns.relplot(data=points, x="x", y="y", row="variance", col="bias", aspect=1, height=6)
fig.set(xlim=(-1.25, 1.25), ylim=(-1.25, 1.25));

```

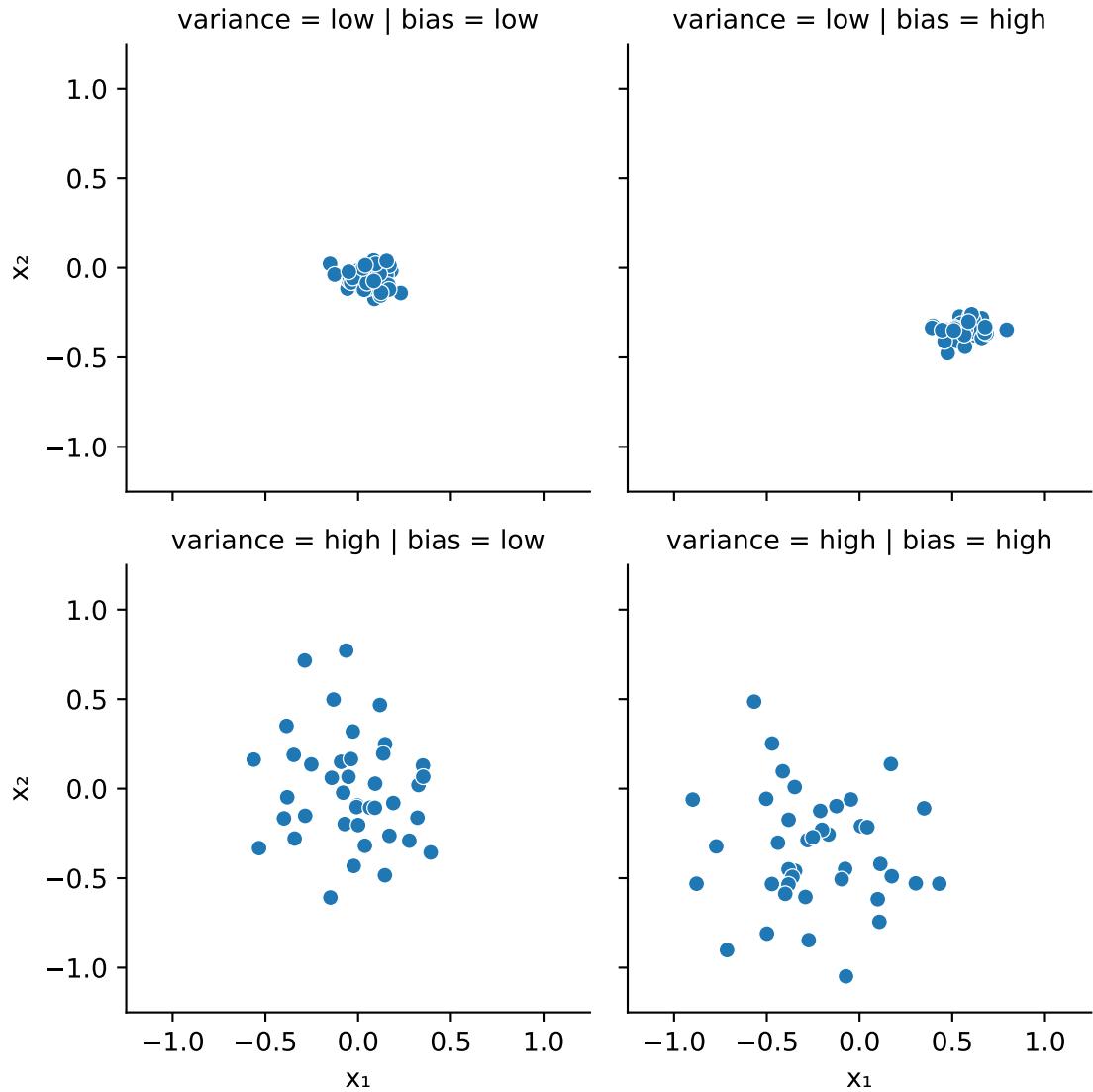


Figure 4.1: Bias versus variance (imagine you are aiming at the center of the box)

Why would these two factors be in opposition? When a learning method has the capacity to capture complex behavior, it potentially has a low bias. However, that same capacity means that the learner will fit itself very well to each individual training set, which increases the potential for variance over the whole collection of training sets.

This tension is known as the *bias-variance tradeoff*. Perhaps we can view this tradeoff as a special case of *Occam's Razor*: it's best to choose the least complex method necessary to reach a particular level of explanatory power.

### 4.1.3 Learning curves

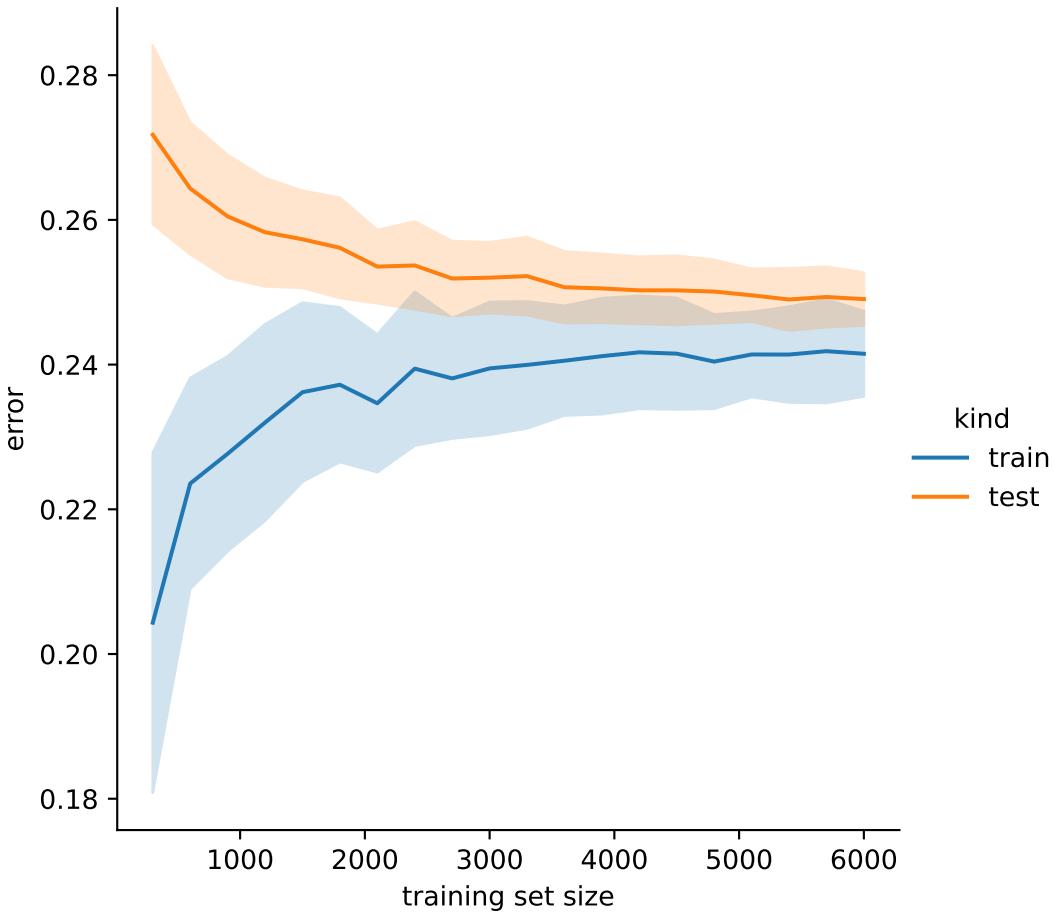
We can illustrate the tradeoff between bias and variance by running an artificial experiment with different sizes for the training datasets.

**Example 4.2.** We will use a subset of a realistic data set used to predict the dominant type of tree in patches of forest. We train a decision tree classifier with fixed depth throughout. (Don't confuse the forest data for the tree classifier, haha.)

```
forest = datasets.fetch_covtype()
X = forest["data"][:250000,:8]    # 250,000 samples, 8 dimensions
y = forest["target"][:250000]
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.05,
    shuffle=True, random_state=0
)

alln = range(300, 6001, 300)      # sizes of the training subsets
results = []                      # for tracking results
tree = DecisionTreeClassifier(max_depth=3)
for n in alln:                   # iterate over training set sizes
    for i in range(100):          # iterate over training sets
        X_train, y_train = shuffle(X_train, y_train, random_state=10*i)
        XX, yy = X_train[:n,:], y_train[:n]      # training subset of size n
        tree.fit(XX, yy)
        results.append( ("train", n, 1-tree.score(XX, yy)) )
        results.append( ("test", n, 1-tree.score(X_test, y_test)) )

cols = [ "kind", "training set size", "error" ]
results = pd.DataFrame(results, columns=cols)
sns.relplot(data=results,
            x=cols[1], y=cols[2],
            kind="line", errorbar="sd", hue=cols[0]
);
```



The plot above shows **learning curves**. The solid line is the mean result over all trials, and the ribbon has a width of one standard deviation. For each small training set, the tree has more than enough resolving power and adapts itself too well to the training data, leading to a large gap with the testing error (i.e., a failure to generalize past the training set). You can also see large variance (ribbon widths) in the results at the smaller sizes. As the training set size increases, we observe the training error increasing, because the task of reproducing the training set is now harder. The testing errors decrease, though, because the model has been fit to a more representative subset of the data. The gap between training and testing closes as more training data is used and the model is pushed to its limits.

Note that the curves seem to approach a horizontal asymptote at a nonzero level of error. This level indicates an unavoidable bias for this tree size, no matter how much of the data we throw at it. As a simple analogy, think about approximating curves in the plane by a parabola. You will be able to do a perfect job for linear and quadratic functions, but if you approximate one period of a cosine curve, you can't do a great job no matter how much information you have.

When you see a large gap between training and test errors, you should suspect that the learner will not generalize well. Ideally, you could bring more data to the table, perhaps by artificially augmenting the training examples. If not, you might as well decrease the resolving power of your learner, because the excess power is likely to make things no better, and maybe worse.

[https://www.  
dropbox.com/  
s/  
90rxrc78e7cg0hn/  
Example4\\_2.  
mp4?raw=1](https://www.dropbox.com/s/90rxrc78e7cg0hn/Example4_2.mp4?raw=1)



## 4.2 Overfitting

One important factor we have not yet considered is noise in the training data—that is, erroneous values. If a learner responds too adeptly to isolated wrong values, it will also respond incorrectly to other nearby inputs. This situation is known as **overfitting**.

### 4.2.1 Overfitting in kNN

To illustrate overfitting, let's use a really simple classification problem: a single feature, with the class being the sign of the feature's value. (We arbitrarily assign zero to have class +1.)

Consider first a kNN classifier with  $k = 1$ . The class assigned to each value is just that of the nearest training example, making for a piecewise constant labelling. Here are the results for four different training sets, each of size 40:

As you can see above, all four results are quite good. The only errors are for queries near zero.

[https://www.  
dropbox.com/  
s/  
0x8jpxmc1zw184/  
Section4\\_2\\_  
1.mp4?raw=1](https://www.dropbox.com/s/0x8jpxmc1zw184/Section4_2_1.mp4?raw=1)

Now suppose we use training sets that have just 3 mislabeled examples each. Here are some resulting classifiers:

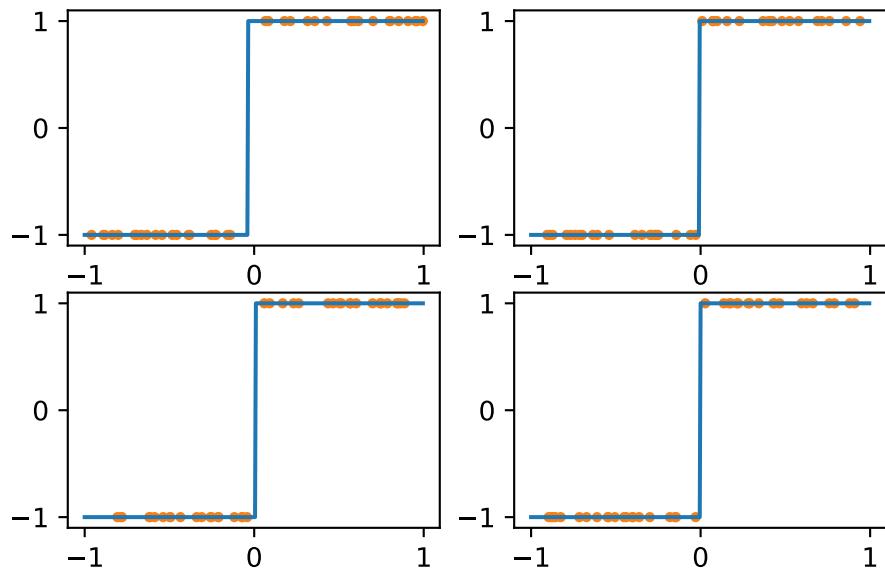


Figure 4.2: kNN with  $k=1$  and perfect data

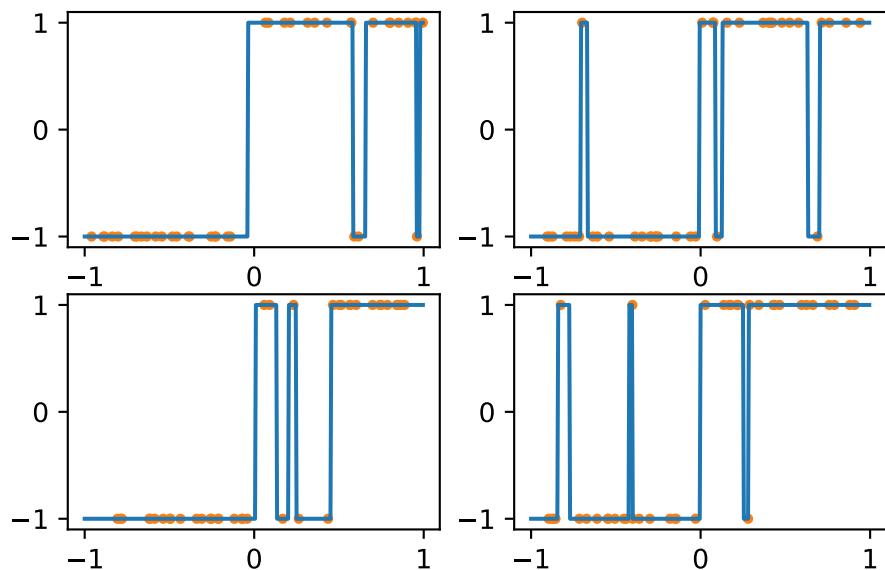


Figure 4.3: kNN with  $k=1$  and noisy data

Every sample is its own nearest neighbor, so this classifier responds to noisy data by reproducing it perfectly, which interferes with the larger trend we actually want to capture. We can generally expect such overfitting with  $k = 1$ , for which the decision boundary can be complex.

Now let's bump up to  $k = 3$ . The results are more like we want, even with noisy data:

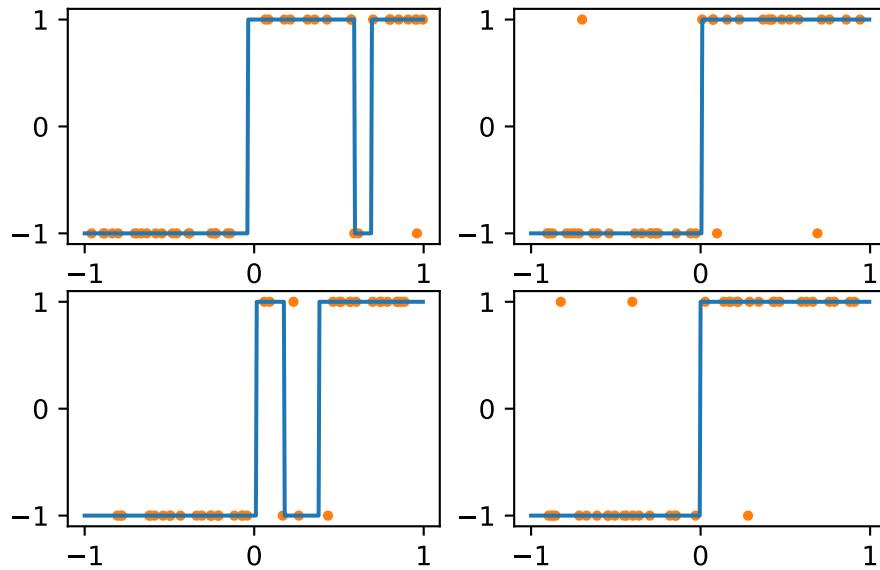


Figure 4.4: kNN with  $k=3$  and noisy data

The voting mechanism of kNN allows the classifier to ignore isolated bad examples. If we continue to  $k = 7$ , then the 3 outliers will never be able to outvote the correct values:

### 🔥 Caution

The lesson here is not simply that “bigger  $k$  is better.” In the case of  $k = 21$  above, for example, the classifier will predict the same value everywhere, which we could describe as *underfitting* the data.

## 4.2.2 Overfitting in decision trees

As mentioned in Example 4.1, the depth of a decision tree correlates with its ability to divide the samples more finely. For  $n = 40$  values, a tree of depth 6 is guaranteed to reproduce every sample value perfectly. Thus, with noisy data, we see clear signs of overfitting:

Using a shallower tree reduces the extent of overfitting:

[https://www.dropbox.com/s/c9j46ab7a0ypxaz/Section4\\_2\\_2.mp4?raw=1](https://www.dropbox.com/s/c9j46ab7a0ypxaz/Section4_2_2.mp4?raw=1)

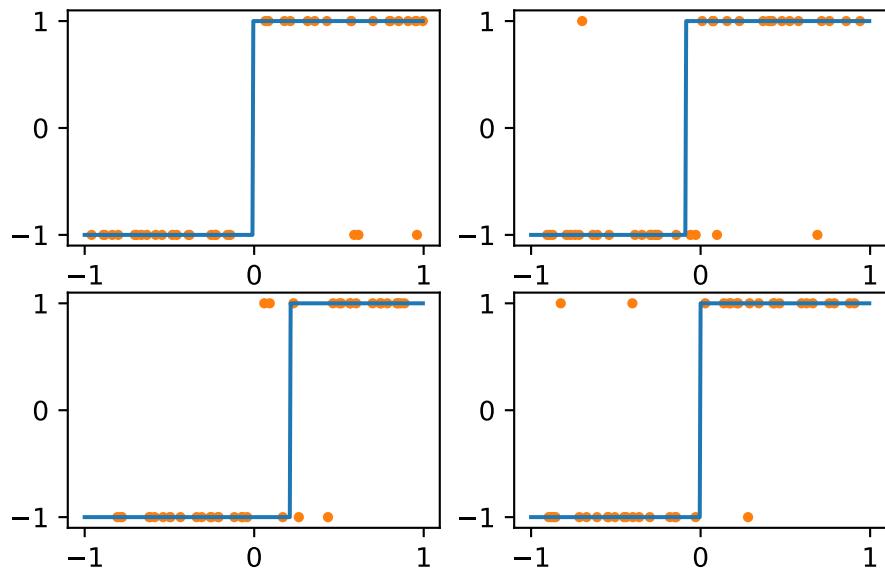


Figure 4.5: kNN with  $k=7$  and noisy data

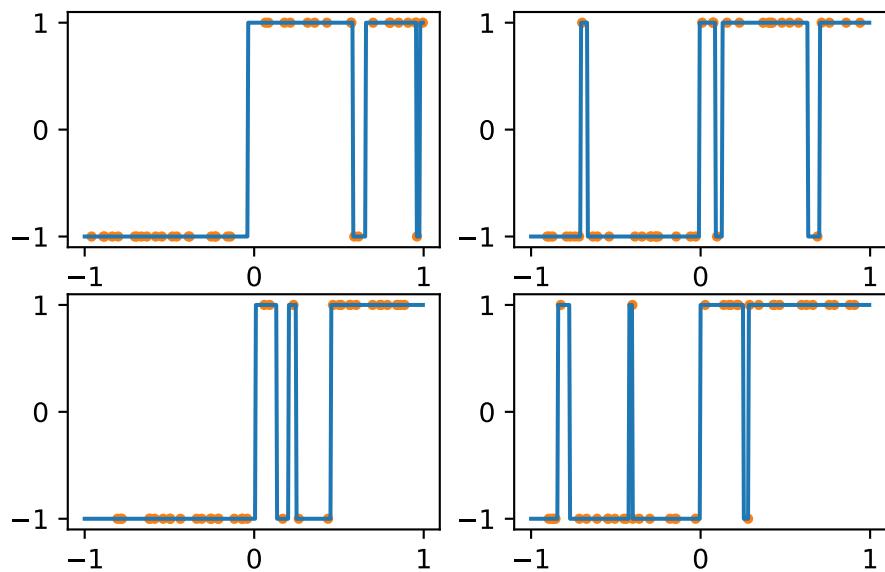


Figure 4.6: Decision tree with depth=6 and noisy data

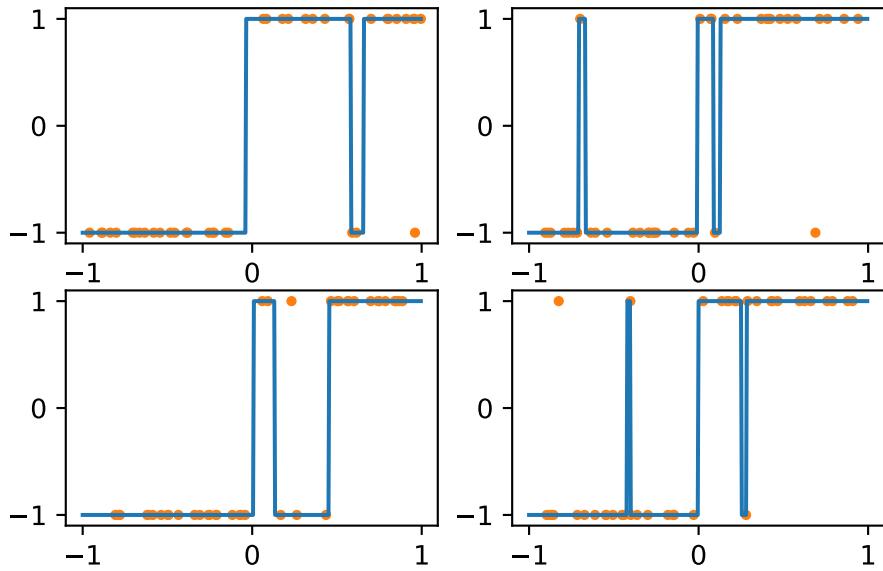


Figure 4.7: Decision tree with depth=3 and noisy data

We can eliminate the overfitting completely and get a single point as the decision boundary, although its location still might not be ideal:

### 4.2.3 Overfitting and variance

The tendency to fit closely to training data also implies that the learner may have a good deal of variance in training (see Figure 4.3, and Figure 4.6, for example). Thus, overfitting is often associated with a large gap between training and testing variance, as observed in Section 4.1.

**Example 4.3.** Returning to the forest data from Example 4.2, we try decision trees of maximum depth  $r = 12$  on 100 random training subsets of size 5000:

```

forest = datasets.fetch_covtype()
X = forest["data"][:50000,:8]
y = (forest["target"][:50000] == 1)

def experiment(learner, X, y, n):
    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=0.2,
        shuffle=True,
        random_state=1
    )

```

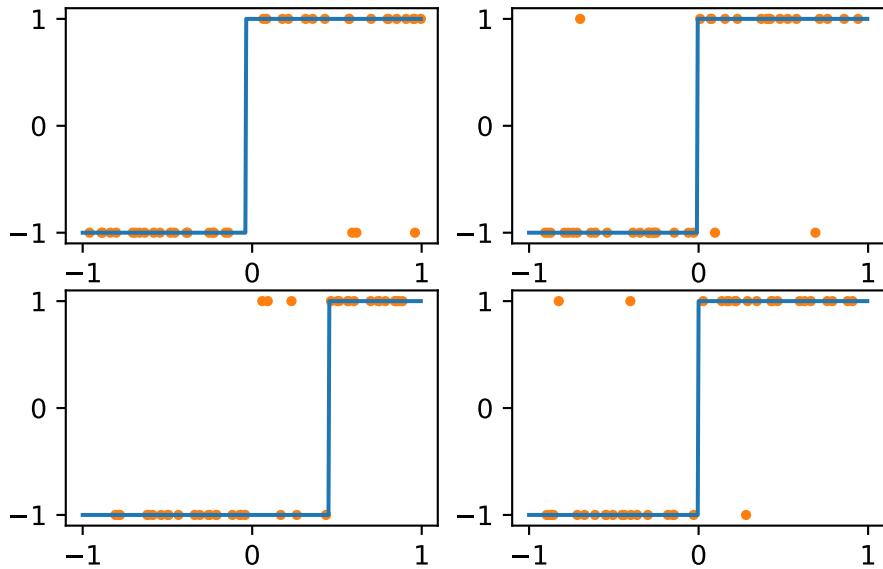


Figure 4.8: Decision tree with depth=2 and noisy data

```

)
results = []
for i in range(100):
    X_train, y_train = shuffle(X_train, y_train, random_state=i)
    XX, yy = X_train[:n,:], y_train[:n]
    learner.fit(XX, yy)
    err = 1 - balanced_accuracy_score(yy, learner.predict(XX))
    results.append( ("train", err) )    # training error
    err = 1 - balanced_accuracy_score(y_test, learner.predict(X_test))
    results.append( ("test", err) )      # test error

results = pd.DataFrame( results, columns=["kind", "error"] )
sns.displot(data=results, x="error", hue="kind", bins=20);

tree = DecisionTreeClassifier(max_depth=12)
experiment(tree, X, y, 5000)

```

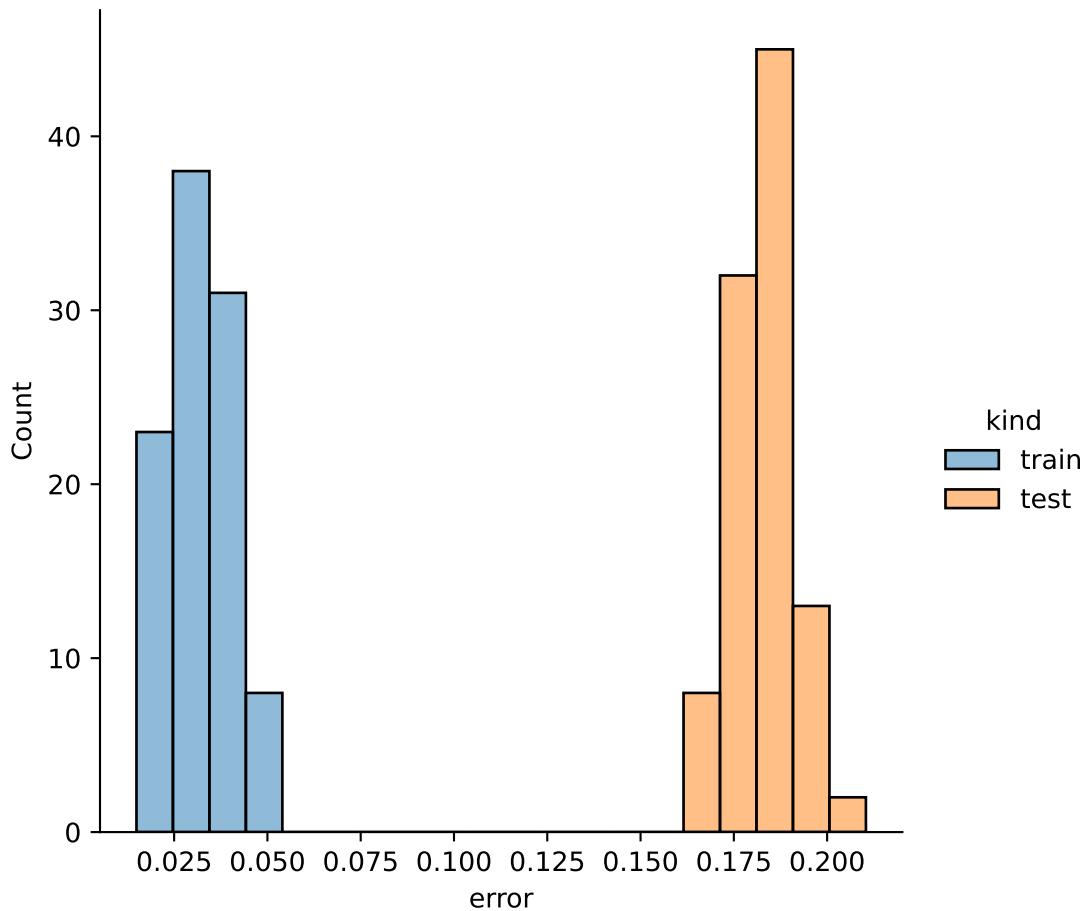
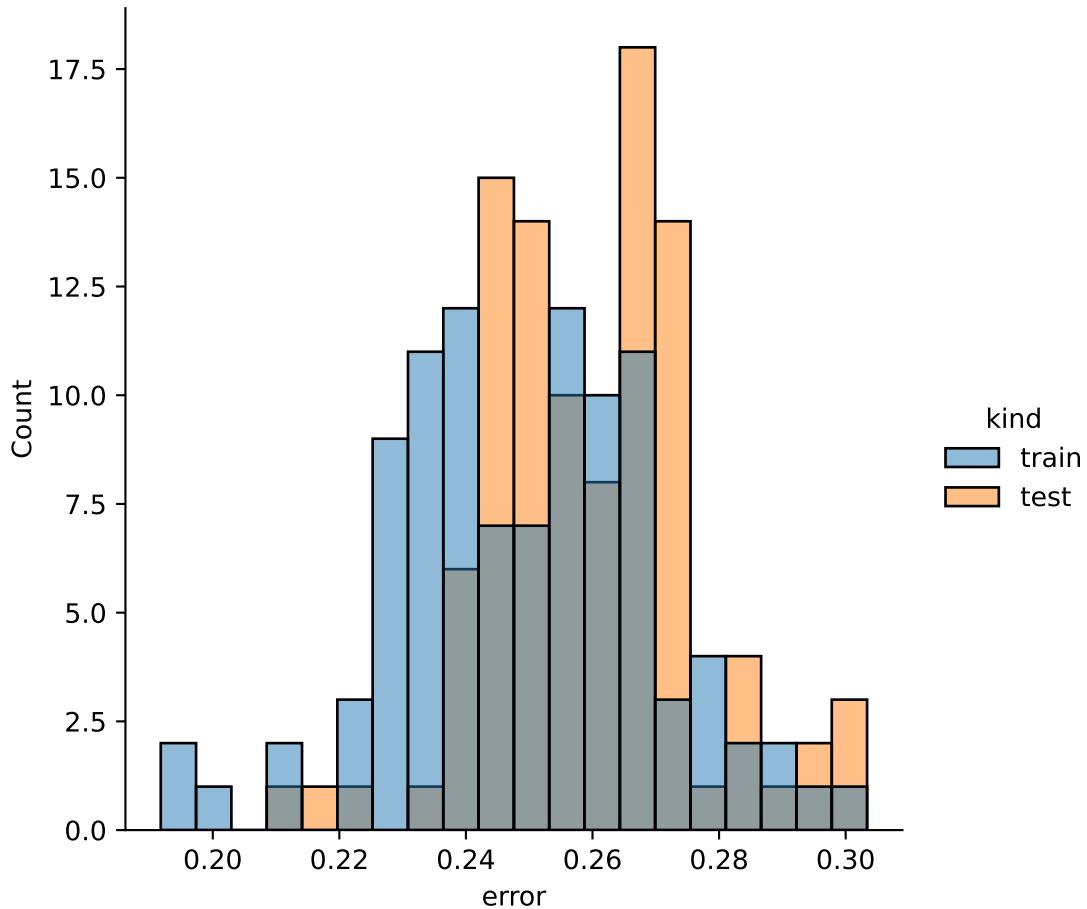


Figure 4.9: Results from an overfit decision tree

Since  $2^{12} = 4096$ , this tree is probably overfit to the training data, and we also see the wide separation between training and testing that suggests the training does not generalize well. With a depth of  $r = 4$ , the training and testing results completely overlap:

```
tree = DecisionTreeClassifier(max_depth=4)
experiment(tree, X, y, 5000)
```



However, notice above that the testing error increased substantially from the overfit case.

We could say that the last tree in Example 4.3 is underfitting the data; the behavior of the labelling function is probably too complex to be replicated well by any tree that shallow. In short, the overfitting/underfitting dilemma is another manifestation of the bias-variance tradeoff.

[https://www.dropbox.com/s/o68bl19we1mbgfv/Example4\\_3.mp4?raw=1](https://www.dropbox.com/s/o68bl19we1mbgfv/Example4_3.mp4?raw=1)



### 4.3 Ensemble methods

When a relatively expressive learning model is used, overfitting and strong dependence on the training set are possible. One meta-strategy for reducing training variance without decreasing the model expressiveness is to use an **ensemble** method. The idea of an ensemble is that averaging over many different training sets will reduce the variance that comes from overfitting. It's a way to simulate the computation of expected values.

**Definition 4.2.** In *bootstrap aggregation*, or **bagging** for short, samples are drawn randomly from the original training set. Usually, this is done *with replacement*, which means that some samples might be selected multiple times.

Why should bagging work? It comes down to the way that bias and variance behave. Suppose we produce  $M$  probabilistic binary classifiers,  $\hat{f}_1, \dots, \hat{f}_M$ , that are identical except in having received independent training sets. They create the (probabilistic) bagging classifier

$$\hat{F}(x) = \frac{1}{M} \sum_{m=1}^M \hat{f}_m(x).$$

[https://www.  
dropbox.com/  
s/  
hi52361ot3zxak2/  
Section4\\_3.  
mp4?raw=1](https://www.dropbox.com/s/hi52361ot3zxak2/Section4_3.mp4?raw=1)

The bias of the bagging classifier is the expectation of

$$y - \hat{F}(x) = \frac{1}{M} (My) - \frac{1}{M} \sum_{m=1}^M \hat{f}_m(x) = \frac{1}{M} \sum_{m=1}^M (y - \hat{f}_m(x)).$$

This is simply the mean of the constituent classifier biases. Since the original classifiers are identical, in expected value they all have the same bias, and we conclude that the bagging classifier has the same expected bias as its constituents.

The story for the variance is different. It [can be derived](#) that the variance of the bagging predictor is  $1/M$  times that of the constituent classifiers. (This effect can be used to explain the *wisdom of crowds*. If each person in a classroom is asked to guess the number of jellybeans in a large jar, the mean of the class' guesses will tend to be much closer to the true value than most individuals' guesses are.)

We should expect bagging to work best with highly expressive classifiers that have low bias and large variance. These tend to occur in large-depth decision trees and small- $k$  kNN classifiers.

Scikit-learn has a `BaggingClassifier` that automates the process of generating an ensemble from just one basic type of estimator.

**Example 4.4.** Here is a dataset collected from images of dried beans:

```
beans = pd.read_excel("_datasets/Dry_Bean_Dataset.xlsx")
X = beans.drop("Class", axis=1)
X.head()
```

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRatio	Eccentricity	ConvexArea
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	28715
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	29172
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	29690
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	30724
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	30417

Although the dataset has data on 7 classes of beans, we will simplify our output by making it a one-vs-rest problem for just one class:

```
y = beans["Class"] == "SIRA"
```

Here is the confusion matrix we get from training a single kNN classifier on this dataset:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=302
)

pipe = make_pipeline(StandardScaler(), KNeighborsClassifier(n_neighbors=3))
pipe.fit(X_train, y_train)

p_hat = pipe.predict_proba(X_test)
auc = roc_auc_score(y_test==True, p_hat[:,1])    # columns are for [False, True]
```

```
print(f"AUC for single classifier is {auc:.4f}")
```

```
AUC for single classifier is 0.9633
```

Here, we create an ensemble with 100 such classifiers, each trained on a different subset that is 75% of the size of the original training set:

```
from sklearn.ensemble import BaggingClassifier
```

```
ensemble = BaggingClassifier(  
    pipe,  
    max_samples=0.75,  
    n_estimators=100,  
    random_state=18621  
)
```

```
ensemble.fit(X_train, y_train)
```

```
BaggingClassifier(estimator=Pipeline(steps=[('standardscaler',  
                                             StandardScaler()),  
                                         ('kneighborsclassifier',  
                                         KNeighborsClassifier(n_neighbors=3))]),  
                  max_samples=0.75, n_estimators=100, random_state=18621)
```

We can use the trained ensemble object much like any learner. For example, here is the prediction obtained for the last row of the training set:

```
query = X_test.iloc[-1:,:]  
p_hat = ensemble.predict_proba(query)  
print(f"Predicted ensemble probability of True on query is {p_hat[0][1]:.2%}")
```

```
Predicted ensemble probability of True on query is 52.67%
```

Internally, the `estimators_` field of the ensemble object is a list of the individual trained classifiers. With a little work, we could find out the prediction for *True* from every constituent:

```
pm = [model.predict_proba(query.to_numpy())[0,1] for model in ensemble.estimators_]  
pm[:6] # first 6 predictions
```

```
[0.6666666666666666,  
 1.0,  
 0.3333333333333333,  
 0.6666666666666666,  
 0.3333333333333333,  
 0.6666666666666666]
```

The ensemble takes the average of this list to create its prediction:

```
print(f"Mean probability of True for query is {np.mean(pm)}")
```

```
Mean probability of True for query is 52.67%
```

The result above matches what we got by predicting directly from the ensemble, which is the normal mode of operation.

Over the testing set, we find that the ensemble has improved the AUC score:

```
p_hat = ensemble.predict_proba(X_test)  
auc = roc_auc_score(y_test==True, p_hat[:,1])    # columns are for [False, True]  
print(f"AUC for ensemble is {auc:.4f}")
```

```
AUC for ensemble is 0.9839
```

There is a significant catch in that the theory requires the constituent learners to be uncorrelated, which is less true as the size of the bagging sample grows relative to the original training set. This can somewhat counterintuitively lead to better results by training on *smaller* individual training sets.

### i Note

An ensemble of decision trees is known as a **random forest**. We can use a `RandomForestClassifier` to accomplish the same thing as a bagged decision tree ensemble.

[https://www.dropbox.com/s/55aiz2i5br6psok/Example4\\_4.mp4?raw=1](https://www.dropbox.com/s/55aiz2i5br6psok/Example4_4.mp4?raw=1)

**Example 4.5.** If we repeat the above but reduce the bagging training sets to just 20% of the full training set, we get a slightly better result:

```
ensemble = BaggingClassifier(  
    pipe,  
    max_samples=0.2,
```

```

        n_estimators=100,
        random_state=18621
    )

ensemble.fit(X_train, y_train)
p_hat = ensemble.predict_proba(X_test)
auc = roc_auc_score(y_test==True, p_hat[:,1])    # columns are for [False, True]
print(f"AUC for the new ensemble is {auc:.4f}")

```

AUC for the new ensemble is 0.9873

We may get better results by increasing the size of the ensemble, too, though in this case there isn't much room left for improvement.

**Example 4.6.** Let's work again with the forest cover dataset. It's got over 500,000 samples and 54 features:

```

forest = datasets.fetch_covtype()
forest["data"].shape

```

(581012, 54)

We'll turn this into a binary classification by looking for just one of the possible label values:

```

X = forest["data"]
y = forest["target"] == 1
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=302
)

```

How should we best make use of all that data? We can use a fairly deep decision tree without fear of overfitting, and the result is not bad:

```

tree = DecisionTreeClassifier(max_depth=12)
tree.fit(X_train, y_train)

from sklearn.metrics import f1_score
F1 = f1_score(y_test, tree.predict(X_test) )
print(f"F1 for a single tree is {F1:.4f}")

```

```
F for a single tree is 0.8106
```

A simple averaging over the same type of tree actually does worse here:

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(
    max_depth=12,
    max_samples=0.2,
    n_estimators=100, n_jobs=-1
)

rf.fit(X_train,y_train)
F1 = f1_score(y_test, rf.predict(X_test) )
print(f"F for a random forest is {F1:.4f}")
```

```
F for a random forest is 0.7781
```

It's possible that these trees are too correlated. We can combat that by exaggerating their degree of overfitting:

```
rf = RandomForestClassifier(
    max_depth=24,
    max_samples=0.2,
    n_estimators=100, n_jobs=-1
)

rf.fit(X_train,y_train)
F1 = f1_score(y_test, rf.predict(X_test) )
print(f"F for a taller forest is {F1:.4f}")
```

```
F for a taller forest is 0.9031
```

In fact, we can decorrelate even better by only using random subsets of the features on each tree. Here, for example, we construct the ensemble so that each tree randomly selects 50% of the original 54 dimensions to work with:

```
rf = RandomForestClassifier(
    max_depth=24,
    max_features=0.5,
    max_samples=0.2,
    n_estimators=100, n_jobs=-1
```

```
)
```

```
rf.fit(X_train,y_train)
F1 = f1_score(y_test, rf.predict(X_test) )
print(f"F for a taller, skinnier forest is {F1:.4f}")
```

F for a taller, skinnier forest is 0.9406

Ensembles can be constructed for any individual model type. Their chief disadvantage is the need to repeat the fitting process multiple times, although this can be mitigated by computing the fits in parallel. For random forests in particular, we also lose the ability to interpret the decision process the way we can for an individual tree.

[https://www.  
dropbox.com/  
s/  
73nq8mjams2imeg/  
Example4\\_6.  
mp4?raw=1](https://www.dropbox.com/s/73nq8mjams2imeg/Example4_6.mp4?raw=1)



## 4.4 Validation

We now return to the opening questions of this chapter: how should we determine optimal hyperparameters and algorithms?

It's tempting to compute some test scores over a range of hyperparameter choices and simply choose the case that scores best. However, if we base hyperparameter optimization on a fixed testing set, then we are effectively learning from that set! The hyperparameters might become too tuned—i.e., overfit—to our particular choice of the test set.

To avoid this pitfall, we can split the data into *three* subsets for training, **validation**, and testing. The validation set is used to tune hyperparameters. Once training is performed at values determined to be best on validation, the test set is used to assess the generalization of the optimized learner.

Unfortunately, a fixed three-way split of the data further reduces the amount of data available for training, so we often turn to an alternative.

[https://www.  
dropbox.com/  
s/  
87jspt2wxhrvhnt/  
Section4\\_4.  
mp4?raw=1](https://www.dropbox.com/s/87jspt2wxhrvhnt/Section4_4.mp4?raw=1)

### 4.4.1 Cross-validation

In **cross-validation**, each learner is trained multiple times using unique training and validation sets drawn from the same pool.

[https://www.  
dropbox.com/  
s/  
amedhad5b13q662/  
Section4\\_4\\_1.  
mp4?raw=1](https://www.dropbox.com/s/amedhad5b13q662/Section4_4_1.mp4?raw=1)

**Definition 4.3.** The steps for ***k*-fold cross-validation** are as follows:

1. Divide the original data into training and testing sets.
2. Further divide the training data set into  $k$  roughly equal parts called *folds*.
3. Train a learner using folds  $2, 3, \dots, k$  and validate on the cases in fold 1. Then train another learner on folds  $1, 3, \dots, k$  and validate against the cases in fold 2. Continue until each fold has served once for validation.
4. Select the hyperparameters producing the best validation score and retrain on the entire training set.
5. Assess performance using the test set.

A different variation is **stratified *k*-fold**, in which the division in step 2 is constrained so that the relative membership of each class is the same in every fold as it is in the full training set. This is advisable when one or more classes is scarce and might otherwise become underrepresented in some folds.

**Example 4.7.** Here is how 16 elements can be split into 4 folds:

```
from sklearn.model_selection import KFold
```

```

kf = KFold(n_splits=4, shuffle=True, random_state=0)
for train,test in kf.split(range(16)):
    print("train:", train, ", test:", test)

train: [ 0  2  3  4  5  7 10 11 12 13 14 15] , test: [1  6  8  9]
train: [ 0  1  3  5  6  7  8  9 10 11 12 15] , test: [ 2  4 13 14]
train: [ 0  1  2  3  4  5  6  8  9 12 13 14] , test: [ 7 10 11 15]
train: [ 1  2  4  6  7  8  9 10 11 13 14 15] , test: [ 0  3  5 12]

```

**Example 4.8.** Let's apply cross-validation to the beans dataset.

```

beans = pd.read_excel("_datasets/Dry_Bean_Dataset.xlsx")
X = beans.drop("Class", axis=1)
y = beans["Class"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.15,
    shuffle=True, random_state=302
)

```

A round of 6-fold cross-validation on a standardized kNN classifier looks like the following:

```

from sklearn.model_selection import cross_validate

knn = KNeighborsClassifier(n_neighbors=5)
learner = make_pipeline(StandardScaler(), knn)

kf = KFold(n_splits=6, shuffle=True, random_state=18621)
scores = cross_validate(
    learner,
    X_train, y_train,
    cv=kf,
    scoring="balanced_accuracy"
)

print("Validation scores:")
print( scores["test_score"] )

```

```

Validation scores:
[0.93344028 0.9297571  0.92846513 0.93272807 0.94025875 0.93982356]

```

The low variance across the folds that we see above is reassurance that they are representative. Conversely, if the scores were spread more widely, we would be concerned that there was strong dependence on the training set, which might indicate overfitting.

#### 4.4.2 Hyperparameter tuning

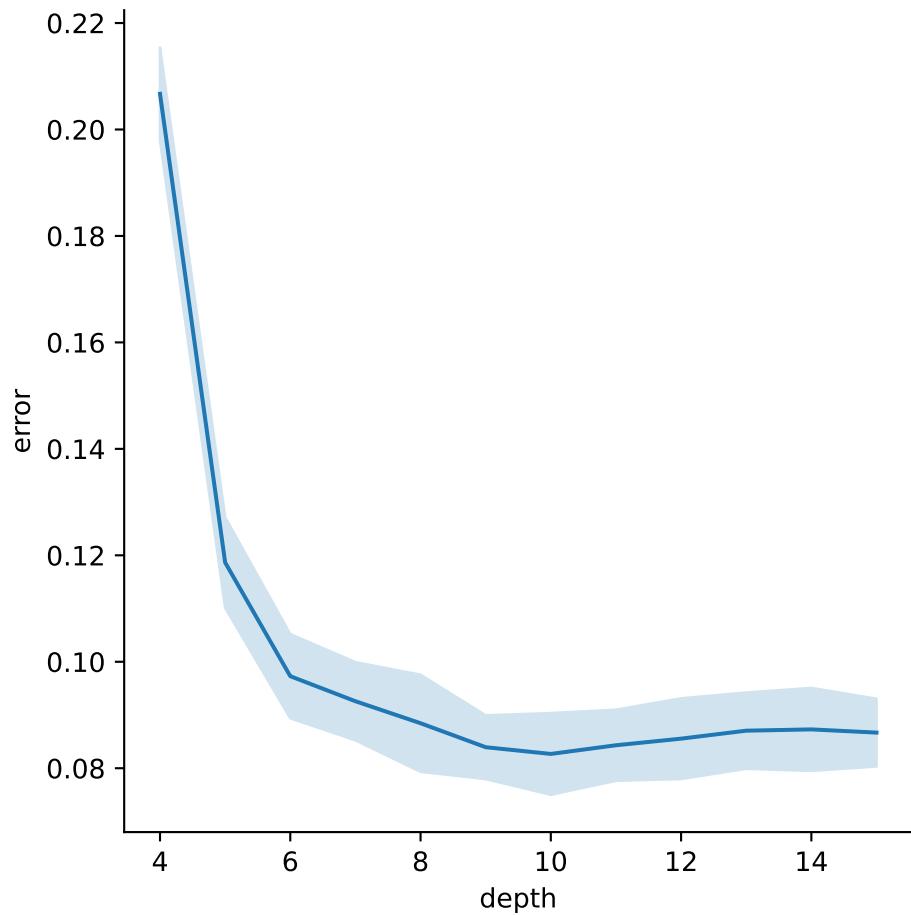
If we perform cross-validations as we vary a hyperparameter, we get a **validation curve**.

**Example 4.9.** Here is a validation curve for the maximum depth of a decision tree classifier on the beans data:

```
from sklearn.model_selection import StratifiedKFold

depths = range(4, 16, 1)
kf = StratifiedKFold(n_splits=8, shuffle=True, random_state=2)
results = []      # for keeping results
for d in depths:
    tree = DecisionTreeClassifier(max_depth=d, random_state=1)
    cv = cross_validate(tree,
        X_train, y_train,
        cv=kf,
        scoring="balanced_accuracy",
        n_jobs=-1
    )
    for err in 1 - cv["test_score"]:
        results.append( (d, err) )

results = pd.DataFrame(results, columns=["depth", "error"] )
sns.relplot(data=results,
    x="depth", y="error",
    kind="line", errorbar="sd"
);
```



Initially the error decreases because the shallowest decision trees are underfit. The minimum error is at max depth 9, after which overfitting seems to take over:

```
results.groupby("depth").mean()
```

depth	error
4	0.206700
5	0.118655
6	0.097290
7	0.092586
8	0.088454
9	0.083948
10	0.082687
11	0.084315
12	0.085555
13	0.087055
14	0.087295
15	0.086693

We can now train this optimal classifier on the entire training set and measure performance on the reserved testing data:

```
tree = DecisionTreeClassifier(max_depth=9, random_state=1)
tree.fit(X_train, y_train)
yhat = tree.predict(X_test)
print("score is", balanced_accuracy_score(y_test, yhat))

score is 0.9192506467389886
```

[https://www.  
dropbox.com/  
s/  
dg0su9kvhvvcx7g/  
Example4\\_9.  
mp4?raw=1](https://www.dropbox.com/s/dg0su9kvhvvcx7g/Example4_9.mp4?raw=1)

#### 4.4.2.1 Grid search

When there is a single hyperparameter in play, the validation curve is useful way to optimize it. When multiple hyperparameters are available, it's common to perform a *grid search*, in which we try cross-validated fitting using every specified combination of parameter values.

**Example 4.10.** Let's work with a dataset on breast cancer detection:

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer(as_frame=True)[["frame"]]
X = cancer.drop("target", axis=1)
y = cancer["target"]

X_train, X_test, y_train, y_test = train_test_split(
```

```

        X, y,
        test_size=0.15,
        shuffle=True, random_state=3383
    )
X_test.head()

```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness
242	11.300	18.19	73.93	389.4	0.09592	0.13250
375	16.170	16.07	106.30	788.5	0.09880	0.14380
446	17.750	28.03	117.30	981.6	0.09997	0.13140
289	11.370	18.89	72.17	396.0	0.08713	0.05008
318	9.042	18.90	60.07	244.5	0.09968	0.19720

We start by trying decision tree classifiers in which we vary the maximum depth as well as some other options.

```

from sklearn.model_selection import GridSearchCV

grid = { "criterion":["gini", "entropy"],
         "max_depth":range(2, 15),
         "min_impurity_decrease":np.arange(0,0.01,0.002) }
learner = DecisionTreeClassifier(random_state=1)
kf = StratifiedKFold(n_splits=4, shuffle=True, random_state=302)

grid_dt = GridSearchCV(
    learner, grid,
    scoring="f1",
    cv=kf,
    n_jobs=-1
)
grid_dt.fit(X_train, y_train)

print("Best parameters:")
print(grid_dt.best_params_)
print()
print("Best score:")
print(grid_dt.best_score_)

```

Best parameters:

{'criterion': 'gini', 'max\_depth': 7, 'min\_impurity\_decrease': 0.0}

Best score:

```
0.9455901279430692
```

Next, we do the same search over kNN classifiers. We always use standardization as a preprocessor; note how the syntax of the grid search is adapted:

```
grid = { "kneighborsclassifier__metric": ["euclidean", "manhattan"],  
        "kneighborsclassifier__n_neighbors": range(1, 20),  
        "kneighborsclassifier__weights": ["uniform", "distance"] }  
  
learner = make_pipeline(StandardScaler(), KNeighborsClassifier())  
  
grid_knn = GridSearchCV(  
    learner, grid,  
    scoring="f1",  
    cv=kf,  
    n_jobs=-1  
)  
grid_knn.fit(X_train, y_train)  
  
print("Best parameters:")  
print(grid_knn.best_params_)  
print()  
print("Best score:")  
print(grid_knn.best_score_)
```

```
Best parameters:
```

```
{'kneighborsclassifier__metric': 'manhattan', 'kneighborsclassifier__n_neighbors': 4, 'knei...}
```

```
Best score:
```

```
0.9734627184207015
```

Each fitted grid search object is itself a classifier that was trained on the full training set at the optimal hyperparameters:

```
dt_score = f1_score( y_test, grid_dt.predict(X_test) )  
knn_score = f1_score( y_test, grid_knn.predict(X_test) )  
print(f"best tree f1 score: {dt_score:.5f}")  
print(f"best knn f1 score: {knn_score:.5f}")  
  
best tree f1 score: 0.94915  
best knn f1 score: 0.99187
```

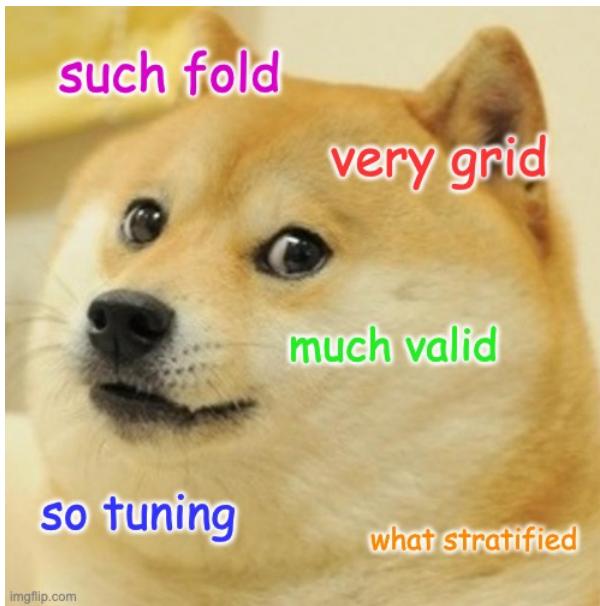
### **i Note**

It may be instructive to rerun the competition above using different random seeds. The meaningfulness of the results is limited by their sensitivity to such choices. Don't let floating-point values give you a false feeling of precision!

#### **4.4.2.2 Alternatives to grid search**

Grid search is a brute-force approach. It is *embarrassingly parallel*, meaning that different processors can work on different locations on the grid at the same time. But it is usually too slow for large training sets, or when the search space has more than two or three dimensions. In such cases you can try searching over crude versions of the grid, perhaps with just part of the training data, and gradually narrow the search while using all the data. When desperate, one may try a randomized search and to guide the process with experience and intuition.

[https://www.  
dropbox.com/  
s/  
di1np5781wkxndy/  
Example4\\_  
10.mp4?raw=  
1](https://www.dropbox.com/s/di1np5781wkxndy/Example4_10.mp4?raw=1)



# 5 Regression

```
import numpy as np
from numpy.random import default_rng
import pandas as pd
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix, f1_score, balanced_accuracy_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_curve
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.model_selection import cross_validate, validation_curve
from sklearn.model_selection import GridSearchCV
```

**Definition 5.1.** **Regression** is the task of approximating the value of a dependent quantitative variable as a function of independent variables, sometimes called *predictors*.

Regression and classification are distinct but not altogether different. Abstractly, both are concerned with reproducing a function  $f$  whose domain is feature space. In classification, the range of  $f$  is a finite set of class labels, while in regression, the range is the real number line (or an interval in it). We can always take the output of a regression and round or bin it to get a finite set of classes; therefore, any regression method can also be used for classification. Likewise, most classification methods have a generalization to regression.

In addition to prediction tasks, some regression methods can be used to identify the relative significance of each feature and whether it has a direct or inverse relationship to the function value. Unimportant features can then be removed to help minimize overfitting.

[https://www.  
dropbox.com/  
s/  
a1h989lg4xe3ak4/  
Section5\\_0.  
mp4?raw=1](https://www.dropbox.com/s/a1h989lg4xe3ak4/Section5_0.mp4?raw=1)

## 5.1 Linear regression

You have likely previously encountered the most basic form of regression: fitting a straight line to data points  $(x_i, y_i)$  in the  $xy$ -plane. In **linear regression**, we have a one-dimensional feature  $x$  and assume a relation

$$y \approx \hat{f}(x) = ax + b.$$

We also define a **loss function** or *misfit function* that adds up how far predictions are from the data. The standard choice is a sum of squared differences between the predictions and the true values:

$$L(a, b) = \sum_{i=1}^n (\hat{f}(x_i) - y_i)^2 = \sum_{i=1}^n (ax_i + b - y_i)^2.$$

The loss can be minimized using a little multidimensional calculus. Momentarily suppose that  $b$  is held fixed and take a derivative with respect to  $a$ :

$$\frac{\partial L}{\partial a} = \sum_{i=1}^n 2x_i(ax_i + b - y_i) = 2a \left( \sum_{i=1}^n x_i^2 \right) + 2b \left( \sum_{i=1}^n x_i \right) - 2 \sum_{i=1}^n x_i y_i.$$

### Note

The symbol  $\frac{\partial}{\partial}$  is called a **partial derivative** and is defined just as described here: differentiate in one variable while all others are temporarily held constant.

Similarly, if we hold  $a$  fixed and differentiate with respect to  $b$ , then

$$\frac{\partial L}{\partial b} = \sum_{i=1}^n 2(ax_i + b - y_i) = 2a \left( \sum_{i=1}^n x_i \right) + 2bn - 2 \sum_{i=1}^n y_i.$$

Setting both derivatives to zero creates a system of two linear equations to be solved for  $a$  and  $b$ :

$$\begin{aligned} a \left( \sum_{i=1}^n x_i^2 \right) + b \left( \sum_{i=1}^n x_i \right) &= \sum_{i=1}^n x_i y_i, \\ a \left( \sum_{i=1}^n x_i \right) + bn &= \sum_{i=1}^n y_i. \end{aligned} \tag{5.1}$$

[https://www.  
dropbox.com/  
s/  
3u136kizgj4qd15/  
Section5\\_1.  
mp4?raw=1](https://www.dropbox.com/s/3u136kizgj4qd15/Section5_1.mp4?raw=1)

**Example 5.1.** Suppose we want to find the linear regressor of the points  $(-1, 0)$ ,  $(0, 2)$ ,  $(1, 3)$ . We need to calculate a few sums:

$$\begin{aligned}\sum_{i=1}^n x_i^2 &= 1 + 0 + 1 = 2, & \sum_{i=1}^n x_i &= -1 + 0 + 1 = 0, \\ \sum_{i=1}^n x_i y_i &= 0 + 0 + 3 = 3, & \sum_{i=1}^n y_i &= 0 + 2 + 3 = 5.\end{aligned}$$

Note that  $n = 3$ . Therefore we must solve

$$\begin{aligned}2a + 0b &= 3, \\ 0a + 3b &= 5.\end{aligned}$$

The regression function is  $\hat{f}(x) = \frac{3}{2}x + \frac{5}{3}$ .

### 5.1.1 Linear algebra

Before moving on, we want to examine a vector-oriented description of the process. If we define

$$\mathbf{e} = [1, 1, \dots, 1] \in \mathbb{R}^n,$$

that is,  $\mathbf{e}$  as a vector of  $n$  ones, then

$$L(a, b) = \|a\mathbf{x} + b\mathbf{e} - \mathbf{y}\|_2^2,$$

Minimizing  $L$  over all values of  $a$  and  $b$  is called the **least squares** problem. (More specifically, this setup is called *simple least squares* or *ordinary least squares*.)

We can write out the equations for  $a$  and  $b$  using another important idea from linear algebra.

**Definition 5.2.** Given any  $d$ -dimensional real-values vectors  $\mathbf{u}$  and  $\mathbf{v}$ , their **inner product** is

$$\mathbf{u}^T \mathbf{v} = \sum_{i=1}^d u_i v_i = u_1 v_1 + u_2 v_2 + \dots + u_d v_d. \quad (5.2)$$

[https://www.dropbox.com/s/pqr1b4yjjdbk9t5/Section5\\_1\\_1.mp4?raw=1](https://www.dropbox.com/s/pqr1b4yjjdbk9t5/Section5_1_1.mp4?raw=1)

The vector inner product is defined only between two vectors of the same length (dimension). There is an important link between the inner product and the 2-norm:

$$\mathbf{u}^T \mathbf{u} = \sum_{i=1}^d u_i^2 = \|\mathbf{u}\|_2^2. \quad (5.3)$$

**i** Note

*Inner product* is a term from linear algebra. In physics and vector calculus with  $d = 2$  or  $d = 3$ , the same thing is often called a *dot product* and written as  $\mathbf{u} \cdot \mathbf{v}$ .

**i** Note

The  $T$  symbol is a *transpose* operation in linear algebra. We won't need it as an independent concept, so we are just using it as notation in the inner product.

**Example 5.2.** Let  $\mathbf{u} = [1, -1, 1, -2]$  and  $\mathbf{v} = [5, 3, -1, 2]$ . Then

$$\mathbf{u}^T \mathbf{v} = (1)(5) + (-1)(3) + (1)(-1) + (-2)(2) = -3.$$

We also have

$$\|\mathbf{u}\|_2^2 = \mathbf{u}^T \mathbf{u} = (1)^2 + (-1)^2 + (1)^2 + (-2)^2 = 7.$$

The equations in Equation 5.1 may now be written as

$$\begin{aligned} a(\mathbf{x}^T \mathbf{x}) + b(\mathbf{x}^T \mathbf{e}) &= \mathbf{x}^T \mathbf{y}, \\ a(\mathbf{e}^T \mathbf{x}) + b(\mathbf{e}^T \mathbf{e}) &= \mathbf{e}^T \mathbf{y}. \end{aligned} \tag{5.4}$$

We can write this as a single equation between two vectors:

$$a \begin{bmatrix} \mathbf{x}^T \mathbf{x} \\ \mathbf{e}^T \mathbf{x} \end{bmatrix} + b \begin{bmatrix} \mathbf{x}^T \mathbf{e} \\ \mathbf{e}^T \mathbf{e} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^T \mathbf{y} \\ \mathbf{e}^T \mathbf{y} \end{bmatrix}.$$

In fact, the operation on the left-hand side is how we define the product of a matrix and a vector, and we can write

$$\begin{bmatrix} \mathbf{x}^T \mathbf{x} & \mathbf{x}^T \mathbf{e} \\ \mathbf{e}^T \mathbf{x} & \mathbf{e}^T \mathbf{e} \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{x}^T \mathbf{y} \\ \mathbf{e}^T \mathbf{y} \end{bmatrix}.$$

This takes the form of the equation  $\mathbf{A}\mathbf{w} = \mathbf{v}$ , where  $\mathbf{A}$  is a known  $2 \times 2$  matrix,  $\mathbf{v}$  is a known 2-vector, and  $\mathbf{w}$  is the 2-vector of the unknowns  $a$  and  $b$ . This equation is referred to as a *linear system* for the unknown vector. Linear systems and their solutions are the central topic of linear algebra. In the background, it is this linear system that is being solved when you perform a linear regression fit.

### 5.1.2 Performance metrics

We need ways to measure regression performance. Unlike with binary classification, in regression it's not just a matter of right and wrong answers—the amount of wrongness matters, too.

A quirk of linear regression is that it's an older and broader idea than most of machine learning, and it's often presented as though the training and testing sets are identical. We follow that convention for the definitions in this section. The same quantities can also be calculated for a set of labels and predictions obtained from a separate testing set, although a few of the properties stated here don't apply in that case.

**Definition 5.3.** The **residuals** of the regression are

$$y_i - \hat{y}_i, \quad i = 1, \dots, n, \quad (5.5)$$

where the  $y_i$  are the true labels and the  $\hat{y}_i$  are the values predicted by the regressor. We can express them compactly as the **residual vector**  $\mathbf{y} - \hat{\mathbf{y}}$ .

 Caution

The terms *error* and *residual* are frequently used interchangeably and even inconsistently. I try to follow the most common practices here, even though the names can be confusing if you think about them too hard.

[https://www.dropbox.com/s/pl9jy2qtqpk8zr2/Section5\\_1\\_2.mp4?raw=1](https://www.dropbox.com/s/pl9jy2qtqpk8zr2/Section5_1_2.mp4?raw=1)

**Definition 5.4.** The **mean squared error** (MSE) is

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \frac{1}{n} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2.$$

The **mean absolute error** (MAE) is

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| = \frac{1}{n} \|\mathbf{y} - \hat{\mathbf{y}}\|_1.$$

The MSE is simply  $1/n$  times the loss function of a linear regression. MAE is less sensitive than MSE to large outliers. Both quantities are dimensional and therefore depend on how the variables are scaled, but only the units of MAE are the same as of the data.

**Example 5.3.** In Example 5.13 the points  $(-1, 0)$ ,  $(0, 2)$ ,  $(1, 3)$  were found to have the least-squares regressor  $\hat{f}(x) = \frac{3}{2}x + \frac{5}{3}$ . Hence

$$\begin{aligned} y_1 &= 0, \quad \hat{y}_1 = \hat{f}(-1) = \frac{1}{6} \\ y_2 &= 2, \quad \hat{y}_2 = \hat{f}(0) = \frac{5}{3}, \\ y_3 &= 3; \quad \hat{y}_3 = \hat{f}(1) = \frac{19}{6}. \end{aligned}$$

This implies

$$\begin{aligned} \text{MSE} &= \frac{1}{3} \left[ \left(0 - \frac{1}{6}\right)^2 + \left(2 - \frac{5}{3}\right)^2 + \left(3 - \frac{19}{6}\right)^2 \right] = \frac{1}{18}, \\ \text{MAE} &= \frac{1}{3} \left( |0 - \frac{1}{6}| + |2 - \frac{5}{3}| + |3 - \frac{19}{6}| \right) = \frac{2}{9}. \end{aligned}$$

**Definition 5.5.** The **coefficient of determination** (CoD) is denoted  $R^2$  and defined as

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2},$$

where  $\bar{y}$  is the sample mean of  $y_1, \dots, y_n$ .

Here are important things to know about the coefficient of determination.

**Theorem 5.1.** Given sample values  $y_1, \dots, y_n$  with mean  $\bar{y}$  and the predictions  $\hat{y}_1, \dots, \hat{y}_n$ ,

1.  $R^2$  is dimensionless and therefore independent of scaling.
2. If  $\hat{y}_i = y_i$  for all  $i$  (i.e., perfect predictions), then  $R^2 = 1$ .
3. If  $\hat{y}_i = \bar{y}$  for all  $i$  (i.e., always predict the sample mean), then  $R^2 = 0$ .
4. If the  $\hat{y}_i$  are found from a linear regression, then  $R^2$  is the square of the Pearson correlation coefficient between  $\mathbf{y}$  and  $\hat{\mathbf{y}}$ .

 Warning

The notation  $R^2$  is *highly* unfortunate. While it is the square of the Pearson coefficient in linear regression,  $R^2$  can actually be negative for other regression methods! Such a result indicates that the predictor is doing worse than just predicting the mean value every time. It has nothing to do with imaginary numbers.

**Example 5.4.** Continuing with Example 5.13 and Example 5.3, we find that  $\bar{y} = (0 + 2 + 3)/3 = \frac{5}{3}$ , and

$$\sum_{i=1}^n (y_i - \bar{y})^2 = (0 - \frac{5}{3})^2 + (2 - \frac{5}{3})^2 + (3 - \frac{5}{3})^2 = \frac{14}{3}.$$

This gives the coefficient of determination

$$\begin{aligned} R^2 &= 1 - \frac{(0 - \frac{1}{6})^2 + (2 - \frac{5}{3})^2 + (3 - \frac{19}{6})^2}{14/3} \\ &= 1 - \frac{1/6}{14/3} = \frac{27}{28}. \end{aligned}$$

This is quite close to 1, indicating a good fit. Compare that to the arbitrary predictor  $\hat{f}(x) = x$ , which has  $\hat{y}_1 = -1$ ,  $\hat{y}_2 = 0$ ,  $\hat{y}_3 = 1$ :

$$\begin{aligned} R^2 &= 1 - \frac{(y_i - \hat{y}_i)^2}{14/3} = \frac{(0+1)^2 + (2-0)^2 + (3-1)^2}{14/3} \\ &= 1 - \frac{9}{14/3} = -\frac{13}{14}. \end{aligned}$$

Since this result is negative, we would be better off always predicting 5/3 rather than using  $\hat{f}(x) = x$ .

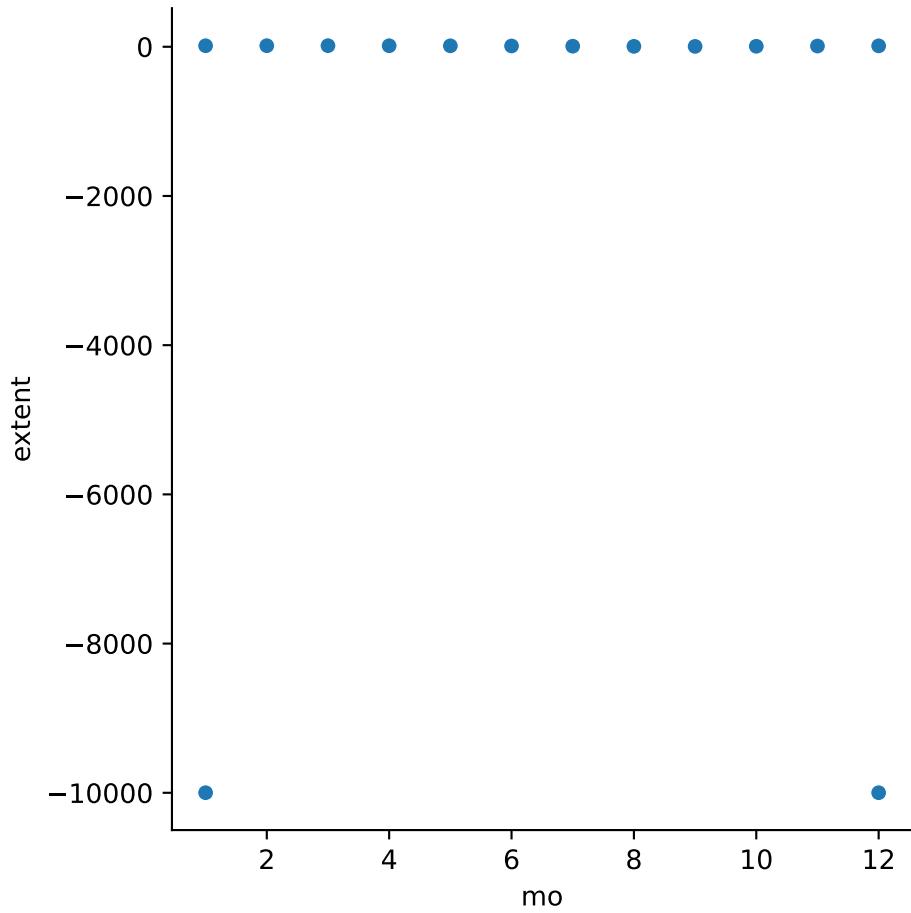
**Example 5.5.** We import data about the extent of sea ice in the Arctic circle, collected monthly since 1979:

```
ice = pd.read_csv("_datasets/sea-ice.csv")
# Simplify column names:
ice.columns = [s.strip() for s in ice.columns]
ice.head()
```

	year	mo	data-type	region	extent	area
0	1979	1	Goddard	N	15.41	12.41
1	1980	1	Goddard	N	14.86	11.94
2	1981	1	Goddard	N	14.91	11.91
3	1982	1	Goddard	N	15.18	12.19
4	1983	1	Goddard	N	14.94	12.01

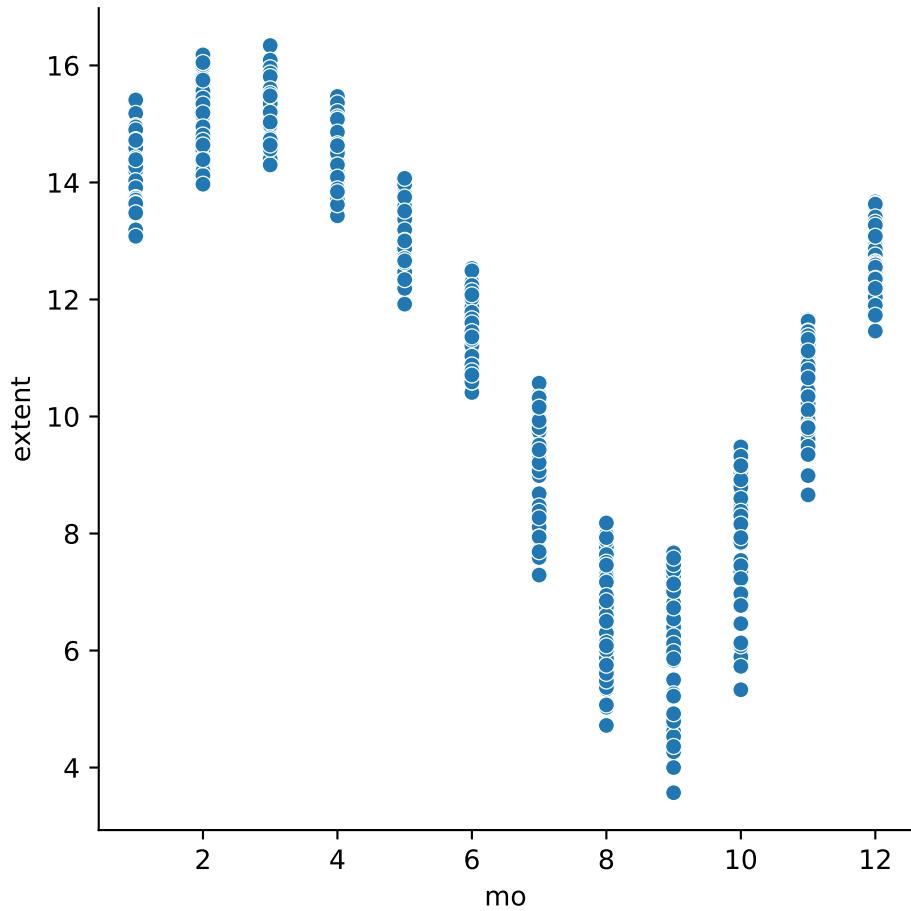
A quick plot reveals something odd-looking:

```
sns.relplot(data=ice, x="mo", y="extent");
```



Everything in the plot above is dominated by two large negative values. These probably represent missing data, so we make a new copy without those rows:

```
ice = ice[ice["extent"] > 0]
sns.relplot(data=ice, x="mo", y="extent");
```



Each dot in the plot above represents one measurement. As you would expect, the extent of ice rises in the winter months and falls in summer:

```
bymonth = ice.groupby("mo")
bymonth[["extent"]].mean()
```

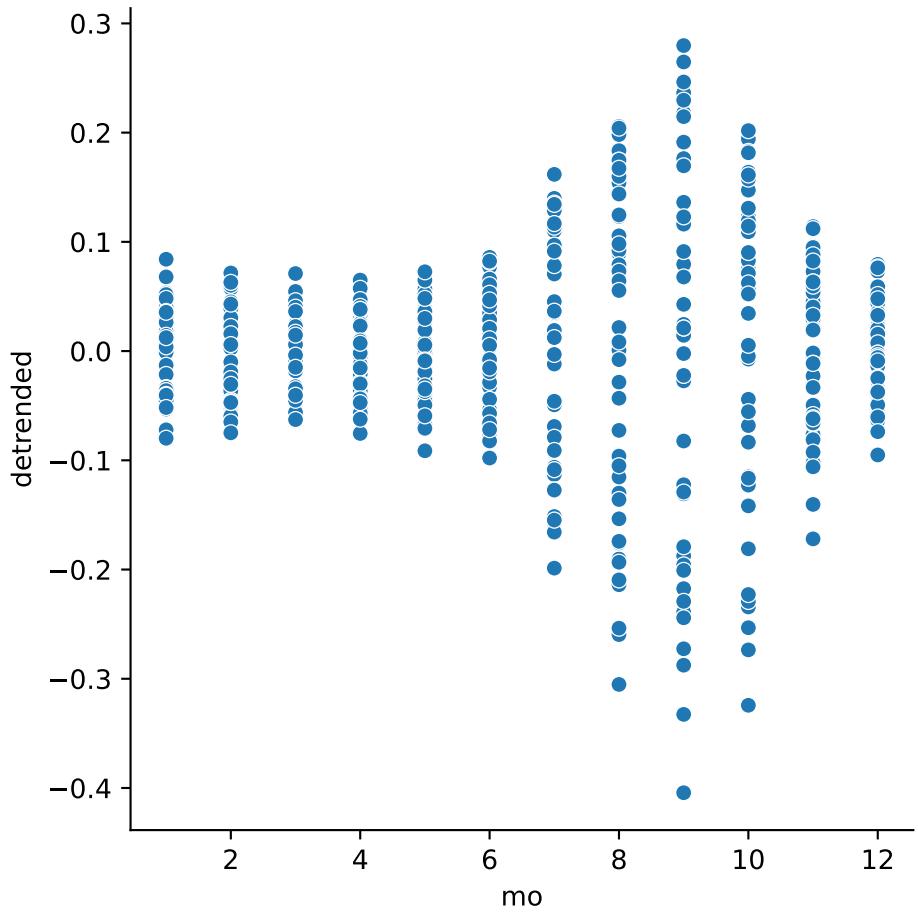
	extent
mo	
1	14.214762
2	15.100233
3	15.256977
4	14.525581
5	13.117442
6	11.539767
7	9.097907
8	6.793256
9	5.993488
10	7.887907
11	10.458182
12	12.664419

While the effect of the seasonal variation somewhat cancels out over time when fitting a line, it's preferable to remove this obvious trend before the fit takes place. To do that, we add a column that measures within each month group the relative change from the mean,

$$\frac{x - \bar{x}}{\bar{x}}.$$

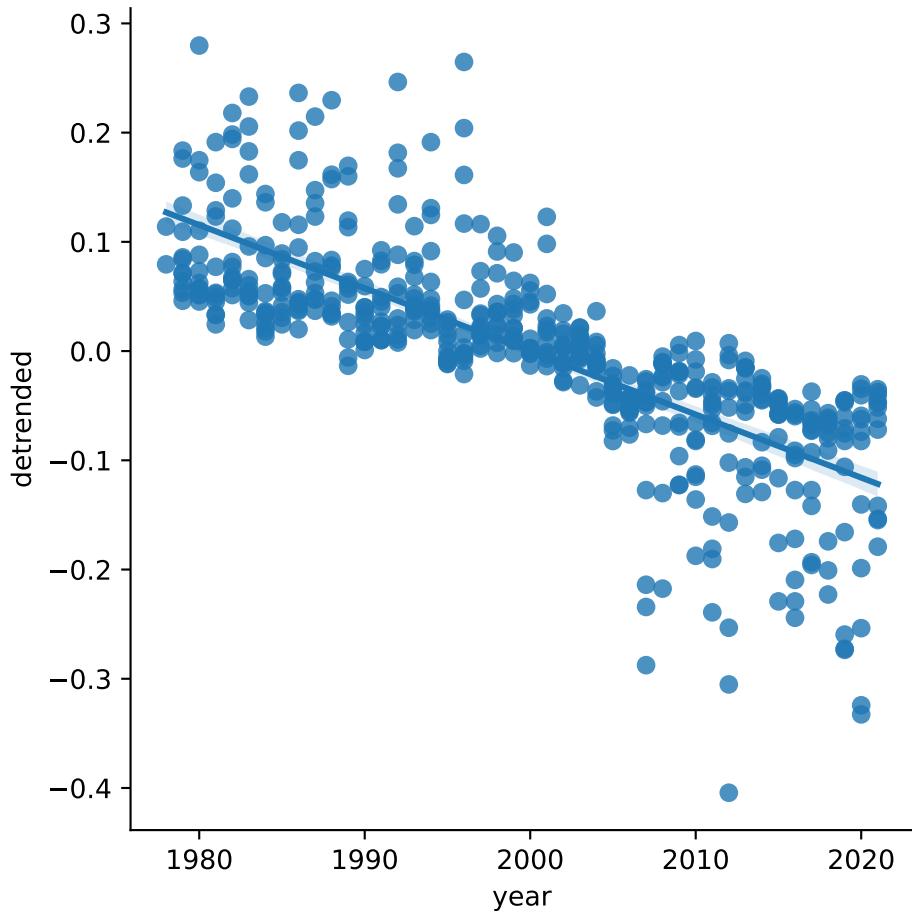
This is done with a `transform` method applied to the grouped frame:

```
recenter = lambda x: x/x.mean() - 1
ice["detrended"] = bymonth["extent"].transform(recenter)
sns.relplot(data=ice, x="mo", y="detrended");
```



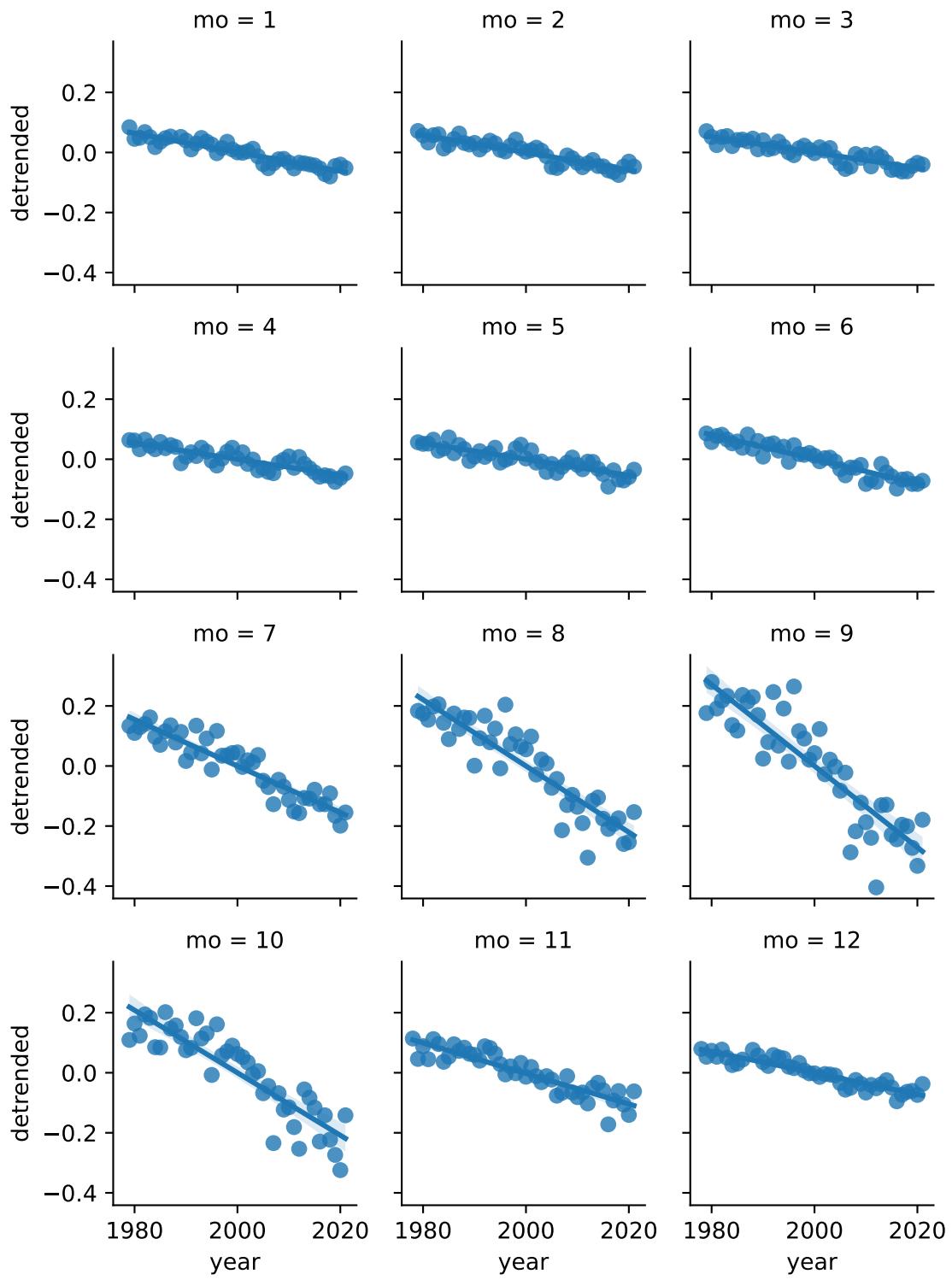
An `lmplot` in seaborn shows the least squares line:

```
sns.lmplot(data=ice, x="year", y="detrended");
```



However, we should be mindful of [Simpson's paradox](#). The previous plot showed considerably more variance within the warm months. How do these fits look for the data *within* each month? This is where a facet plot shines:

```
sns.lmplot(data=ice,
            x="year", y="detrended",
            col="mo", col_wrap=3, height=2
            );
```



Thus, while the correlation is negative within each month, the effect size is clearly larger in the summer and early fall.

We can get numerical information about a regression line from a `LinearRegression()` learner in `sklearn`. We will focus on the data for August:

```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()

ice = ice[ ice["mo"]==8 ]
X = ice[ ["year"] ]
y = ice["detrended"]
lm.fit(X, y)

LinearRegression()
```

We can get the slope and  $y$ -intercept of the regression line from the learner's properties. (Calculated parameters tend to have underscores at the ends of their names in `sklearn`.)

```
slope, intercept = lm.coef_[0], lm.intercept_
print(f"Slope is {slope:.3g} and intercept is {intercept:.3g}")

Slope is -0.011 and intercept is 22.1
```

The slope indicates average decrease over time.

Next, we assess the performance on the training set. Both the MSE and mean absolute error are small relative to dispersion within the values themselves:

```
from sklearn.metrics import mean_squared_error, mean_absolute_error
yhat = lm.predict(X)
mse = mean_squared_error(y, yhat)
mae = mean_absolute_error(y, yhat)

print(f"MSE: {mse:.2e}, compared to variance {y.var():.2e}")
print(f"MAE: {mae:.2e}, compared to standard deviation {y.std():.2e}")

MSE: 4.01e-03, compared to variance 2.33e-02
MAE: 4.93e-02, compared to standard deviation 1.53e-01
```

The `score` method of the regressor object computes the coefficient of determination:

```
R2 = lm.score(X, y)
print(f"R-squared: {R2:.3f}")
```

R-squared: 0.824

An  $R^2$  value this close to 1 would usually be considered a sign of a good fit, although we have not tested for generalization to new data.

[https://www.  
dropbox.com/  
s/  
wtlrgm8m8yzhge7/  
Example5\\_4.  
mp4?raw=1](https://www.dropbox.com/s/wtlrgm8m8yzhge7/Example5_4.mp4?raw=1)

## 5.2 Multilinear regression

We can extend linear regression to  $d$  predictor variables  $x_1, \dots, x_d$ :

$$y \approx \hat{f}(\mathbf{x}) = b + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d.$$

First, observe that we can actually drop the intercept term  $b$  from the discussion, because we could always define an additional constant feature  $x_0 = 1$  and get the same effect in one higher dimension. So we will use the following.

[https://www.  
dropbox.com/  
s/  
6tschaluumjq65e/  
Section5\\_2.  
mp4?raw=1](https://www.dropbox.com/s/6tschaluumjq65e/Section5_2.mp4?raw=1)

**Definition 5.6. Multilinear regression** is the approximation

$$y \approx \hat{f}(\mathbf{x}) = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d = \mathbf{w}^T \mathbf{x},$$

for a constant vector  $\mathbf{w}$  known as the **weight vector**.

### Note

Multilinear regression is also simply called *linear regression* most of the time. What we previously called linear regression is just a special case. The `LinearRegression` learner class does both types of fits. It has a keyword option `fit_intercept` that determines whether or not the  $b$  term above is used.

As before, we find the unknown weight vector  $\mathbf{w}$  by minimizing a loss function. To create the least-squares loss function, we use  $\mathbf{X}_i$  to denote the  $i$ th row of an  $n \times d$  feature matrix  $\mathbf{X}$ . Then

$$L(\mathbf{w}) = \sum_{i=1}^n (y_i - \hat{f}(\mathbf{X}_i))^2 = \sum_{i=1}^n (y_i - \mathbf{X}_i^T \mathbf{w})^2.$$

We encountered a matrix-vector product earlier. It turns out that the following definition is equivalent to that earlier one.

**Definition 5.7.** Given an  $n \times d$  matrix  $\mathbf{X}$  with rows  $\mathbf{X}_1, \dots, \mathbf{X}_n$  and a  $d$ -vector  $\mathbf{w}$ , the product  $\mathbf{X}\mathbf{w}$  is defined by

$$\mathbf{X}\mathbf{w} = \begin{bmatrix} \mathbf{X}_1^T \mathbf{w} \\ \mathbf{X}_2^T \mathbf{w} \\ \vdots \\ \mathbf{X}_n^T \mathbf{w} \end{bmatrix}.$$

**Example 5.6.** Suppose that

$$\mathbf{X} = \begin{bmatrix} 3 & -1 \\ 0 & 2 \\ 1 & 4 \end{bmatrix}, \quad \mathbf{w} = [5, -2].$$

Then  $\mathbf{X}_1 = [3, -1]$ ,  $\mathbf{X}_2 = [0, 2]$ ,  $\mathbf{X}_3 = [1, 4]$ , and

$$\mathbf{X}\mathbf{w} = \begin{bmatrix} (3)(5) + (-1)(-2) \\ (0)(5) + (2)(-2) \\ (1)(5) + (4)(-2) \end{bmatrix} = \begin{bmatrix} 17 \\ -4 \\ -3 \end{bmatrix}.$$

We now have the compact expression

$$L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2. \quad (5.6)$$

As in the  $d = 1$  case, minimizing the loss is equivalent to solving a linear system of equations known as the *normal equations* for the weight vector  $\mathbf{w}$ . We do not present the details here.

### 🔥 Caution

Be careful interpreting the magnitudes of regression coefficients (i.e., entries of the weight vector). These are sensitive to the units and scales of the features. For example, distances expressed in meters would have a coefficient that is 1000 times larger than the same distances expressed in kilometers. For quantitative comparisons, it helps to standardize the features first, which does not affect the quality of the fit.

**Example 5.7.** We return to the data set regarding the fuel efficiency of cars:

```
cars = sns.load_dataset("mpg").dropna()
cars.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chev
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylan
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth s
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel s
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino

In order to ease experimentation, we define a function that fits a given learner to the *mpg* variable using a given list of features from the data frame:

```
def fitcars(model, features):
    X = cars[features]
    y = cars["mpg"]

    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=0.2,
        shuffle=True, random_state=302
    )

    model.fit(X_train, y_train)
    MSE = mean_squared_error(y_test, model.predict(X_test))
    print(f'MSE: {MSE:.3f}, compared to variance {y_test.var():.3f}')
    return None
```

### 💡 Tip

When you run the same lines of code over and over with only a slight change at the beginning, it's advisable to put that code into a function. It makes the overall code shorter and easier to understand and adapt.

First, we try using *horsepower* as the only feature in a linear regression to fit *mpg*:

```
features = ["horsepower"]
lm = LinearRegression( fit_intercept=True )
fitcars(lm, features)
```

MSE: 26.354, compared to variance 56.474

As we would expect, there is an inverse relationship between horsepower and vehicle efficiency:

```
lm.coef_
array([-0.1596552])
```

Next, we add *displacement* to the regression:

```
features = ["horsepower", "displacement"]
fitcars(lm, features)
```

MSE: 19.683, compared to variance 56.474

The error has decreased from the univariate case because we have a more capable model.

Finally, we try using 4 features as predictors. In order to help us compare the regression coefficients, we chain the model with a `StandardScaler` so that all columns are z-scores:

```
features = ["horsepower", "displacement", "cylinders", "weight"]
pipe = make_pipeline(StandardScaler(), lm)
fitcars(pipe, features)
```

MSE: 19.266, compared to variance 56.474

We did not get much improvement in the fit this time. But by comparing the coefficients of the individual features, some interesting information emerges:

```
weights = pd.Series(pipe[1].coef_, index=features).sort_values()
weights
```

	0
weight	-4.847433
horsepower	-1.892329
cylinders	-0.870727
displacement	0.722049

We now have evidence that *weight* is the most significant negative factor for MPG, by a wide margin.

In the next example we will see that we can create new features out of the ones that are initially given. Sometimes these new features add a lot to the quality of the regression.

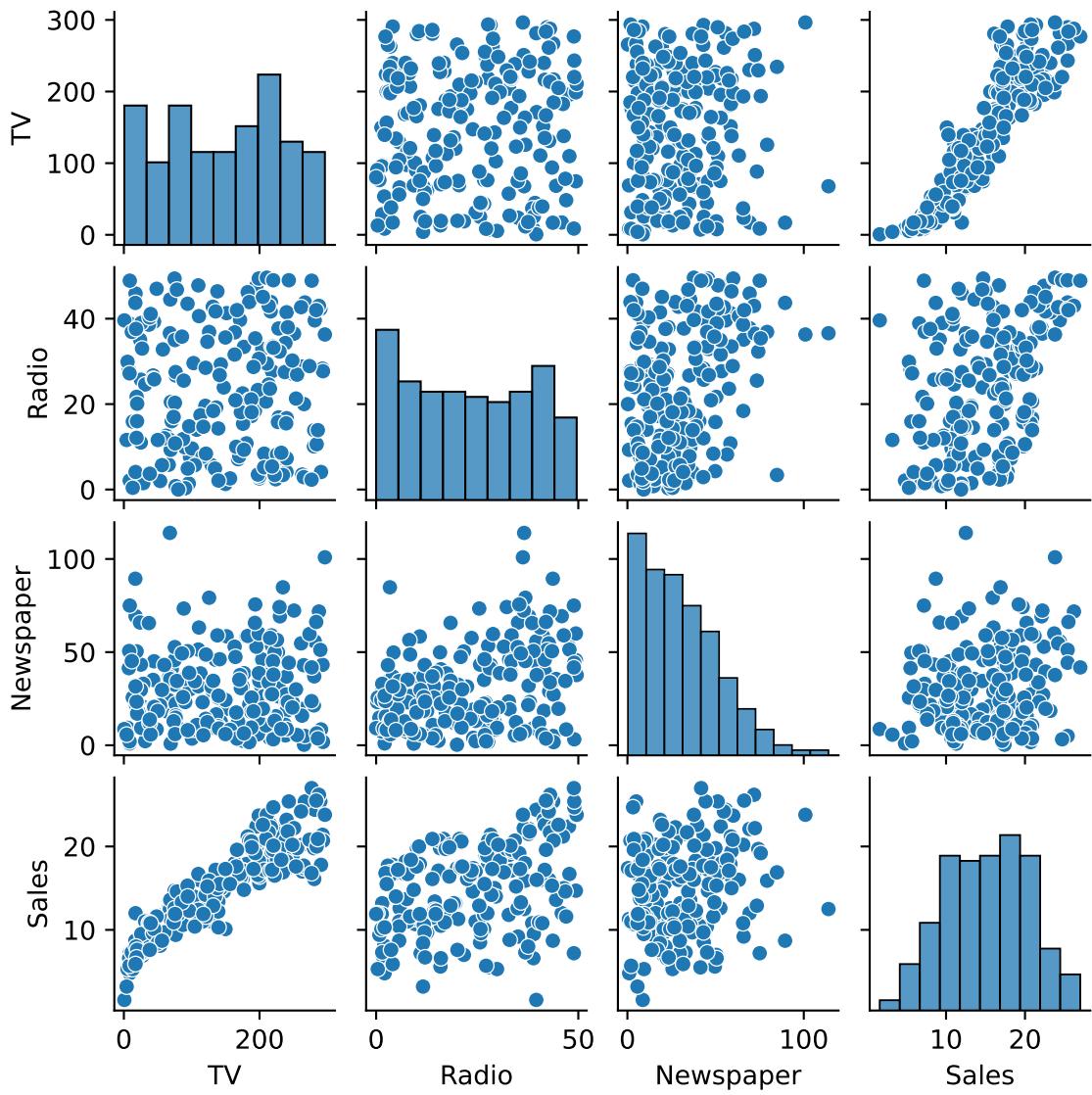
**Example 5.8.** Here we load data about advertising spending on different media in many markets:

```
ads = pd.read_csv("_datasets/advertising.csv")
ads.head(6)
```

	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9
5	8.7	48.9	75.0	7.2

Pairwise scatter plots yield some hints about what to expect from this dataset:

```
sns.pairplot(data=ads, height=1.5);
```



The last column, which shows relationships with *Sales*, is of greatest interest. From it we see that the clearest association between *Sales* and spending is with *TV*. So we first try a univariate linear fit of sales against *TV* spending alone:

```
X = ads[ ["TV"] ]      # has to be a frame, so ["TV"] not "TV"
y = ads["Sales"]

lm = LinearRegression()
lm.fit(X, y)
print("R^2 score:", f"{lm.score(X, y):.4f}")
```

```

print("Regression weight:", lm.coef_)

R^2 score: 0.8122
Regression weight: [0.05546477]

```

The coefficient of determination is already quite good. Since we are going to do multiple fits with different features, we write a function that does the grunt work:

```

def regress(lm, data, y, features):
    X = data[features]
    lm.fit(X, y)
    R2 = lm.score(X,y)
    print("R^2 score:", f"{R2:.5f}")
    print("Regression weights:")
    print( pd.Series(lm.coef_, index=features) )
    return None

```

Next, we try folding in *Newspaper* as well:

```
regress(lm, ads, y, ["TV", "Newspaper"])
```

```

R^2 score: 0.82364
Regression weights:
TV          0.055091
Newspaper   0.026021
dtype: float64

```

The additional feature had very little effect on the quality of fit. We go on to fit using all three features:

```
regress(lm, ads, y, ["TV", "Newspaper", "Radio"])
```

```

R^2 score: 0.90259
Regression weights:
TV          0.054446
Newspaper   0.000336
Radio       0.107001
dtype: float64

```

Judging by the weights of the model, it's even clearer now that we can explain *Sales* very well without contributions from *Newspaper*. In order to reduce model variance, it would be reasonable to leave that column out. Doing so has a negligible effect:

```
regress(lm, ads, y, ["TV", "Radio"])
```

```
R^2 score: 0.90259
Regression weights:
TV      0.054449
Radio   0.107175
dtype: float64
```

While we have a very good  $R^2$  score now, we can try to improve it. We can add an additional feature that is the product of *TV* and *Radio*, representing the possibility that these media reinforce one another's effects:

 Tip

In order to modify a frame, it has to be an independent copy, not just a subset of another frame.

```
X = ads[ ["Radio", "TV"] ].copy()
X["Radio*TV"] = X["Radio"]*X["TV"]
regress(lm, X, y, X.columns)
```

```
R^2 score: 0.91404
Regression weights:
Radio      0.042270
TV        0.043578
Radio*TV   0.000443
dtype: float64
```

We did see a small increase in the  $R^2$  score, and the combination of both types of spending does have a positive effect on *Sales*.

It's not uncommon to introduce a product term as done in Example 5.8, and more exotic choices are also possible. Keep in mind, though, that additional variables usually add variance to the model, even if they don't seriously affect the bias.

Interpreting linear regression is a major topic in statistics. There are tests that can lend much more precision and rigor to the brief discussion in this section.

[https://www.  
dropbox.com/  
s/  
armvar96hf03sta/  
Example5\\_6.  
mp4?raw=1](https://www.dropbox.com/s/armvar96hf03sta/Example5_6.mp4?raw=1)

### 5.2.1 Polynomial regression

A special case of multilinear regression is when there is initially a single predictor variable  $t$ , and then we define

$$x_1 = t^0, x_2 = t^1, \dots, x_d = t^{d-1}.$$

This makes the regressive approximation into

$$y \approx w_1 + w_2 t + \dots + w_d t^{d-1},$$

[https://www.  
dropbox.com/  
s/  
ghqex9s63odohja/  
Section5\\_2\\_  
1.mp4?raw=1](https://www.dropbox.com/s/ghqex9s63odohja/Section5_2_1.mp4?raw=1)

which is a polynomial of degree  $d - 1$ . This allows representation of data that depends on  $t$  in ways more complicated than a straight line. However, it can lead to overfitting if taken too far.

We don't have to add polynomial features manually, if we use a pipeline instead.

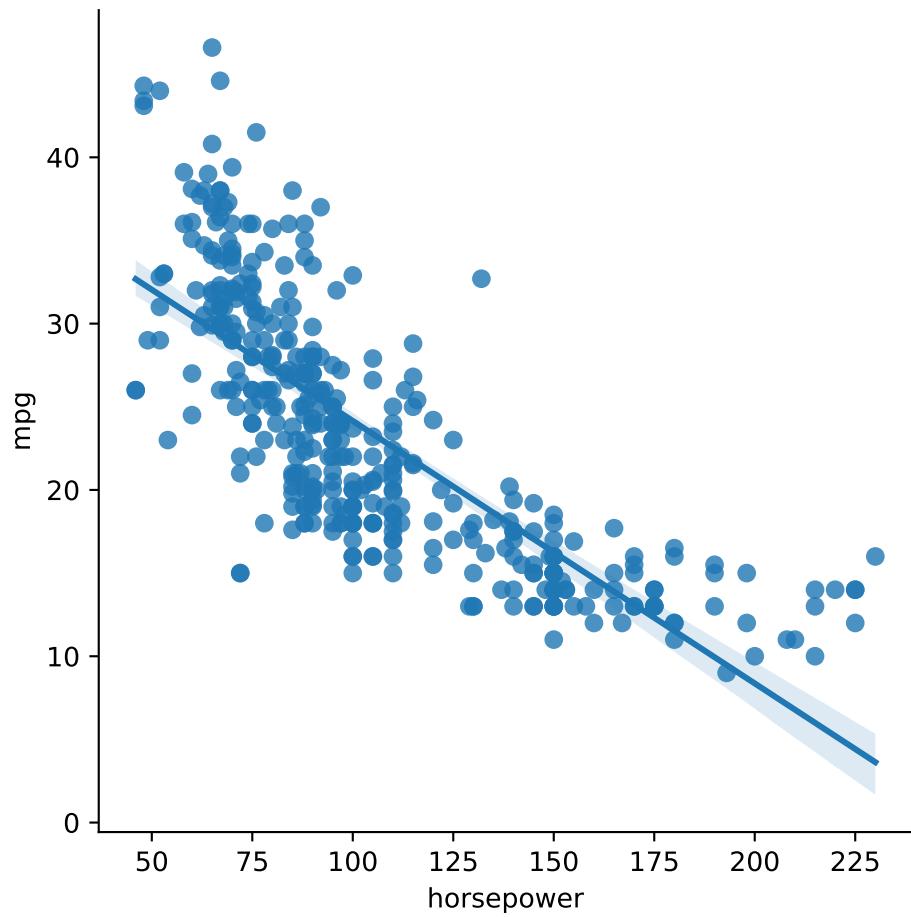
**Example 5.9.** We return to the data set regarding the fuel efficiency of cars:

```
cars = sns.load_dataset("mpg").dropna()  
cars.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chev
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylan
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth s
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel s
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino

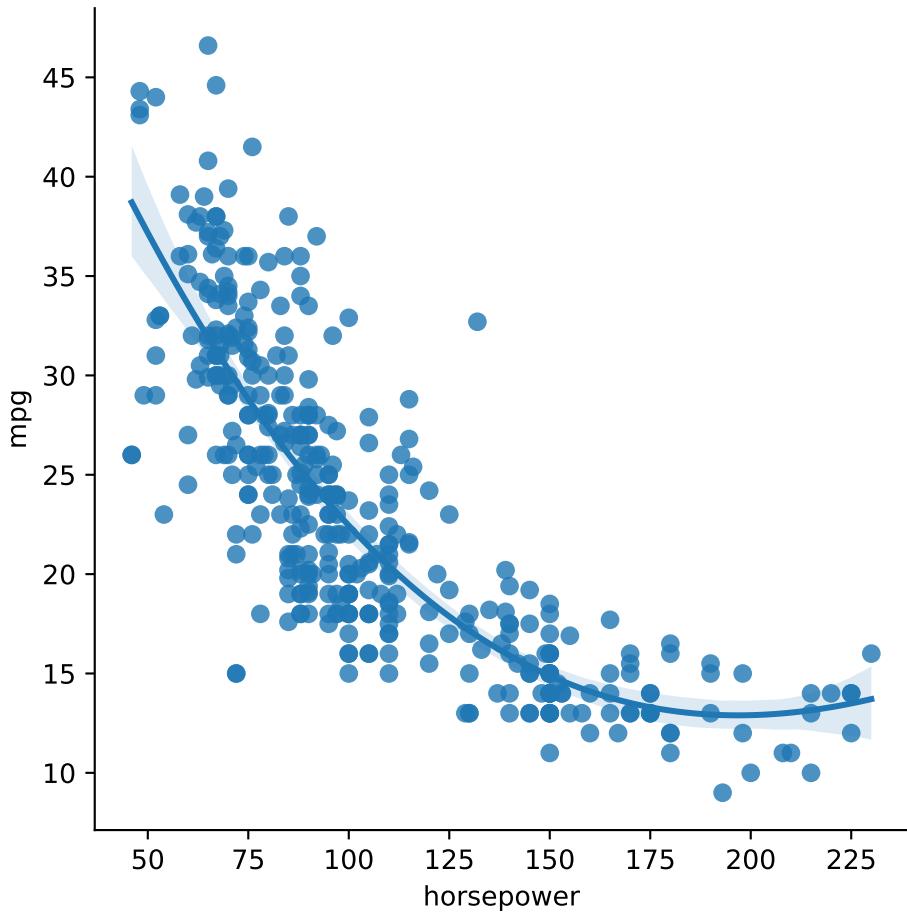
As we would expect, horsepower and miles per gallon are negatively correlated. However, the relationship is not well captured by a straight line:

```
sns.lmplot(data=cars, x="horsepower", y="mpg");
```



Seaborn can show us how a cubic polynomial regression would look:

```
sns.lmplot(data=cars, x="horsepower", y="mpg", order=3);
```



The figure above suggests that the cubic regression produces a better fit—that is, lower bias over the full range of horsepower.

In order to obtain the cubic fit within sklearn, we use the `PolynomialFeatures` preprocessor in a pipeline. If the original predictor variable is  $t$ , then the preprocessor will create features for  $1, t, t^2$ , and  $t^3$ . (Since the constant feature is added in by the preprocessor, we don't need it again within the regression, so we set `fit_intercept=False` in `LinearRegression`.)

```
from sklearn.preprocessing import PolynomialFeatures

X = cars[ ["horsepower"] ]
y = cars["mpg"]
lm = LinearRegression( fit_intercept=False )
cubic = make_pipeline(PolynomialFeatures(degree=3), lm)
cubic.fit(X, y)
```

```
# Predict MPG at horsepower=200:
query = pd.DataFrame([200], columns=X.columns)
print("prediction at hp=200:", cubic.predict(query))
```

prediction at hp=200: [12.90220247]

Let's compare the coefficients of determination for the linear and cubic fits:

```
linscore = LinearRegression().fit(X,y).score(X,y)
print(f"CoD for linear fit: {linscore:.4f}")
print(f"CoD for cubic fit: {cubic.score(X,y):.4f}")
```

CoD for linear fit: 0.605948  
 CoD for cubic fit: 0.688214

If a cubic polynomial can fit better than a line, it seems reasonable that increasing the degree more will lead to even better fits. In fact, the training error can only go down, because a lower-degree polynomial case is a subset of a higher-degree case. But there is a major catch, in the form of overfitting.

**Example 5.10.** Continuing with Example 5.9, we explore the effect of polynomial degree after splitting into training and test sets:

```
from sklearn.metrics import mean_squared_error

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=302
)

results = []
for deg in range(1,9):
    poly = make_pipeline(
        StandardScaler(),
        PolynomialFeatures(degree=deg),
        lm
    )
    poly.fit(X_train, y_train)
    MSE_tr = mean_squared_error(y_train, poly.predict(X_train))
    MSE_te = mean_squared_error(y_test, poly.predict(X_test))
```

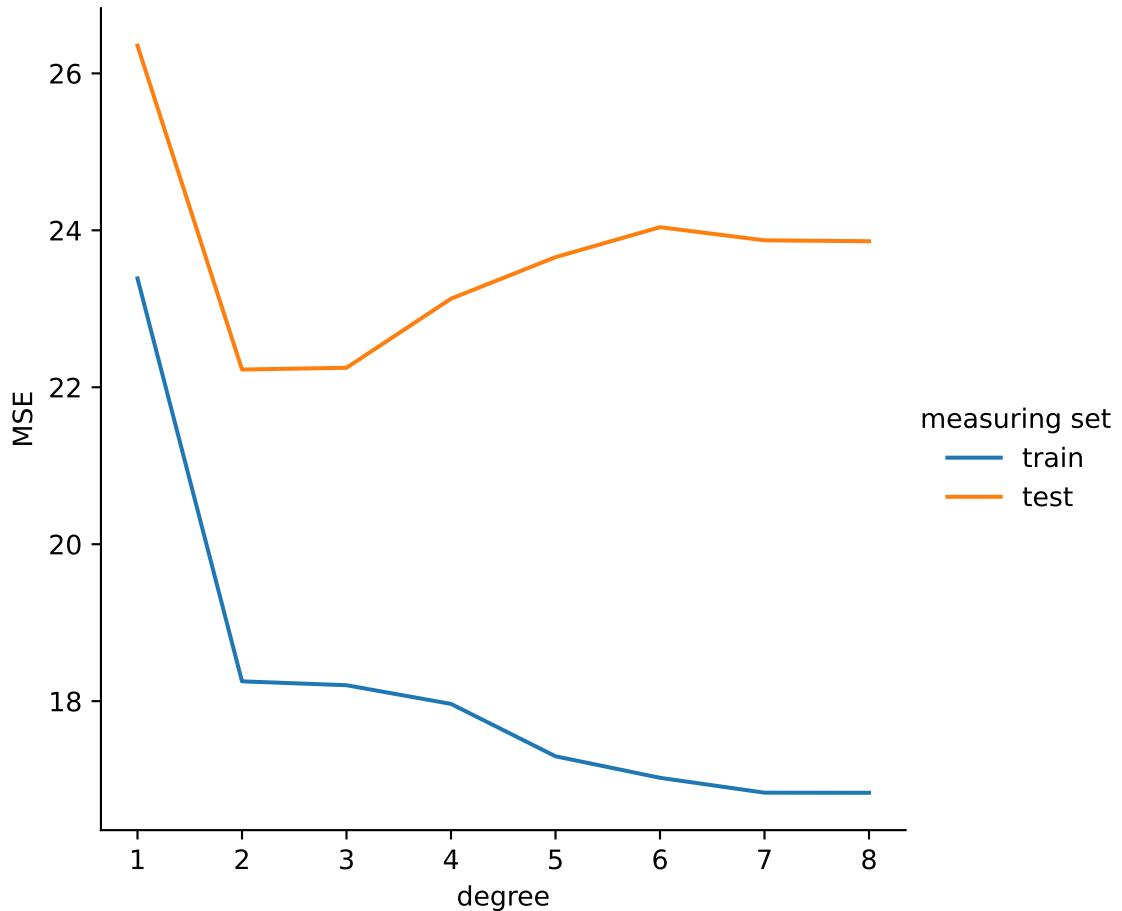
[https://www.  
dropbox.com/  
s/  
69fdzj8sn2rgdza/  
Example5\\_7.  
mp4?raw=1](https://www.dropbox.com/s/69fdzj8sn2rgdza/Example5_7.mp4?raw=1)

```

results.append((deg, "train", MSE_tr))
results.append((deg, "test", MSE_te))

results = pd.DataFrame(results, columns=["degree", "measuring set", "MSE"])
sns.relplot(data=results,
             x="degree", y="MSE",
             kind="line", hue="measuring set"
            );

```



The results above demonstrate that increasing the polynomial degree eventually leads to overfitting. In fact, in this case it seems that we don't want to go beyond the cubic fit.

Let's summarize the situation:

[https://www.  
dropbox.com/  
s/  
mgkf1r9wlrxeql/  
Example5\\_8.  
mp4?raw=1](https://www.dropbox.com/s/mgkf1r9wlrxeql/Example5_8.mp4?raw=1)

### 🔥 Caution

Adding polynomial features improves the expressive power of a regression model, which can decrease bias but also increase overfitting.

## 5.3 Regularization

As a general term, *regularization* refers to modifying something that is difficult to compute accurately with something more tractable. For learning models, regularization is a useful way to combat overfitting.

Suppose we had an  $\mathbb{R}^{n \times 4}$  feature matrix in which the features are identical; that is, the predictor variables satisfy  $x_1 = x_2 = x_3 = x_4$ , and suppose the target  $y$  also equals  $x_1$ . Clearly, we get a perfect regression if we use

$$y = 1x_1 + 0x_2 + 0x_3 + 0x_4.$$

[https://www.  
dropbox.com/  
s/  
lq2mydwh1z5074b/  
Section5\\_3.  
mp4?raw=1](https://www.dropbox.com/s/lq2mydwh1z5074b/Section5_3.mp4?raw=1)

But an equally good regression is

$$y = \frac{1}{4}x_1 + \frac{1}{4}x_2 + \frac{1}{4}x_3 + \frac{1}{4}x_4.$$

For that matter, so is

$$y = 1000x_1 - 500x_2 - 500x_3 + 1x_4.$$

A problem with more than one valid solution is called **ill-posed**. If we made tiny changes to the predictor variables in this thought experiment, the problem would technically be well-posed, but there would be a wide range of solutions that were very nearly correct, in which case the problem is said to be **ill-conditioned**; for practical purposes, it remains just as difficult.

The poor conditioning can be regularized away by modifying the least-squares loss function to penalize complexity in the model, in the form of excessively large regression coefficients. There are two dominant variants for doing this.

**Definition 5.8. Ridge regression** minimizes the loss function

$$L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \alpha \|\mathbf{w}\|_2^2, \quad (5.7)$$

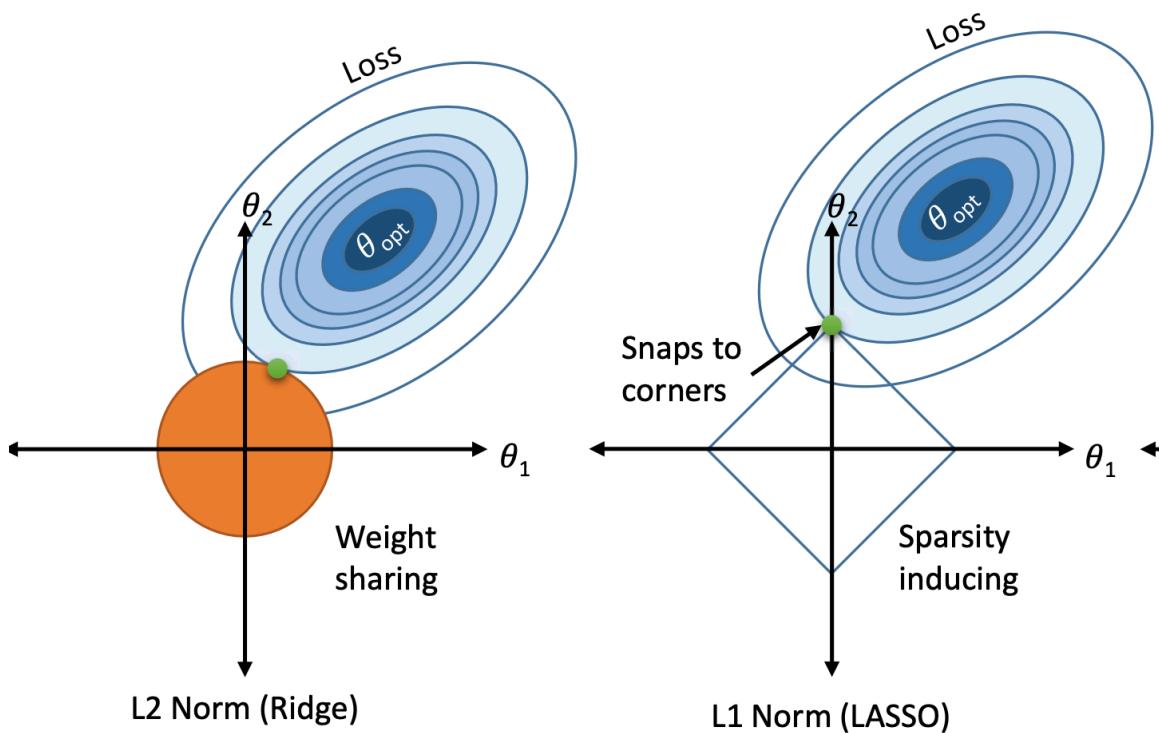
where  $\alpha$  is a nonnegative hyperparameter. **LASSO regression** minimizes the loss function

$$L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \alpha \|\mathbf{w}\|_1,$$

again for a nonnegative value of  $\alpha$ .

As  $\alpha \rightarrow 0$ , both types of regularization revert to the usual least-squares loss, but as  $\alpha \rightarrow \infty$ , the optimization becomes increasingly concerned with prioritizing a small result for  $\mathbf{w}$ .

While ridge regression is an easier function to minimize quickly, LASSO has an interesting advantage, as illustrated in this figure.



LASSO tends to produce *sparse* results, meaning that some of the regression coefficients are zero or negligible. These zeros indicate predictor variables that have minor predictive value, which can be valuable information in itself. Moreover, these variables can often be removed from the regression to reduce variance without noticeably affecting the bias.

**Example 5.11.** We'll apply regularized regression to data collected about the progression of diabetes:

```
diabetes = datasets.load_diabetes(as_frame=True) ["frame"]
diabetes.head(10)
```

	age	sex	bmi	bp	s1	s2	s3	s4	s5
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019907
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068332
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.002861
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022688
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031988
5	-0.092695	-0.044642	-0.040696	-0.019442	-0.068991	-0.079288	0.041277	-0.076395	-0.041176
6	-0.045472	0.050680	-0.047163	-0.015999	-0.040096	-0.024800	0.000779	-0.039493	-0.062917
7	0.063504	0.050680	-0.001895	0.066629	0.090620	0.108914	0.022869	0.017703	-0.035816
8	0.041708	0.050680	0.061696	-0.040099	-0.013953	0.006202	-0.028674	-0.002592	-0.014960
9	-0.070900	-0.044642	0.039062	-0.033213	-0.012577	-0.034508	-0.024993	-0.002592	0.067737

The features in this dataset were standardized, making it easy to compare the magnitudes of the regression coefficients.

First, we look at basic linear regression on all ten predictive features in the data:

```
X = diabetes.drop("target", axis=1)
y = diabetes["target"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2, random_state=2
)

from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(X_train, y_train)
print(f"linear model CoD score: {lm.score(X_test, y_test):.4f}")
```

linear model CoD score: 0.4399

First, we find that ridge regression can improve the score a bit:

```
from sklearn.linear_model import Ridge

rr = Ridge(alpha=0.5)
rr.fit(X_train, y_train)
print(f"ridge CoD score: {rr.score(X_test, y_test):.4f}")
```

ridge CoD score: 0.4411

Ridge regularization added a penalty for the 2-norm of the regression coefficients vector. Accordingly, the regularized solution has smaller coefficients:

```
from numpy.linalg import norm
print(f"2-norm of unregularized coefficients: {norm(lm.coef_):.1f}")
print(f"2-norm of ridge coefficients: {norm(rr.coef_):.1f}")
```

```
2-norm of unregularized coefficients: 1525.2
2-norm of ridge coefficients: 605.9
```

As we continue to increase the regularization parameter, the method becomes increasingly obsessed with keeping the coefficient vector small and pays ever less attention to the data:

```
for alpha in [0.25, 0.5, 1, 2]:
    rr = Ridge(alpha=alpha)      # more regularization
    rr.fit(X_train, y_train)
    print(f"alpha = {alpha:.2f}")
    print(f"2-norm of coefficient vector: {norm(rr.coef_):.1f}")
    print(f"ridge regression CoD score: {rr.score(X_test, y_test):.4f}")
    print()

alpha = 0.25
2-norm of coefficient vector: 711.7
ridge regression CoD score: 0.4527

alpha = 0.50
2-norm of coefficient vector: 605.9
ridge regression CoD score: 0.4411

alpha = 1.00
2-norm of coefficient vector: 480.8
ridge regression CoD score: 0.4078

alpha = 2.00
2-norm of coefficient vector: 353.5
ridge regression CoD score: 0.3478
```

**Example 5.12.** We continue with the diabetes data from Example 5.11, but this time, we try LASSO regularization. A validation curve suggests initial gains in the  $R^2$  score as the regularization parameter  $\alpha$  is varied, followed by a decrease:

[https://www.  
dropbox.com/  
s/  
20da7wu8istuoud/  
Example5\\_  
11.mp4?raw=  
1](https://www.dropbox.com/s/20da7wu8istuoud/Example5_11.mp4?raw=1)

```

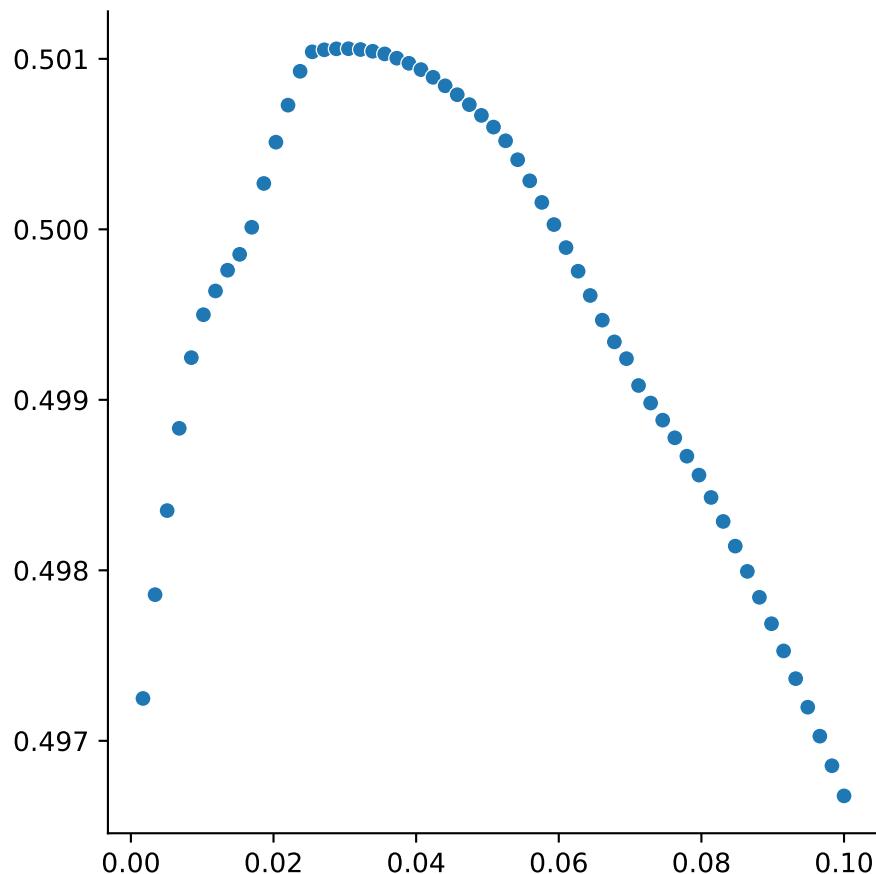
from sklearn.linear_model import Lasso

kf = KFold(n_splits=4, shuffle=True, random_state=302)
alpha = np.linspace(0, 0.1, 60)[1:] # exclude alpha=0

_, scores = validation_curve(
    Lasso(),
    X_train, y_train,
    cv=kf,
    n_jobs=-1,
    param_name="alpha", param_range=alpha
)

sns.relplot(x=alpha, y=np.mean(scores, axis=1));

```



Moreover, while ridge regression still used all of the features, LASSO put zero weight on

three of them:

```
lass = Lasso(alpha=0.05)
lass.fit(X_train, y_train)
pd.DataFrame( {
    "feature": X.columns,
    "ridge": rr.coef_,
    "LASSO": lass.coef_
} )
```

	feature	ridge	LASSO
0	age	43.113029	-0.000000
1	sex	-23.953301	-155.276227
2	bmi	199.535945	529.173009
3	bp	144.586873	313.419043
4	s1	25.977923	-132.507438
5	s2	2.751708	-0.000000
6	s3	-106.337626	-165.167100
7	s4	89.526889	0.000000
8	s5	185.660175	580.262391
9	s6	85.576399	30.557703

We can isolate the coefficients that were (within small rounding errors) zeroed out:

```
# Get the locations (indices) of the very small weights:
zeroed = np.nonzero( np.abs(lass.coef_) < 1e-5 )
# Names of the corresponding columns:
dropped = X.columns[zeroed].values
```

Now we can drop these features from the dataset:

```
X_train_reduced = X_train.drop(dropped, axis=1)
X_test_reduced = X_test.drop(dropped, axis=1)
```

Returning to a fit with no regularization, we find that little is lost by using the reduced feature set:

```
print(f"original linear model score: {lm.score(X_test, y_test):.4f}")

lm.fit(X_train_reduced, y_train)
Rsq = lm.score(X_test_reduced, y_test)
print(f"reduced linear model score: {Rsq:.4f}")
```

```
original linear model score: 0.4399
reduced linear model score: 0.4388
```

[https://www.  
dropbox.com/  
s/  
hismymdgpbddy0e/  
Example5\\_  
12.mp4?raw=1](https://www.dropbox.com/s/hismymdgpbddy0e/Example5_12.mp4?raw=1)

## 5.4 Nonlinear regression

Multilinear regression limits the representation of the dataset to a function of the form

$$\hat{f}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}. \quad (5.8)$$

This is a **linear** function, meaning that two key properties are satisfied. For all possible vectors  $\mathbf{u}, \mathbf{v}$  and numbers  $c$ ,

1.  $\hat{f}(\mathbf{u} + \mathbf{v}) = \hat{f}(\mathbf{u}) + \hat{f}(\mathbf{v}),$
2.  $\hat{f}(c\mathbf{u}) = c\hat{f}(\mathbf{u}).$

[https://www.  
dropbox.com/  
s/  
6ajvo8twg3ouj1d/  
Section5\\_4.  
mp4?raw=1](https://www.dropbox.com/s/6ajvo8twg3ouj1d/Section5_4.mp4?raw=1)

These properties are the essence of what makes a function easy to manipulate, solve for, and analyze.

**Example 5.13.** In two dimensions, the function  $\hat{f}(\mathbf{x}) = x_1 - x_2$  is linear. To prove this, first suppose that  $\mathbf{u}$  and  $\mathbf{v}$  are any 2-vectors. Then

$$\begin{aligned} \hat{f}(\mathbf{u} + \mathbf{v}) &= \hat{f}([u_1 + v_1, u_2 + v_2]) \\ &= (u_1 + u_2) - (v_1 + v_2) \\ &= (u_1 - v_1) + (u_2 - v_2) \\ &= \hat{f}(\mathbf{u}) + \hat{f}(\mathbf{v}). \end{aligned}$$

Now, suppose that  $\mathbf{u}$  is a 2-vector and  $c$  is any real number. Then

$$\begin{aligned} \hat{f}(c\mathbf{u}) &= \hat{f}([cu_1, cu_2]) \\ &= cu_1 - cu_2 \\ &= c(u_1 - u_2) \\ &= c \cdot \hat{f}(\mathbf{u}). \end{aligned}$$

**Example 5.14.** In two dimensions, the function  $\hat{f}(\mathbf{x}) = |x_1 + x_2|$  is not linear. Suppose  $\mathbf{u}$  is any 2-vector and  $c$  is any real number. If we try to prove the second property of linearity, we would start with:

$$\begin{aligned} \hat{f}(c\mathbf{u}) &= \hat{f}([cu_1, cu_2]) \\ &= |cu_1 + cu_2| \\ &= |c| \cdot |u_1 + u_2|. \end{aligned}$$

We are trying to show that this equals  $c \cdot \hat{f}(\mathbf{u})$ , which is  $c \cdot |u_1 + u_2|$ . However, it doesn't look as though this is equivalent to the last line above in all cases. In fact, they must be different if  $c < 0$  and  $|u_1 + u_2|$  is nonzero.

To prove *nonlinearity*, we only have to find one counterexample where one of the properties of a linear function doesn't hold. The attempt above suggests, for instance,  $\mathbf{u} = [1, 1]$  and the number  $c = -1$ . Then

$$\hat{f}(c\mathbf{u}) = |-1 - 1| = 2,$$

but at the same time,

$$c \cdot \hat{f}(\mathbf{u}) = -1 \cdot |1 + 1| = -2.$$

Since these values are different,  $\hat{f}$  can't be linear.

For our regression function Equation 5.8, the linearity properties follow easily from how the inner product is defined. For example,

$$\hat{f}(c\mathbf{u}) = (c\mathbf{u})^T \mathbf{w} = \sum_{i=1}^d (cu_i)w_i = c \sum_{i=1}^d u_i w_i = c(\mathbf{u}^T \mathbf{w}) = c\hat{f}(\mathbf{u}).$$

One major benefit of the linear approach is that the dependence of the weight vector  $\mathbf{w}$  on the regressed data is also linear, which makes solving for it straightforward.

As the simplest type of multidimensional function, linear relationships are a good first resort. Furthermore, we can augment the features with powers in order to get polynomial relationships. However, that approach becomes infeasible for more than 2 or 3 dimensions, because the number of polynomial terms needed explodes. (While there is a way around this restriction known as the *kernel trick*, that's beyond our mathematical scope here.)

Alternatively, we can resort to fully nonlinear regression methods. Two of them come from generalizations of our staple classifiers.

### 5.4.1 Nearest neighbors

To use kNN for regression, we find the  $k$  nearest examples as with classification, but replace voting on classes with the mean or median of the neighboring values. A simple example confirms that the resulting approximation is not linear.

**Example 5.15.** Suppose we have just two samples with one-dimensional features:  $x_1 = 0$  and  $x_2 = 2$ , and let the corresponding sample values be  $y_1 = 0$  and  $y_2 = 1$ . Using kNN with  $k = 1$ , the resulting approximation  $\hat{f}(x)$  is

$$\hat{f}(x) = \begin{cases} 0, & x < 1, \\ \frac{1}{2}, & x = 1, \\ 1, & x > 1. \end{cases}$$

[https://www.  
dropbox.com/  
s/  
762plrk0qqw501w/  
Section5\\_4\\_  
1.mp4?raw=1](https://www.dropbox.com/s/762plrk0qqw501w/Section5_4_1.mp4?raw=1)

(Convince yourself that the result is the same whether the mean or the median is used.) Thus, for instance,  $\hat{f}(2 \cdot 0.6) = \hat{f}(1.2) = 1$ , while  $2 \cdot \hat{f}(0.6) = 0$ .

kNN regression can produce a function that conforms itself to the training data much more closely than a linear regressor does. This can both decrease bias and increase variance, especially for small values of  $k$ . As illustrated in the following video, increasing  $k$  flattens out the approximation, decreasing variance while increasing bias.

[\\_media/knn\\_regression.mp4](#)

As with classification, we can choose the norm to use and whether to weight the neighbors equally or by inverse distance. As a reminder, it is usually advisable to work with z-scores for the features rather than raw data.

**Example 5.16.** We return again to the dataset of cars and their fuel efficiency. A linear regression on four quantitative features is only OK:

```
cars = sns.load_dataset("mpg").dropna()
features = ["displacement", "horsepower", "weight", "acceleration"]
X = cars[features]
y = cars["mpg"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2, random_state=0
)

lm = LinearRegression()
lm.fit(X_train, y_train)
print(f"linear model CoD: {lm.score(X_test, y_test):.4f}")
```

linear model CoD: 0.6928

Next we try a kNN regressor, doing a grid search to find good hyperparameters:

```
from sklearn.neighbors import KNeighborsRegressor

kf = KFold(n_splits=6, shuffle=True, random_state=3383)
grid = {
    "kneighborsregressor__n_neighbors": range(2, 25),
    "kneighborsregressor__weights": ["uniform", "distance"]
}
knn = make_pipeline(StandardScaler(), KNeighborsRegressor())
```

```

optim = GridSearchCV(
    knn, grid,
    cv=kf,
    n_jobs=-1
)
optim.fit(X_train, y_train)

print(f"best kNN CoD: {optim.score(X_test, y_test):.4f}")

```

best kNN CoD: 0.7439

As you can see above, we got some improvement over the linear regressor.

[https://www.  
dropbox.com/  
s/  
2v4wgkm8r2915aq/  
Example5\\_  
14.mp4?raw=1](https://www.dropbox.com/s/2v4wgkm8r2915aq/Example5_14.mp4?raw=1)

### 5.4.2 Decision tree

Recall that a decision tree recursively divides the examples into subsets. As with kNN regression, we can replace taking a classification vote over a leaf subset with taking a mean or median of the values in the leaf. As in classification, a decision tree regressor seeks the best possible split of samples along coordinate planes. A proposal to split into subsets  $S$  and  $T$  is assigned the weighted score

$$Q = |S|H(S) + |T|H(T),$$

where  $H$  is an empirical error measure. The split location is chosen to minimize  $Q$ . After the process is applied recursively, each leaf of the decision tree contains a subset of the training samples. There are two common variations:

1. The mean of the leaf labels is the regression value, and  $H$  is mean square error from the prediction.
2. The median of the leaf labels is the regression value, and  $H$  is the mean absolute error from the prediction.

[https://www.  
dropbox.com/  
s/  
741v6g7skp3a9ct/  
Section5\\_4\\_  
2.mp4?raw=1](https://www.dropbox.com/s/741v6g7skp3a9ct/Section5_4_2.mp4?raw=1)

**Example 5.17.** Suppose we are given the observations  $x_i = i$ ,  $i = 1, \dots, 4$ , where  $y_1 = 2$ ,  $y_2 = -1$ ,  $y_3 = 1$ ,  $y_4 = 0$ . Let's find the decision tree regressor using medians.

The original value set has median  $\frac{1}{2}$ , so its MAE is

$$\frac{1}{4} \left( |2 - \frac{1}{2}| + |-1 - \frac{1}{2}| + |1 - \frac{1}{2}| + |0 - \frac{1}{2}| \right) = \frac{1}{4} \cdot \frac{8}{2} = 1.$$

Thus, the  $Q$ -score of the full set is 4.

There are three allowable ways to split the data, as it is ordered from left to right:

- $S = \{2\}$ ,  $T = \{-1, 1, 0\}$ . Note that  $|S| = 1$  and  $|T| = 3$ , so

$$\begin{aligned} Q &= 1 \left[ \frac{1}{1} |2 - 2| \right] + 3 \left[ \frac{1}{3} (|-1 - 0| + |1 - 0| + |0 - 0|) \right] \\ &= 0 + 2 = 2. \end{aligned}$$

- $S = \{2, -1\}$ ,  $T = \{1, 0\}$

$$\begin{aligned} Q &= 2 \left[ \frac{1}{2} (|2 - \frac{1}{2}| + |-1 - \frac{1}{2}|) \right] + 2 \left[ \frac{1}{2} (|1 - \frac{1}{2}| + |0 - \frac{1}{2}|) \right] \\ &= 3 + 1 = 4. \end{aligned}$$

- $S = \{2, -1, 1\}$ ,  $T = \{0\}$

$$\begin{aligned} Q &= 3 \left[ \frac{1}{3} (|2 - 1| + |-1 - 1| + |1 - 1|) \right] + 1 \left[ \frac{1}{1} |0 - 0| \right] \\ &= 3 + 0 = 3. \end{aligned}$$

Thus, the first split above produces the smallest  $Q$ , and it improves on the original set.

**Example 5.18.** Here is some simple 2D data:

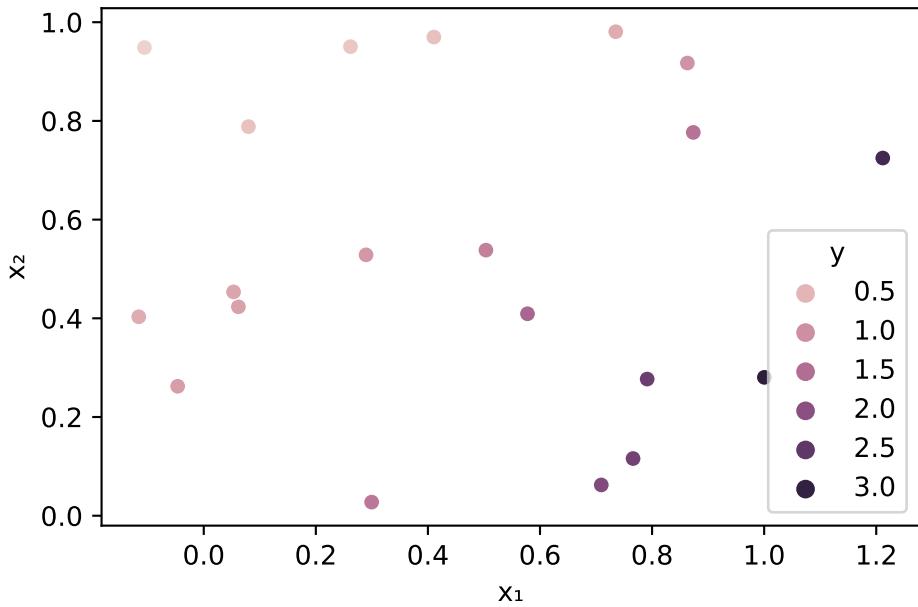
```

rng = default_rng(1)
x1 = rng.random((10,2))
x1[:,0] -= 0.25
x2 = rng.random((10,2))
x2[:,0] += 0.25
X = np.vstack((x1,x2))
y = np.exp( X[:,0]-2*X[:,1]**2+X[:,0]*X[:,1] )

df = pd.DataFrame({"x": X[:,0], "x": X[:,1], "y": y})
sns.scatterplot(data=df, x="x", y="x", hue="y");

```

[https://www.  
dropbox.com/  
s/  
vb4dmpbq358dxwl/  
Example5\\_  
15.mp4?raw=  
1](https://www.dropbox.com/s/vb4dmpbq358dxwl/Example5_15.mp4?raw=1)

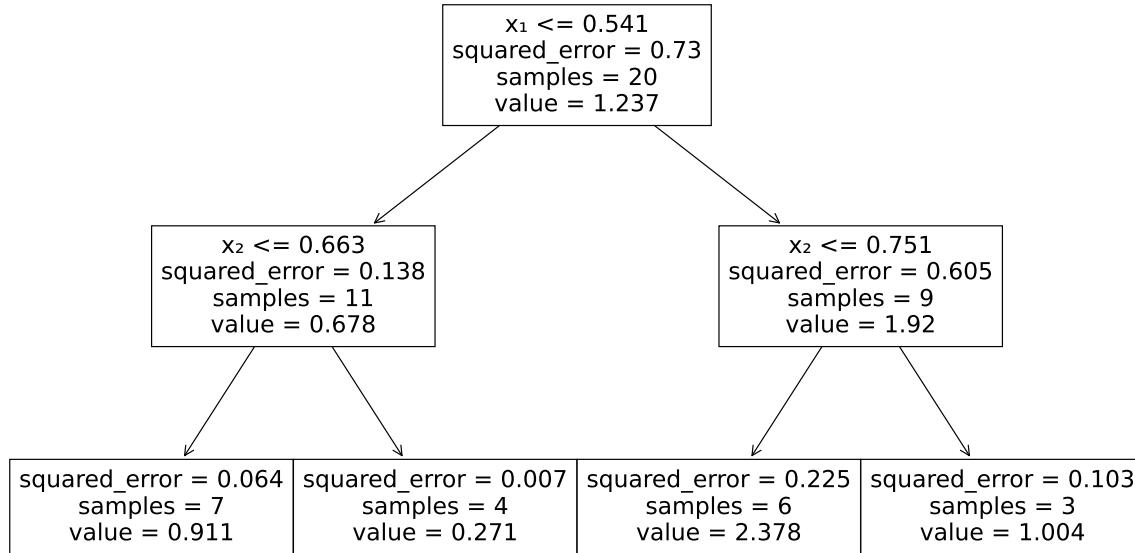


The default in `sklearn` is to use means on leaves and MSE (called `squared_error` in `sklearn`) as the quality measure. Here is a shallow tree fitted to the data:

```
from sklearn.tree import DecisionTreeRegressor, plot_tree
dtree = DecisionTreeRegressor(max_depth=2)
dtree.fit(X, y)
```

```
DecisionTreeRegressor(max_depth=2)
```

```
from matplotlib.pyplot import figure
figure(figsize=(17,10), dpi=160)
plot_tree(dtree, feature_names=["x ", "x "]);
```



All of the original samples end up in one of the four leaves. We can find out the tree node number that each sample ends up at using `apply`:

```
leaf = dtree.apply(X)
print(leaf)
```

```
[3 3 2 5 2 2 3 2 2 2 5 6 5 5 3 5 6 6 2 5]
```

From the above we deduce that the leaves are the nodes numbered 2, 3, 5, and 6. With some pandas grouping, we can find out the mean value for the samples within each of these:

```
leaves = pd.DataFrame( {"y": y, "leaf": leaf} )
leaves.groupby("leaf")["y"].mean()
```

leaf	y
2	0.911328
3	0.270725
5	2.378427
6	1.003786

All values of the regressor will be one of the four values above. This is exactly what is done internally by the `predict` method of the regressor:

```
print( dtree.predict(X) )
```

```
[0.27072468 0.27072468 0.91132782 2.37842709 0.91132782 0.91132782  
 0.27072468 0.91132782 0.91132782 0.91132782 2.37842709 1.00378567  
 2.37842709 2.37842709 0.27072468 2.37842709 1.00378567 1.00378567  
 0.91132782 2.37842709]
```

**Example 5.19.** Continuing with the data from Example 5.16, we find that we can do even better with a random forest of decision tree regressors:

[https://www.  
dropbox.com/  
s/  
wf0djkbbadjtytx/  
Example5\\_  
16.mp4?raw=  
1](https://www.dropbox.com/s/wf0djkbbadjtytx/Example5_16.mp4?raw=1)

```
X = cars[features]  
y = cars["mpg"]  
  
X_train, X_test, y_train, y_test = train_test_split(  
    X, y,  
    test_size=0.2, random_state=302  
)  
  
from sklearn.ensemble import RandomForestRegressor  
  
grid = {  
    "max_depth": range(3, 8),  
    "max_samples": np.arange(0.2, 0.6, 0.1),  
}  
knn = RandomForestRegressor(n_estimators=60)  
optim = GridSearchCV(  
    knn,  
    grid,  
    cv=kf,  
    n_jobs=-1  
)  
optim.fit(X_train, y_train)  
  
print(f"best forest CoD: {optim.score(X_test, y_test):.4f}")
```

```
best forest CoD: 0.7331
```

## 5.5 Logistic regression

Sometimes a regressed value is subject to certain known bounds or other conditions. A major example is probability, which has to be between 0 and 1 (inclusive).

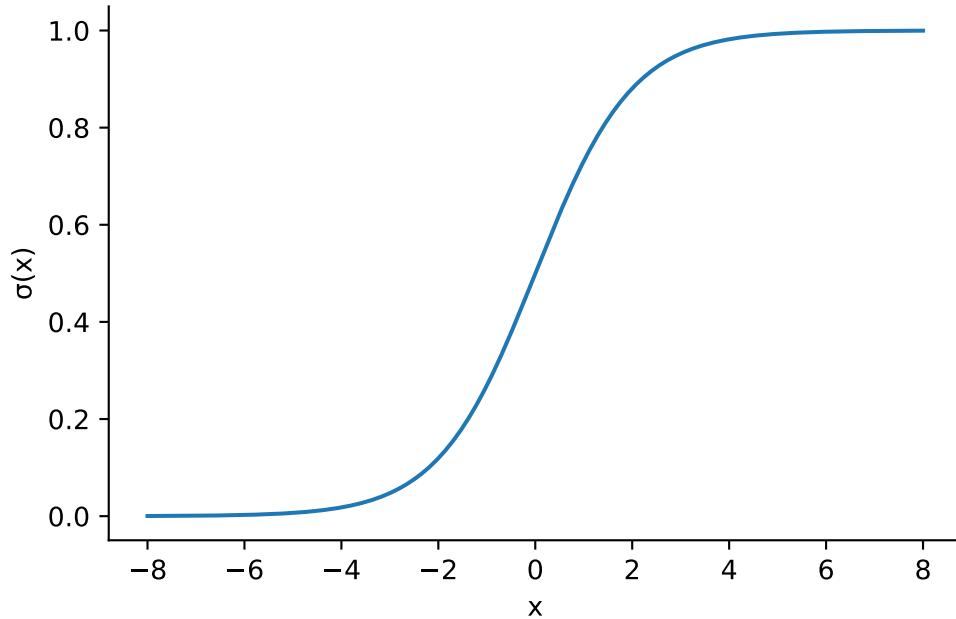
A linear regressor,  $\hat{f}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$  for a constant vector  $\mathbf{w}$ , typically ranges over all of  $(-\infty, \infty)$ . In order to get a result that must lie within  $[0, 1]$ , we can transform its output.

**Definition 5.9.** The **logistic function** is

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

defined for all real values of  $x$ .

The logistic function takes the form of a smoothed step increasing from 0 to 1:



[https://www.  
dropbox.com/  
s/  
c5xsfxyw2ie4hkl/  
Section5\\_5.  
mp4?raw=1](https://www.dropbox.com/s/c5xsfxyw2ie4hkl/Section5_5.mp4?raw=1)

Given samples of a probability variable  $p(\mathbf{x})$ , the regression task is to find a weight vector  $\mathbf{w}$  so that

$$p \approx \sigma(\mathbf{x}^T \mathbf{w}).$$

The result is known as **logistic regression**. A common way to use logistic regression is for binary classification. Suppose we have training samples  $(\mathbf{x}_i, y_i)$ ,  $i = 1, \dots, n$ , where for each  $i$  either  $y_i = 0$  or  $y_i = 1$ . The resulting approximation to  $p$  at some query  $\mathbf{x}$  can then be interpreted as the probability of observing a 1 at  $\mathbf{x}$ .

In order to fully specify the regressor, we need to specify a loss function to be optimized.

### 5.5.1 Loss function

Defining  $\hat{p}_i = \sigma(\mathbf{x}_i^T \mathbf{w})$  at all the training points, a straightforward loss function would be

$$\sum_{i=1}^n (\hat{p}_i - y_i)^2.$$

However, it's more common to use the **cross-entropy** loss function

$$L(\mathbf{w}) = -\sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]. \quad (5.9)$$

(The logarithms in Equation 5.9 can be in any base, since that choice only affects  $L$  by a constant factor.) In cross-entropy loss, sample  $i$  contributes

$$-\log(1 - \hat{p}_i)$$

if  $y_i = 0$ , which becomes infinite as  $\hat{p}_i \rightarrow 1^-$ , and

$$-\log(\hat{p}_i)$$

if  $y_i = 1$ , which becomes infinite as  $\hat{p}_i \rightarrow 0^+$ . Thus, there is a steep penalty for being almost completely wrong about an observation.

**Example 5.20.** Suppose we have true labels  $\mathbf{y} = [0, 0, 0, 1]$  and predicted probabilities  $\hat{\mathbf{p}} = [0.1, 0.4, 0.2, 0.3]$ . This gives the following terms in the cross-entropy sum:

$$\begin{aligned} y_1 \log(\hat{p}_1) + (1 - y_1) \log(1 - \hat{p}_1) &= \log(0.9), \\ y_2 \log(\hat{p}_2) + (1 - y_2) \log(1 - \hat{p}_2) &= \log(0.6), \\ y_3 \log(\hat{p}_3) + (1 - y_3) \log(1 - \hat{p}_3) &= \log(0.8), \\ y_4 \log(\hat{p}_4) + (1 - y_4) \log(1 - \hat{p}_4) &= \log(0.3). \end{aligned}$$

Hence the total cross-entropy is

$$-\log[(0.9)(0.6)(0.8)(0.3)] = -\log(0.1296).$$

Using the natural log, this is about 2.043.

Logistic regression does have a major disadvantage compared to linear regression: the minimization of loss does *not* lead to a linear problem for the weight vector  $\mathbf{w}$ . The difference in practice is usually not a concern, though.

Cross-entropy is the default loss function for a `LogisticRegression` in scikit-learn. In Example 5.21, we use logistic regression on a binary classification problem, where we interpret the regression output as the probability of being in class 1, as opposed to class 0.

**Example 5.21.** We will try logistic regression for a simple spam filter. The data set is based on work and personal emails for one individual. The features are calculated word and character frequencies, as well as the appearance of capital letters.

[https://www.  
dropbox.com/  
s/  
wdnqanl9ny5h9p7/  
Section5\\_5\\_  
1.mp4?raw=1](https://www.dropbox.com/s/wdnqanl9ny5h9p7/Section5_5_1.mp4?raw=1)

```
spam = pd.read_csv("_datasets/spambase.csv")
spam.head()
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq
0	0.00	0.64	0.64	0.0	0.32	
1	0.21	0.28	0.50	0.0	0.14	
2	0.06	0.00	0.71	0.0	1.23	
3	0.00	0.00	0.00	0.0	0.63	
4	0.00	0.00	0.00	0.0	0.63	

The labels in this case are 0 for “ham” (not spam) and 1 for “spam”. We create a feature matrix and label vector, and split into train/test sets:

```
X = spam.drop("class", axis="columns")
y = spam["class"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=19716
)
```

We fit a logistic regression just like any other learner. (The meaning of the argument `penalty` in the call below is explained in Section 5.5.2.)

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(penalty="none")
logreg.fit(X_train, y_train)

LogisticRegression(penalty='none')
```

Despite the name, though, scikit-learn treats a `LogisticRegression` object like a classifier, and its predictions are true/false:

```
logreg.predict( X_test.iloc[:5,:] )

array([1, 0, 0, 0, 0])
```

The default scoring function is classification accuracy:

```

acc = logreg.score(X_test, y_test)
print(f"spam classification accuracy is {acc:.2%}")

spam classification accuracy is 93.59%

```

The actual logistic outputs are the *probabilistic* predictions:

```

logreg.predict_proba( X_test.iloc[:5,:] )

array([[4.65419538e-01, 5.34580462e-01],
       [9.87165442e-01, 1.28345580e-02],
       [5.58611611e-01, 4.41388389e-01],
       [7.54138243e-01, 2.45861757e-01],
       [1.00000000e+00, 3.21186479e-12]])

```

From these, we could use ROC and AUC metrics as we did in Section 3.5.

Let's repeat the computation with a standardization pipeline, so that we can easily compare the magnitudes of the weights:

```

pipe = make_pipeline(StandardScaler(), logreg)
pipe.fit(X_train, y_train)

weights = pd.Series( logreg.coef_[0], index=X.columns)
weights = weights.sort_values()

print("most hammy features:")
print(weights[:4])
print()
print("most spammy features:")
print(weights[-4:])

```

```

most hammy features:
word_freq_george    -30.595550
word_freq_cs        -12.567214
word_freq_hp         -3.609056
word_freq_meeting   -1.606267
dtype: float64

most spammy features:
word_freq_remove      0.864465

```

```

char_freq_%24          1.104584
capital_run_length_longest 1.759774
word_freq_3d            4.509272
dtype: float64

```

We see above that the word “george” is a strong counter-indicator for spam; remember that this data set comes from an individual’s inbox. Its presence makes the inner product  $\mathbf{x}^T \mathbf{w}$  more negative, which drives the logistic function closer to 0. Conversely, the presence of consecutive capital letters increases the inner product and pushes the probability of spam closer to 1.

[https://www.  
dropbox.com/  
s/  
zpu18qxdgzkwskn/  
Example5\\_  
19.mp4?raw=  
1](https://www.dropbox.com/s/zpu18qxdgzkwskn/Example5_19.mp4?raw=1)

### 5.5.2 Regularization

As with other forms of regression, the loss function may be regularized using the ridge or LASSO penalty. The standard formulation is

$$\tilde{L}(\mathbf{w}) = C L(\mathbf{w}) + \|\mathbf{w}\|, \quad (5.10)$$

where the vector norm is either the 1-norm (LASSO style) or 2-norm (ridge style). In either case,  $C$  is a positive hyperparameter. For  $C$  close to zero, the regression is mainly concerned with the penalty term, and as  $C$  increases, the regression gradually pays more and more attention to the data, at the expense of the penalty term.

! Important

The parameter  $C$  functions like the inverse of the regularization parameter  $\alpha$  we used in the linear regressor. It’s because of a different convention chosen historically.

**Example 5.22.** We will continue Example 5.21 with an exploration of regularization. When using norm-based regularization, it’s good practice to standardize the variables, so we will use a standardization pipeline:

In Example 5.21, we disabled regularization. The default call uses 2-norm regularization with  $C = 1$ , which, in this case, improves the accuracy a bit:

💡 Tip

The use of the `solver` keyword argument in `LogisticRegression` is optional, but the default choice does not work with the LASSO-style penalty term.

```

logreg = LogisticRegression(solver="liblinear")
logreg.fit(X_train, y_train)
acc = logreg.score(X_test, y_test)
print(f"accuracy with default regularization is {acc:.2%}")

```

accuracy with default regularization is 93.81%

A validation-based grid search is one way to look for the optimal regularization. Usually, we want the grid values of  $C$  to be spaced exponentially, not equally. For example,

```

10 ** np.linspace(-2, 2, 13)

array([1.0000000e-02, 2.15443469e-02, 4.64158883e-02, 1.0000000e-01,
       2.15443469e-01, 4.64158883e-01, 1.0000000e+00, 2.15443469e+00,
       4.64158883e+00, 1.0000000e+01, 2.15443469e+01, 4.64158883e+01,
       1.0000000e+02])

```

We use that strategy along with a search over both the 2-norm and the 1-norm for the regularization penalty term:

```

grid = { "logisticregression__C": 10 ** np.linspace(-1, 4, 40),
         # ridge and LASSO cases:
         "logisticregression__penalty": ["l2", "l1"]
     }

learner = make_pipeline(
    StandardScaler(),
    LogisticRegression( solver="liblinear" )
)

kf = StratifiedKFold(n_splits=6, shuffle=True, random_state=302)

search = GridSearchCV(
    learner, grid,
    cv=kf,
    n_jobs=-1
)

search.fit(X_train, y_train)

print("Best parameters:")

```

```

print(search.best_params_)
print()
print(f"Best score is {search.best_score_:.2%}")

Best parameters:
{'logisticregression__C': 1.0608183551394483, 'logisticregression__penalty': 'l1'}

Best score is 92.55%

```

In this case, we failed to improve on the default regularization. (Recall that cross-validation means that each learner is trained on less data, so the grid metrics might not be completely accurate.)

[https://www.dropbox.com/s/yr4mi1zo8nh0jf8/Example5\\_20.mp4?raw=1](https://www.dropbox.com/s/yr4mi1zo8nh0jf8/Example5_20.mp4?raw=1)

### 5.5.3 Multiclass classification

When there are more than two unique labels possible in a classification, logistic regression can be extended through the one-vs-rest (OVR) paradigm we have used previously. Given  $K$  classes, there are  $K$  binary regressors fit for the outcomes “class 1/not class 1,” “class 2/not class 2,” and so on. These give  $K$  different weight vectors,  $\mathbf{w}_1, \dots, \mathbf{w}_K$ .

For a query vector  $\mathbf{x}$ , we can predict the probabilities for it being in each class:

$$\hat{q}_k(\mathbf{x}) = \sigma(\mathbf{x}^T \mathbf{w}_k), \quad k = 1, \dots, K.$$

However, since the regressions were performed separately, there is no reason to think the  $q_k$  are probabilities that sum to 1 over all the classes. So we must normalize them:

$$\hat{p}_k(\mathbf{x}) = \frac{\hat{q}_k(\mathbf{x})}{\sum_{k=1}^K \hat{q}_k(\mathbf{x})}.$$

Now we can interpret  $\hat{p}_k(\mathbf{x})$  as the probability of  $\mathbf{x}$  belonging to class  $k$ .

**Example 5.23.** Suppose we have  $d = 2$  features and three classes, and that the three logistic regressions gave the weight vectors

$$\mathbf{w}_1 = [-2, 0], \quad \mathbf{w}_2 = [1, 1], \quad \mathbf{w}_3 = [0, 1].$$

For the query  $\mathbf{x} = [1, 0]$ , we get the predictions

$$\begin{aligned}\hat{q}_1 &= \sigma(\mathbf{x}^T \mathbf{w}_1) = \sigma((1)(-2) + (0)(0)) \approx 0.11920, \\ \hat{q}_2 &= \sigma(\mathbf{x}^T \mathbf{w}_2) = \sigma((1)(1) + (0)(1)) \approx 0.73106, \\ \hat{q}_3 &= \sigma(\mathbf{x}^T \mathbf{w}_3) = \sigma((1)(0) + (0)(1)) = 0.5.\end{aligned}$$

So, if we ask, “How much does this  $\mathbf{x}$  look like class 1?” the answer is, “11.9%”, for “How much like class 2?” it’s “73.1%”, and so on. Since there are three exclusive options for the class of  $\mathbf{x}$ , the probabilities assigned to the classes are

$$\begin{aligned}\hat{p}_1 &= \frac{\hat{q}_1}{\hat{q}_1 + \hat{q}_2 + \hat{q}_3} \approx 0.08828, \\ \hat{p}_2 &= \frac{\hat{q}_2}{\hat{q}_1 + \hat{q}_2 + \hat{q}_3} \approx 0.54142, \\ \hat{p}_3 &= \frac{\hat{q}_3}{\hat{q}_1 + \hat{q}_2 + \hat{q}_3} \approx 0.37030.\end{aligned}$$

These probabilities, in exact arithmetic, add up to 100%.

The situation is now the same as for probabilistic classification in Section 3.5. Over a testing set, we get a matrix of probabilities. Each of the rows gives the class probabilities at a single query point. We can simply select the most likely class at each test query, or use ROC and AUC to better understand the probabilistic results.

**Example 5.24.** As a multiclass example, we return to the dataset for classifying forest cover:

```
forest = datasets.fetch_covtype()
X = forest["data"][:250000,:12]    # 250,000 samples, 12 features
y = forest["target"][:250000]
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.15,
    shuffle=True, random_state=302
)

logr = LogisticRegression(solver="liblinear")
pipe = make_pipeline(StandardScaler(), logr)
pipe.fit(X_train, y_train)
print(f"accuracy score is {pipe.score(X_test, y_test):.2%}")

accuracy score is 72.74%
```

We can now look at probabilistic predictions for each class:

```

p_hat = pipe.predict_proba(X_test)
p_hat[:3]

array([[3.27753115e-01, 6.71344154e-01, 8.83821740e-06, 1.47289531e-05,
       6.20692039e-04, 2.57492056e-04, 9.79562864e-07],
       [4.20545770e-01, 5.66374061e-01, 2.90422085e-06, 4.50982065e-07,
       1.40842869e-03, 8.15605577e-06, 1.16602291e-02],
       [2.40111415e-01, 7.59067843e-01, 1.21609701e-05, 1.42404726e-05,
       3.93333619e-04, 3.98538413e-04, 2.46805163e-06]])

```

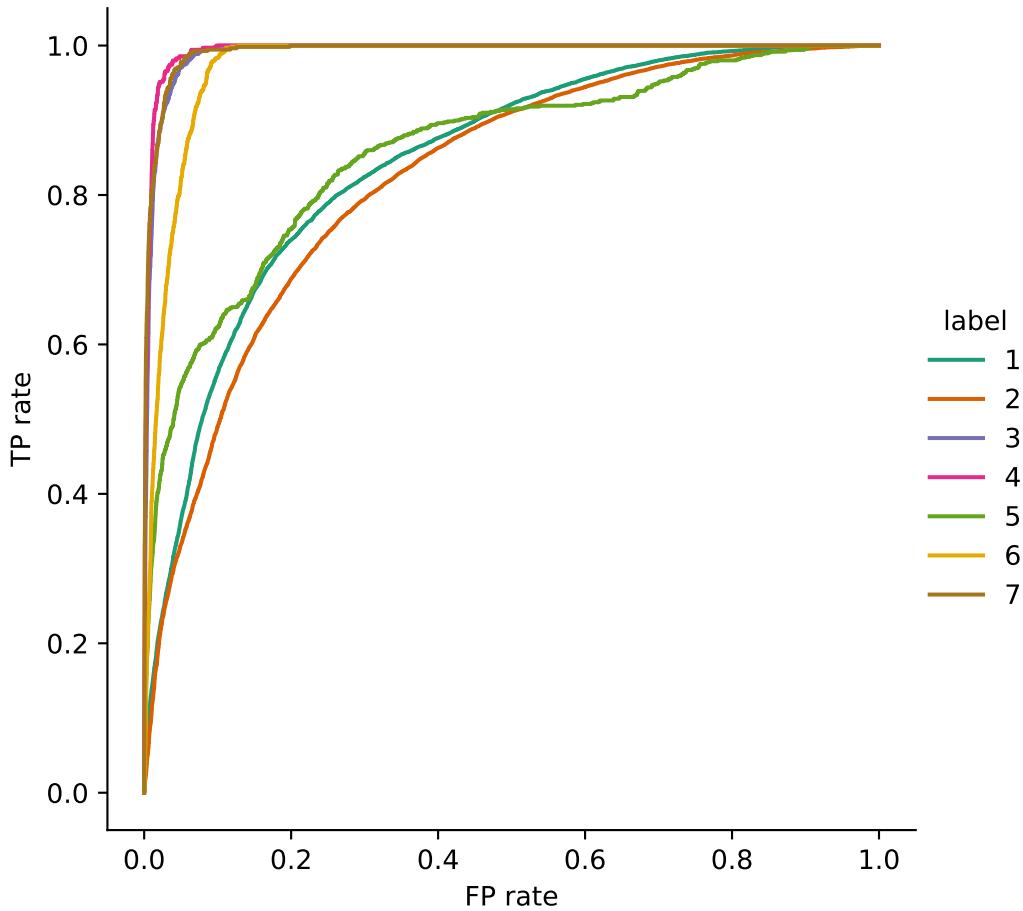
The ROC curves show that the performance is much worse for 3 of the classes than for the others:

```

results = []
for i, label in enumerate(pipe.classes_):
    actual = (y_test==label)
    fp, tp, theta = roc_curve(actual, p_hat[:,i])
    results.extend( [ (label,fp,tp) for fp,tp in zip(fp,tp) ] )

roc = pd.DataFrame( results, columns=["label", "FP rate", "TP rate"] )
sns.relplot(data=roc,
             x="FP rate", y="TP rate",
             hue="label", kind="line", estimator=None,
             palette="Dark2"
            );

```



## Exercises

**Exercise 5.1.** (5.1) Suppose that the distinct plane points  $(x_i, y_i)$  for  $i = 1, \dots, n$  are to be fit using a linear function without intercept,  $\hat{f}(x) = wx$ . Use calculus to find a formula for the value of  $w$  that minimizes the sum of squared residuals,

$$r = \sum_{i=1}^n (y_i - \hat{f}(x_i))^2.$$

**Exercise 5.2.** (5.1) Using the formulas derived in Section 5.1, show that the point  $(\bar{x}, \bar{y})$  always lies on the linear regression line. (Hint: You only have to show that  $\hat{f}(\bar{x}) = \bar{y}$ , which can be done without first solving for  $a$  and  $b$ .)

**Exercise 5.3.** (5.1) Suppose that

$$\begin{aligned}\mathbf{x} &= [-2, 0, 1, 3] \\ \mathbf{y} &= [4, 1, 2, 0].\end{aligned}$$

Find the **(a)** MSE, **(b)** MAE, and **(c)** coefficient of determination on this set for the regression function  $\hat{f}(x) = 1 - x$ .

**Exercise 5.4.** (5.2) Suppose for  $d = 3$  features you have the  $n = 4$  sample vectors

$$\mathbf{x}_1 = [1, 0, 1], \quad \mathbf{x}_2 = [-1, 2, 2], \quad \mathbf{x}_3 = [3, -1, 0], \quad \mathbf{x}_4 = [0, 2, -2],$$

and a multilinear regression computes the weight vector  $\mathbf{w} = [2, 1, -1]$ . Find **(a)** the matrix-vector product  $\mathbf{X}\mathbf{w}$ , and **(b)** the predictions of the regressor on the sample vectors.

**Exercise 5.5.** (5.2) Suppose that values  $y_i$  for  $i = 1, \dots, n$  are to be fit to 2D sample vectors using a multilinear regression function  $\hat{f}(\mathbf{x}) = w_1 x_1 + w_2 x_2$ . Define the sum of squared residuals

$$r = \sum_{i=1}^n (y_i - \hat{f}(\mathbf{x}_i))^2.$$

Show that by holding  $w_1$  constant and taking a derivative with respect to  $w_2$ , and then holding  $w_2$  constant and taking a derivative with respect to  $w_1$ , at the minimum residual we must have

$$\begin{aligned}\left( \sum X_{i,1}^2 \right) w_1 + \left( \sum X_{i,1} X_{i,2} \right) w_2 &= \sum X_{i,1} y_i, \\ \left( \sum X_{i,1} X_{i,2} \right) w_1 + \left( \sum X_{i,2}^2 \right) w_2 &= \sum X_{i,2} y_i,\end{aligned}$$

where  $X_{i,1}$  and  $X_{i,2}$  are the entries in the  $i$ th row of the feature matrix  $\mathbf{X}$ . (In each case above the sum is from  $i = 1$  to  $i = n$ .)

**Exercise 5.6.** (5.3) If we fit the model  $\hat{f}(x) = wx$  to the single data point  $(2, 6)$ , then the ridge loss is

$$r(w) = (2w - 6)^2 + \alpha w^2,$$

where  $\alpha$  is a nonnegative constant. When  $\alpha = 0$ , it's clear that  $w = 3$  is the minimizer of  $r(w)$ . Show that if  $\alpha > 0$ , then  $r'(w)$  is zero at a value of  $w$  in the interval  $(0, 3)$ . (This shows that the weight decreases in the presence of the regularization penalty.)

**Exercise 5.7.** (5.3) If we fit the model  $\hat{f}(x) = wx$  to the single data point  $(2, 6)$ , then the LASSO loss is

$$r(w) = (2w - 6)^2 + \alpha|w|,$$

where  $\alpha$  is a nonnegative constant. When  $\alpha = 0$ , it's clear that  $w = 3$  is the minimizer of  $r(w)$ . Below you will show that the minimizer is less than this if  $\alpha > 0$ .

- (a)** Show that if  $w < 0$ , then  $r'(w)$  cannot be zero. (Remember that for such  $w$ ,  $|w| = -w$ .)
- (b)** Show that if  $w > 0$  and  $0 < \alpha < 24$ , then  $r'(w)$  has a single root in the interval  $(0, 3)$ .

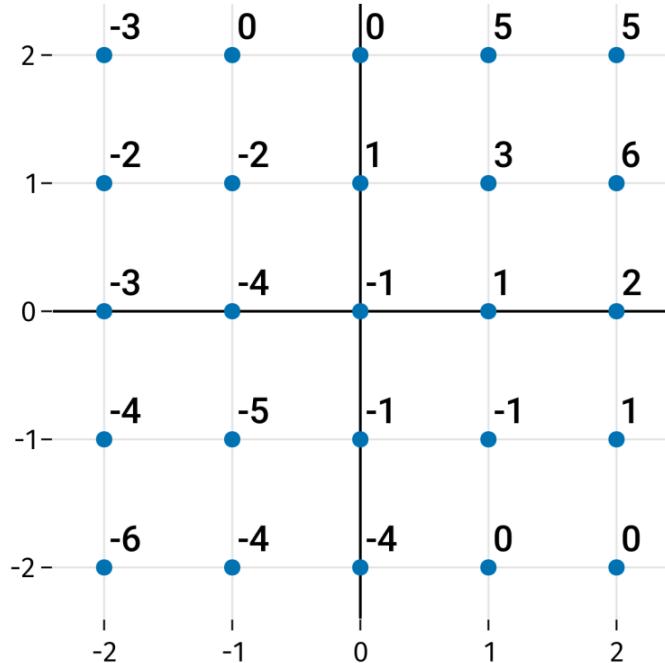
**Exercise 5.8.** (5.4) For each function on two-dimensional vectors, either prove that it is linear or produce a counterexample that shows it cannot be linear.

- (a)  $\hat{f}(\mathbf{x}) = x_1 x_2$
- (b)  $\hat{f}(\mathbf{x}) = x_2$
- (c)  $\hat{f}(\mathbf{x}) = x_1 + x_2 + 1$

**Exercise 5.9.** (5.4) Given the data set  $(x_i, y_i) = \{(0, -1), (1, 1), (2, 3), (3, 0), (4, 3)\}$ , find the MAD-based  $Q$  score for the following hypothetical decision tree splits.

- (a)  $x \leq 0.5$ ,
- (b)  $x \leq 1.5$ ,
- (c)  $x \leq 2.5$ ,
- (d)  $x \leq 3.5$ .

**Exercise 5.10.** (5.4) Here are values (labels) on an integer lattice.



Let  $\hat{f}(x_1, x_2)$  be the kNN regressor using  $k = 4$ , Euclidean metric, and mean averaging. In each case below, a function  $g(t)$  is defined from values of  $\hat{f}$  along a vertical or horizontal line. Carefully sketch a plot of  $g(t)$  for  $-2 \leq t \leq 2$ .

- (a)  $g(t) = \hat{f}(1.2, t)$
- (b)  $g(t) = \hat{f}(t, -0.75)$
- (c)  $g(t) = \hat{f}(t, 1.6)$
- (d)  $g(t) = \hat{f}(-0.25, t)$

**Exercise 5.11.** (5.5) Here are some label values and probabilistic predictions by a logistic regressor:

$$\mathbf{y} = [0, 0, 1, 1], \\ \hat{\mathbf{p}} = [\frac{3}{4}, 0, 1, \frac{1}{2}].$$

Using base-2 logarithms, calculate the cross-entropy loss for these predictions.

**Exercise 5.12.** (5.5) Let  $\mathbf{x} = [-1, 0, 1]$  and  $\mathbf{y} = [0, 1, 0]$ . This small dataset is fit to a probabilistic predictor  $\hat{p}(x) = \sigma(wx)$  for weight  $w$ .

(a) Let  $L(w)$  be the cross-entropy loss function using natural logarithms. Show that

$$L'(w) = \frac{e^w - 1}{e^w + 1}.$$

- (b) Explain why part (a) implies that  $w = 0$  is the global minimizer of the loss  $L$ .  
(c) Using the result of part (b), simplify the optimum predictor function  $\hat{p}(x)$ .

**Exercise 5.13.** (5.5) Let  $\mathbf{x} = [-1, 1]$  and  $\mathbf{y} = [0, 1]$ . This small dataset is fit to a probabilistic predictor  $\hat{p}(x) = \sigma(wx)$  for weight  $w$ . Without regularization, the best fit takes  $w \rightarrow \infty$ , which makes the predictor become infinitely steep at  $x = 0$ . To combat this behavior, let  $L$  be the cross-entropy loss function with LASSO penalty, i.e.,

$$L(w) = \ln[1 - \hat{p}(-1)] - \ln[\hat{p}(1)] + \alpha|w|,$$

for a positive regularization constant  $\alpha$ .

- (a) Show that  $L'$  is never zero for  $w < 0$ .  
(b) Show that if  $0 < \alpha < 1$ , then  $L'$  has a zero at

$$w = \ln\left(\frac{2}{\alpha} - 1\right).$$

- (c) Show that  $w$  from part (b) is a decreasing function of  $\alpha$ . (Therefore, increasing  $\alpha$  makes the predictor less steep as a function of  $x$ .)

# 6 Clustering

```
import numpy as np
from numpy.random import default_rng
import pandas as pd
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix, f1_score, balanced_accuracy_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_curve
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.model_selection import cross_validate, validation_curve
from sklearn.model_selection import GridSearchCV
```

In supervised learning, the data samples are supplied with labels, and the goal of the learner is to generalize the examples to new values. In unsupervised learning, there are no labels. Instead, the goal is to discover structure that is intrinsic to the feature matrix. Common problem types in unsupervised learning are

**Clustering** Determine whether the samples roughly divide into a small number of classes.

**Dimension reduction** Find a reduced set of features, or create a small set of new features, that describe the data well.

**Outlier detection** Find anomalous values in the data set and remove them or impute replacements.

In this chapter we will look at clustering. The goal is to assign each feature vector a number from 1 to  $k$ , where  $k$  is much smaller than the number of samples. More formally:

**Definition 6.1.** Given an  $n \times d$  feature matrix with rows  $\mathbf{X}_1, \dots, \mathbf{X}_n$ , a **clustering** is a labelling function  $c$  defined on the feature vectors such that

$$c(\mathbf{X}_i) \in \{1, 2, \dots, k\} \text{ for all } i = 1, \dots, n,$$

[https://www.  
dropbox.com/  
s/  
9qhqczo8ugcoobq/  
Section6\\_0.  
mp4?raw=1](https://www.dropbox.com/s/9qhqczo8ugcoobq/Section6_0.mp4?raw=1)

where  $k$  is a positive integer. The **cluster**  $C_j$  is the collection of all  $\mathbf{X}_i$  such that  $c(\mathbf{X}_i) = j$ .

The clusters  $C_1, \dots, C_k$  divide the samples into  $k$  disjoint subsets. Depending on the algorithm, the number of clusters  $k$  may be imposed (i. e., as a hyperparameter) or determined automatically.

Intuitively, we want similar examples to be clustered together (that is, to receive the same label). The motivation for clustering is often to find a way to classify the samples by intrinsic properties when no such classification is known in advance. It is *not* necessary that the clustering function  $c$  be defined for vectors  $\mathbf{x}$  other than the feature vectors, although many clustering algorithms can be used to do that as well.

### ! Important

The universe doesn't owe you a clustering. Not all phenomena are amenable to clustering in whatever features you happen to choose.

While classification only requires us to separate different classes of examples, clustering is more specific and more demanding: samples in a cluster need to be more like each other, or the “average” cluster member, than they are like members of other clusters. We should expect that edge cases will look ambiguous.

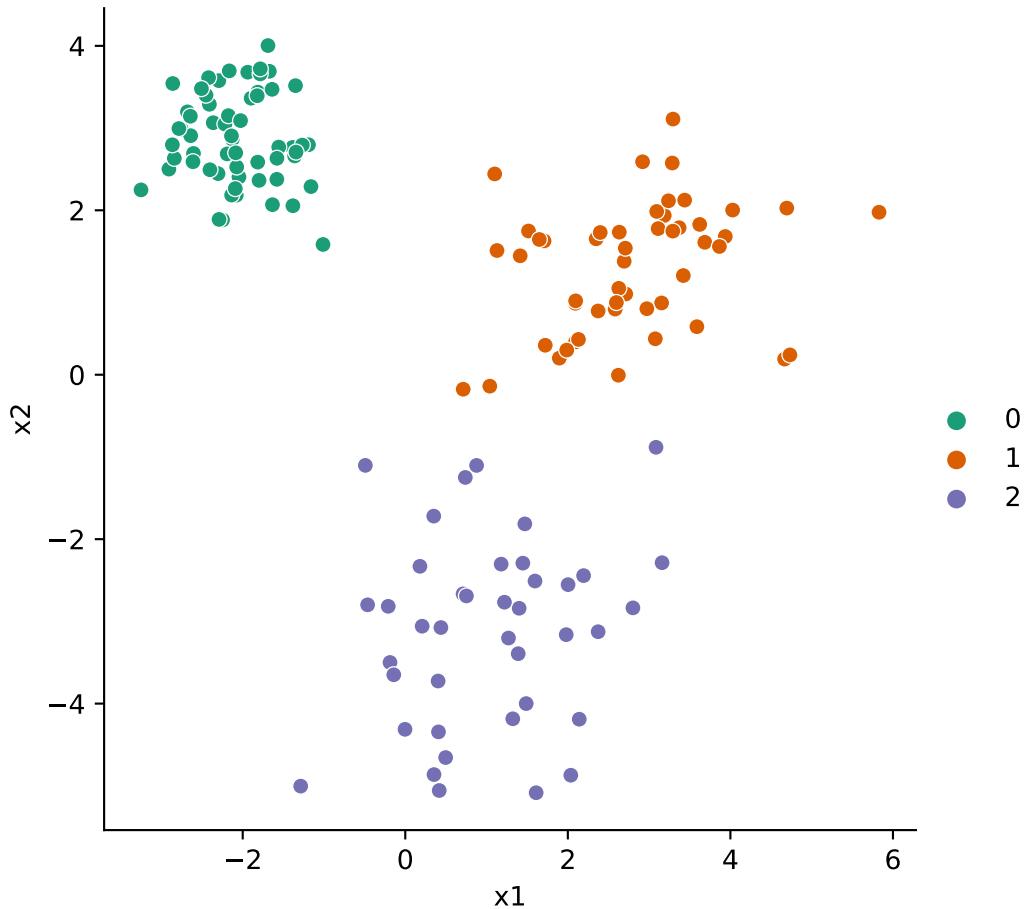
In order to get a feeling for the algorithms, we will apply them to three illustrative datasets:

- **blobs** This dataset has one distinct blob, plus two that kind of overlap a bit:

```
from sklearn.datasets import make_blobs
def blobs_data():
    X, y = make_blobs(
        n_samples=[60, 50, 40],
        centers=[[2,3], [3,1.5], [1,-3]],
        cluster_std=[0.5, 0.9, 1.2],
        random_state = 19716
    )

    return pd.DataFrame({"x1": X[:,0], "x2": X[:,1]}), pd.Series(y)

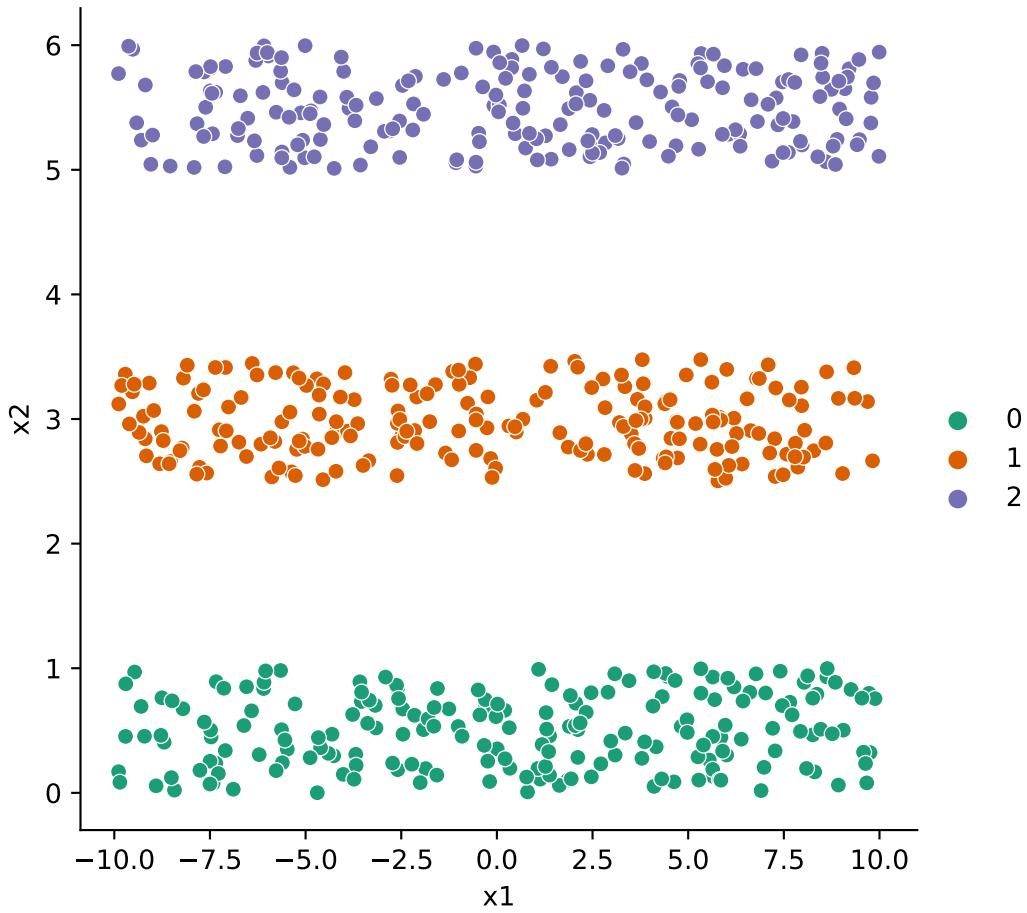
X, y = blobs_data()
sns.relplot(data=X, x="x1", y="x2", hue=y, palette="Dark2");
```



- **stripes** In this dataset, there is clear separation along one axis and a continuous blob along the other:

```
def stripes_data():
    rng = default_rng(9)
    x1,x2,cls = [],[],[]
    for i in range(3):
        x1.extend( rng.uniform(-10, 10, size=200) )
        x2.extend( 2.5*i+rng.uniform(0, 1, size=200) )
        cls.extend( [i]*200)
    return pd.DataFrame({"x1": x1, "x2": x2}), pd.Series(cls)

X, y = stripes_data()
sns.relplot(data=X, x="x1", y="x2", hue=y, palette="Dark2");
```



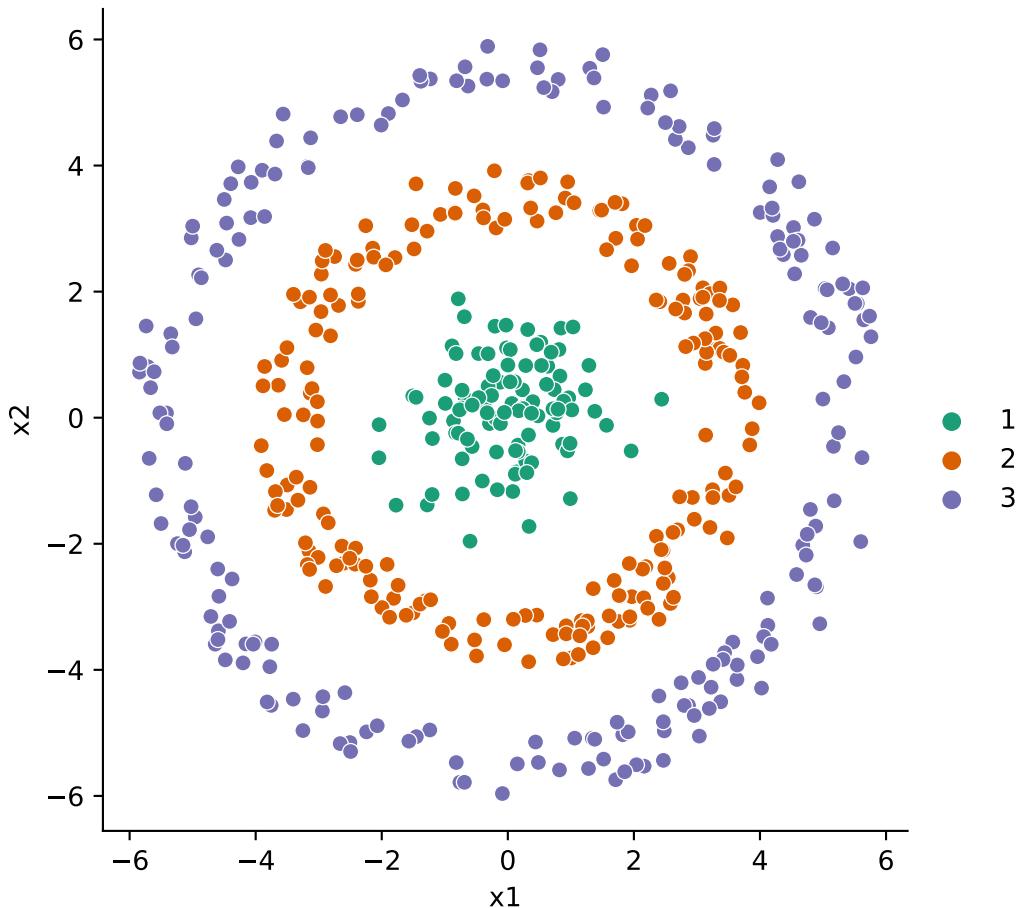
- **bullseye** This is the most challenging dataset, because the clusters can't be separated by anything like straight lines:

```
def bullseye_data():
    rng = default_rng(6)
    inner = 0.8*rng.normal(size=(100, 2))
    theta = rng.uniform(0, 2*np.pi, size=200)
    r = rng.uniform(3, 4, size=200)
    middle = np.vstack((r*np.cos(theta), r*np.sin(theta))).T
    r = rng.uniform(5, 6, size=200)
    outer = np.vstack((r*np.cos(theta), r*np.sin(theta))).T
    cls = np.hstack( ([1]*100, [2]*200, [3]*200))
    X = pd.DataFrame( np.vstack((inner,middle,outer)), columns=["x1", "x2"] )
    return X, pd.Series(cls)
```

```

x, y = bullseye_data()
p = sns.relplot(data=X, x="x1", y="x2", hue=y, palette="Dark2")
p.set(aspect=1);

```



## 6.1 Similarity

Ideally, samples within a cluster are more similar to each other than they are to samples in other clusters. The first decision we have to make is how to define *similarity* between an arbitrary pair.

When a distance metric is available, we intuitively expect similarity to be inversely related to distance. There are various ways to quantify the relationship, but we will not use them. Instead, we will take “maximize similarity” to be equivalent to “minimize distance.”

### 6.1.1 Distance matrix

**Definition 6.2.** Given the feature vectors  $\mathbf{X}_1, \dots, \mathbf{X}_n$ , the pairwise distances between them are collected in the  $n \times n$  **distance matrix**

$$D_{ij} = \text{dist}(\mathbf{X}_i, \mathbf{X}_j).$$

Note that  $D_{ii} = 0$  and  $D_{ji} = D_{ij}$ .

**Example 6.1.** Using 1-norm for distance, find the distance matrix for the feature matrix

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 1 & -2 \\ 0 & 3 & 3 & 1 \\ 1 & -1 & 0 & 4 \end{bmatrix}.$$

[https://www.  
dropbox.com/  
s/  
lp98e4bn5drn5r8/  
Section6\\_1\\_  
1.mp4?raw=1](https://www.dropbox.com/s/lp98e4bn5drn5r8/Section6_1_1.mp4?raw=1)

*Solution.* There are  $n = 3$  feature vectors.

$$D_{12} = |1| + |-1| + |-2| + |-3| = 7$$

$$D_{13} = |0| + |3| + |1| + |-6| = 10$$

$$D_{23} = |-1| + |4| + |3| + |-3| = 11.$$

Hence,

$$\mathbf{D} = \begin{bmatrix} 0 & 7 & 10 \\ 7 & 0 & 11 \\ 10 & 11 & 0 \end{bmatrix}.$$

#### Tip

Many clustering algorithms allow supplying  $\mathbf{D}$  in lieu of the feature vectors. When you consider that in many problems,  $n$  is much larger than  $d$ , it can be faster and easier to work with.

#### Note

A practical advantage of working with similarity rather than distance is that small values of similarity can be rounded down to zero. Such rounding has negligible effect on the results, but it can create big gains in execution time and memory usage.

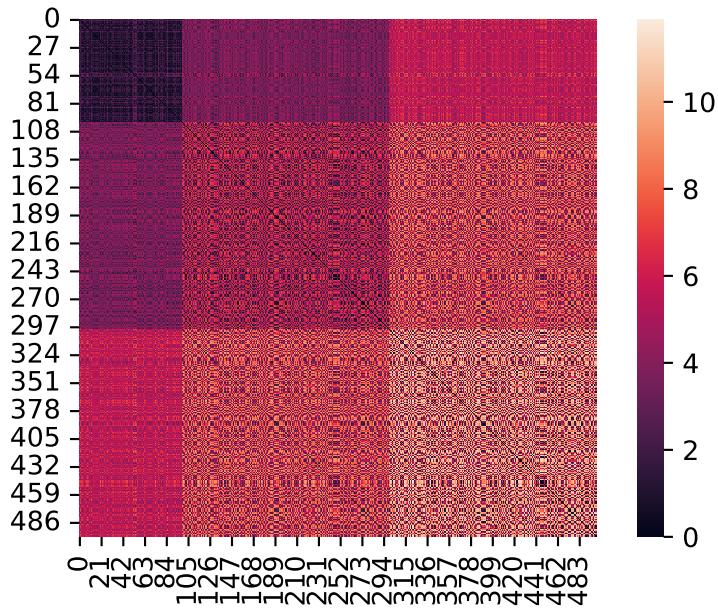
The scikit-learn function `pairwise_distances` computes a distance matrix efficiently.

**Example 6.2.** The distance matrix of our *bullseye* dataset has some interesting structure:

```

from sklearn.metrics import pairwise_distances
X, y = bullseye_data()
D2 = pairwise_distances(X, metric="euclidean")    # use 2-norm metric
ax = sns.heatmap(D2)
ax.set_aspect(1);

```



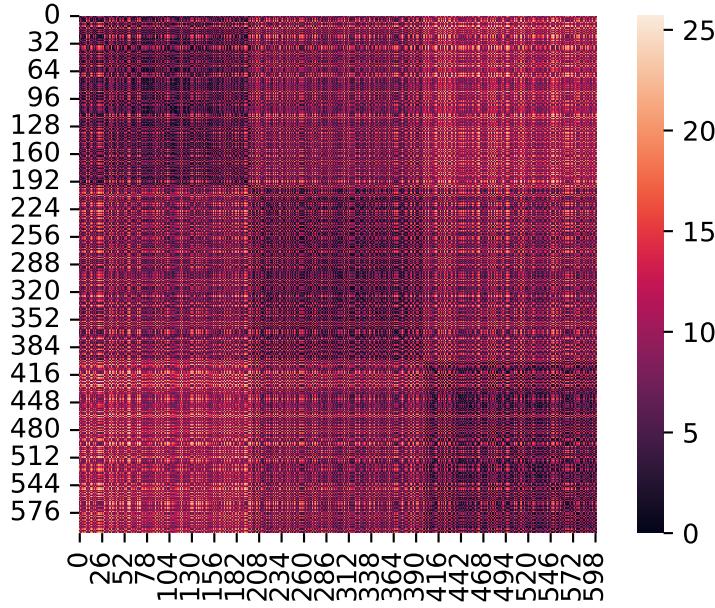
Because we set up three geometrically distinct groups of points, the distances of pairs within and between groups are fairly homogeneous. The lower-right corner, for example, shows that points in the outermost ring tend to be separated by the greatest distance.

In the 1-norm, the *stripes* dataset is also interesting:

```

X, y = stripes_data()
D1 = pairwise_distances(X, metric="manhattan")    # use 1-norm metric
ax = sns.heatmap(D1)
ax.set_aspect(1);

```



Points in different stripes are always separated by at least the inter-stripe distance, while points within the same stripe have a range of possible distances.

### 6.1.2 Angular distance

An alternative to vector norms for distances is **angular distance**. In three dimensions, imagine drawing a ray from the origin to each of the two given vectors. Those rays lie in a common plane, and we can let  $\theta$  be the (smallest) angle between them. The general formula in any number of dimensions is

$$\cos(\theta) = \frac{\mathbf{u}^T \mathbf{v}}{\|\mathbf{u}\|_2 \|\mathbf{v}\|_2}. \quad (6.1)$$

We then let  $\text{dist}(\mathbf{x}, \mathbf{y}) = \theta \in [0, \pi]$ , using the standard branch of  $\arccos$ .

However,  $\arccos$  is a relatively expensive computational operation, and sometimes we use the pseudodistance

$$d(\mathbf{x}, \mathbf{y}) = 1 - \cos(\theta)$$

in place of  $\theta$ . This function (which ranges between 0 and 2) is not a true distance function, though, as  $d$  fails to satisfy the triangle inequality in all cases (see Exercise 6.2).

Angular distance and cosine pseudodistance are useful when we want to ignore the magnitudes of vectors and consider only their directions.

[https://www.  
dropbox.com/  
s/  
7lfw96r67olltnu/  
Section6\\_1\\_  
2.mp4?raw=1](https://www.dropbox.com/s/7lfw96r67olltnu/Section6_1_2.mp4?raw=1)

**Example 6.3.** We might represent a text document by a vector of the number of occurrences of certain keywords. In sentiment analysis, these would be terms such as *happy*, *excited*, *sad*, *angry*, *confused*, and so on. For sake of argument, suppose three documents have the feature matrix

$$\mathbf{X} = \begin{bmatrix} 6 & 1 & 10 & 2 & 5 \\ 14 & 0 & 23 & 3 & 7 \\ 2 & 3 & 1 & 5 & 0 \end{bmatrix}.$$

Using the 2-norm, we get the distances:

```
X = np.array( [[6,1,10,2,5], [14,0,23,3,7], [2,3,1,5,0]] )
pairwise_distances( X, metric="euclidean")

array([[ 0.          , 15.45962483, 11.61895004],
       [15.45962483,  0.          , 26.26785107],
       [11.61895004, 26.26785107,  0.          ]])
```

According to this metric, the most similar documents are the first and the last. However, cosine distance tells a different story:

```
pairwise_distances( X, metric="cosine")

array([[0.          , 0.01532383, 0.56500757],
       [0.01532383, 0.          , 0.62231412],
       [0.56500757, 0.62231412, 0.          ]])
```

Now the first two documents look most similar. Looking at the vectors, we see that these two have the most similar relative frequencies, which is related to what we want to measure. The apparent dissimilarity in the 2-norm arises solely because the total word counts are very different.

### 6.1.3 Distance in high dimensions

High-dimensional space [does not conform to some intuitions](#) formed by our experiences in 2D and 3D.

For example, consider the unit hyperball  $\|\mathbf{x}\|_2 \leq 1$  in  $d$  dimensions. We'll take it as given that scaling a  $d$ -dimensional object by a number  $r$  will scale the volume by  $r^d$ . Then for any  $r < 1$ , the fraction of the unit hyperball's volume lying *outside* the smaller hyperball of fixed radius  $r$  is  $1 - r^d$ , which approaches 1 as  $d \rightarrow \infty$ . That is, *if we choose points randomly within a hyperball, almost all of them will be near the outer boundary*.

The volume of the unit hyperball also vanishes as  $d \rightarrow \infty$ . This is because the inequality

[https://www.dropbox.com/s/hxdmolatua15skd/Section6\\_1\\_3.mp4?raw=1](https://www.dropbox.com/s/hxdmolatua15skd/Section6_1_3.mp4?raw=1)

$$x_1^2 + x_2^2 + \cdots + x_d^2 \leq 1,$$

where each  $x_i$  is chosen randomly in  $[-1, 1]$ , becomes ever harder to satisfy as the number of terms in the sum grows, and the relative occurrence of such points is increasingly rare.

There are other, similar mathematical results demonstrating the weirdness of distances in high-dimensional space. These go under the colorful name *curse of dimensionality*, and the advice given in response to them is sometimes stated flatly as, “Don’t use distance metrics in high-dimensional space.”

But that form of the advice may be overstated. The curse is essentially about *randomly* chosen points, and it is correct that dimensions of noisy or irrelevant features will make most learning algorithms less effective. But when features carry useful information, adding them usually makes matters better, not worse.

## 6.2 Performance measures

Before we start generating clusterings, we will discuss how to evaluate them. A clustering is essentially a partitioning of the samples into disjoint subsets. We will use some nonstandard terminology that makes the definitions a bit easier to state and read.

**Definition 6.3.** We say that two sample points in a clustering are **buddies** if they are in the same cluster, and **strangers** otherwise.

### 6.2.1 Rand index and ARI

A clustering can be interpreted as a classification, and vice versa. If a reference classification is available, then we can compare any clustering result to it. This allows us to use classification datasets as proving grounds for clustering.

Let  $b$  be the number of pairs that are buddies in both clusterings, and let  $s$  be the number of pairs that are strangers in both clusterings. We define the **Rand index** by

$$\text{RI} = \frac{b + s}{\binom{n}{2}}.$$

The reason for the denominator is that there are  $\binom{n}{2}$  distinct pairs of  $n$  sample points, so we know that  $0 \leq \text{RI} \leq 1$ .

One way to interpret the Rand index is through binary classification. If we define a positive result on a pair of samples to mean “in the same cluster” and a negative result to mean “in different clusters”, then the Rand index is the accuracy of that classifier over all pairs of samples, taking the reference clustering as providing ground truth.

[https://www.  
dropbox.com/  
s/  
gd2nfz5oxgqduf/  
Section6\\_2  
1.mp4?raw=1](https://www.dropbox.com/s/gd2nfz5oxgqduf/Section6_2.mp4?raw=1)

**Example 6.4.** Suppose that samples  $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_4$  are classified as blue, and  $\mathbf{X}_3, \mathbf{X}_5$  are classified as red. Compute the Rand index relative to the reference classification for the clustering  $C_1 = \{\mathbf{X}_1, \mathbf{X}_2\}$  and  $C_2 = \{\mathbf{X}_3, \mathbf{X}_4, \mathbf{X}_5\}$ . (Note that the actual feature vectors themselves are not important in this setting.)

*Solution.* Here is a table showing the buddy–stranger status for every pair in the coloring and clustering:

	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{X}_4$	$\mathbf{X}_5$
$\mathbf{X}_1$		B/B	S/S	B/S	S/S
$\mathbf{X}_2$			S/S	B/S	S/S
$\mathbf{X}_3$				S/B	B/B
$\mathbf{X}_4$					S/B
$\mathbf{X}_5$					

Hence, the Rand index is  $(2+4)/10 = 0.6$ .

The Rand index has some attractive features:

- The value is between 0 (complete disagreement) and 1 (complete agreement).
- It is symmetric in the two clusterings; it doesn't matter which is considered the reference.
- There is no need to find a correspondence between the clusters in the two clusterings. In fact, the clusterings need not even have the same number of clusters.

A weakness of the Rand index is that it can be fairly close to 1 even for a random clustering. The **adjusted Rand index** rescales the result by comparing it to how a random clustering would fare. It can be written as

$$\text{ARI} = \frac{\text{RI} - \mathbb{E}[\text{RI}]}{\max(\text{RI}) - \mathbb{E}[\text{RI}]},$$

where the expectation and max operations are performed over all possible clusterings. (These values can be worked out exactly using combinatorics, but we do not give them here.)

An ARI of 0 indicates no better agreement than a random clustering, while an ARI of 1 is complete agreement. Unlike the RI, the ARI value can be negative, indicating a performance worse than on average.

**Example 6.5.** Our *blobs* dataset has a reference clustering that we take to be the gold standard. Let's create another clustering based entirely on the quadrants of the plane:

```

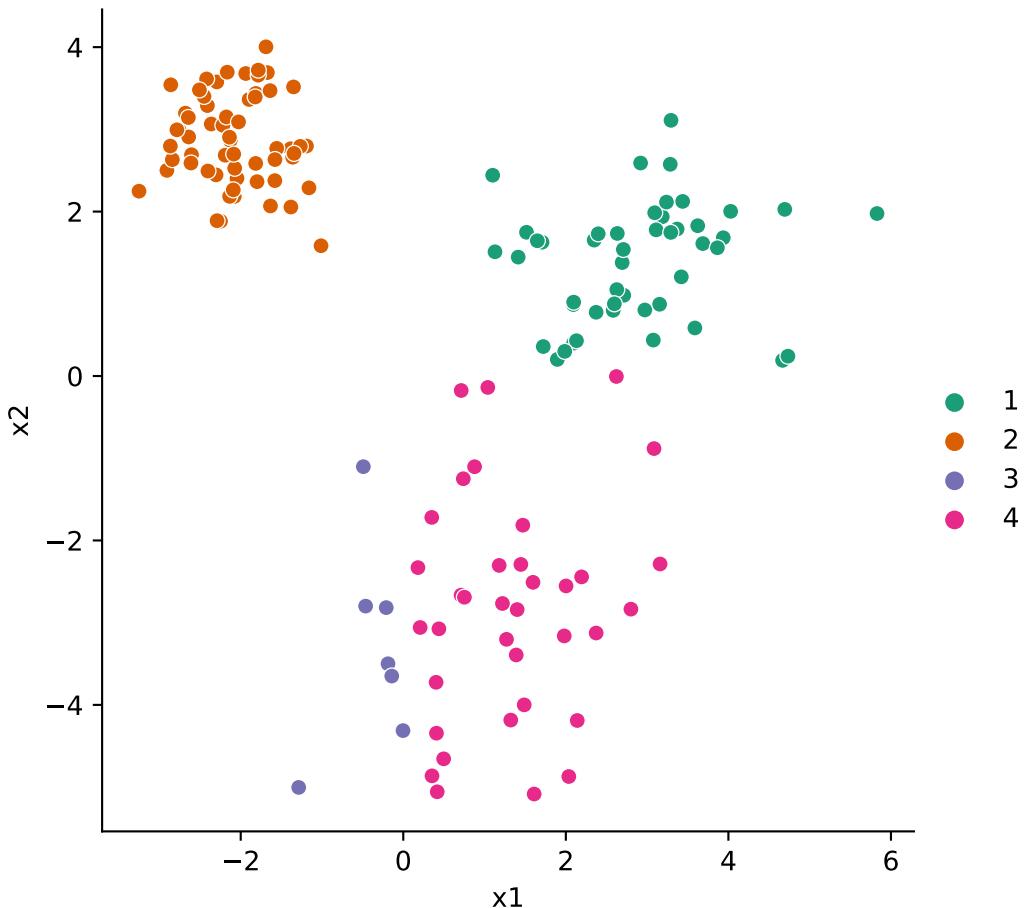
X, y = blobs_data()

def quad(x1, x2):
    if x1 > 0:
        if x2 > 0: return 1
        else: return 4
    else:
        if x2 > 0: return 2
        else: return 3

q = pd.Series( [ quad(x1, x2) for (x1, x2) in zip(X["x1"], X["x2"]) ] )

sns.relplot(data=X, x="x1", y="x2", hue=q, palette="Dark2");

```



The reference clustering has three classes, and there are four quadrants. Yet we can still

compare them by adjusted Rand index:

```
from sklearn.metrics import adjusted_rand_score  
  
adjusted_rand_score(y, q)
```

0.904092765401111

Not surprisingly, they are seen as fairly similar.

[https://www.  
dropbox.com/  
s/  
fj759qpo8j38wei/  
Example6\\_5.  
mp4?raw=1](https://www.dropbox.com/s/fj759qpo8j38wei/Example6_5.mp4?raw=1)

### 6.2.2 Silhouettes

If no reference clustering/classification is available, then the only way we can assess the quality of a clustering is to check how well it creates clusters whose members are more similar to their buddies than to strangers.

**Definition 6.4.** Suppose  $\mathbf{X}_i$  is a sample in a clustering. Let  $\bar{b}_i$  be the mean distance between  $\mathbf{X}_i$  and its buddies, and let  $\bar{r}_i$  be the mean distance between  $\mathbf{X}_i$  and the members of the nearest cluster of strangers. Then the **silhouette value** of  $\mathbf{X}_i$  is

$$s_i = \frac{\bar{r}_i - \bar{b}_i}{\max\{\bar{r}_i, \bar{b}_i\}}.$$

The value  $s_i$  is always in the interval  $[-1, 1]$ . A negative value indicates that the sample seems to be more similar to a cluster other than its own.

[https://www.  
dropbox.com/  
s/  
yxjf804d2ugayp7/  
Section6\\_2\\_  
2.mp4?raw=1](https://www.dropbox.com/s/yxjf804d2ugayp7/Section6_2_2.mp4?raw=1)

#### ! Important

Most sources, and scikit-learn, define an overall *silhouette score* as the mean of the  $s_i$ . However, because the distributions of these values are often asymmetric and have significant outliers, I am taking the view that medians are preferred when a summary score is needed.

**Example 6.6.** Suppose that two clusters in one dimension are defined as  $A = \{-4, -1, 1\}$  and  $B = \{2, 6\}$ . Find the silhouette values of all the samples.

$X_i$	$\bar{b}_i$	$\bar{r}_i$	$s_i$
$X_i$	$\bar{b}_i$	$\bar{r}_i$	$s_i$
-4	$\frac{ -4-1 + -4+1 }{2}$	$\frac{ -4-2 + -4-6 }{2}$	$\frac{8-4}{8} = \frac{1}{2}$
-1	$\frac{ -1+4 + -1-1 }{2}$	$\frac{ -1-2 + -1-6 }{2}$	$\frac{5-2.5}{5} = \frac{1}{2}$
1	$\frac{ 1+4 + 1+1 }{2}$	$\frac{ 1-2 + 1-6 }{2}$	$\frac{3-3.5}{3.5} = -\frac{1}{7}$
2	$\frac{ 2-6 }{1}$	$\frac{ 2+6 + 2+1 + 2-1 }{3}$	$\frac{(10/3)-4}{4} = -\frac{1}{6}$
6	$\frac{ 6-2 }{1}$	$\frac{ 6+4 + 6+1 + 6-1 }{3}$	$\frac{(22/3)-4}{(22/3)} = \frac{5}{11}$

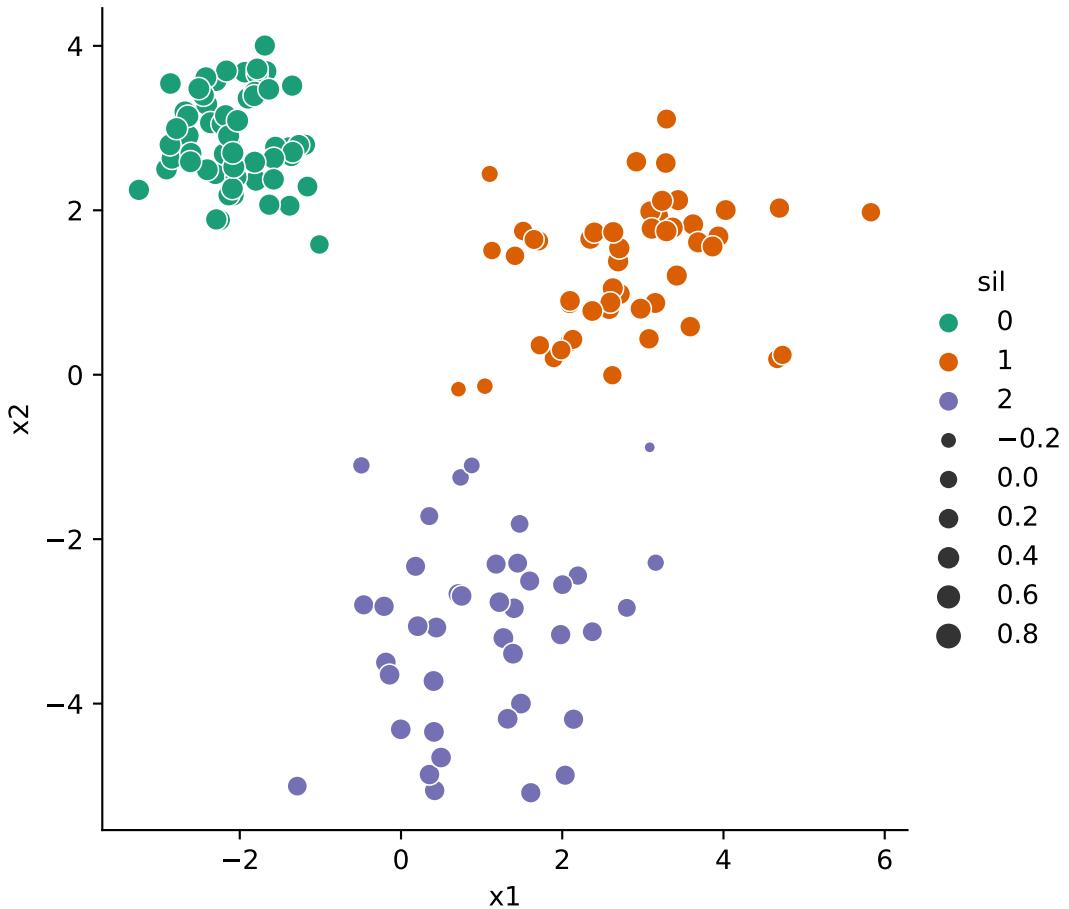
*Solution.*

**Example 6.7.** Let's use the predefined cluster assignments in our blobs dataset. We will add a column to the data frame that records the silhouette score for each point:

```
from sklearn.metrics import silhouette_samples

X, y = blobs_data()

X["sil"] = silhouette_samples(X, y)
sns.relplot(data=X,
    x="x1", y="x2",
    hue=y, size="sil", palette="Dark2"
);
```



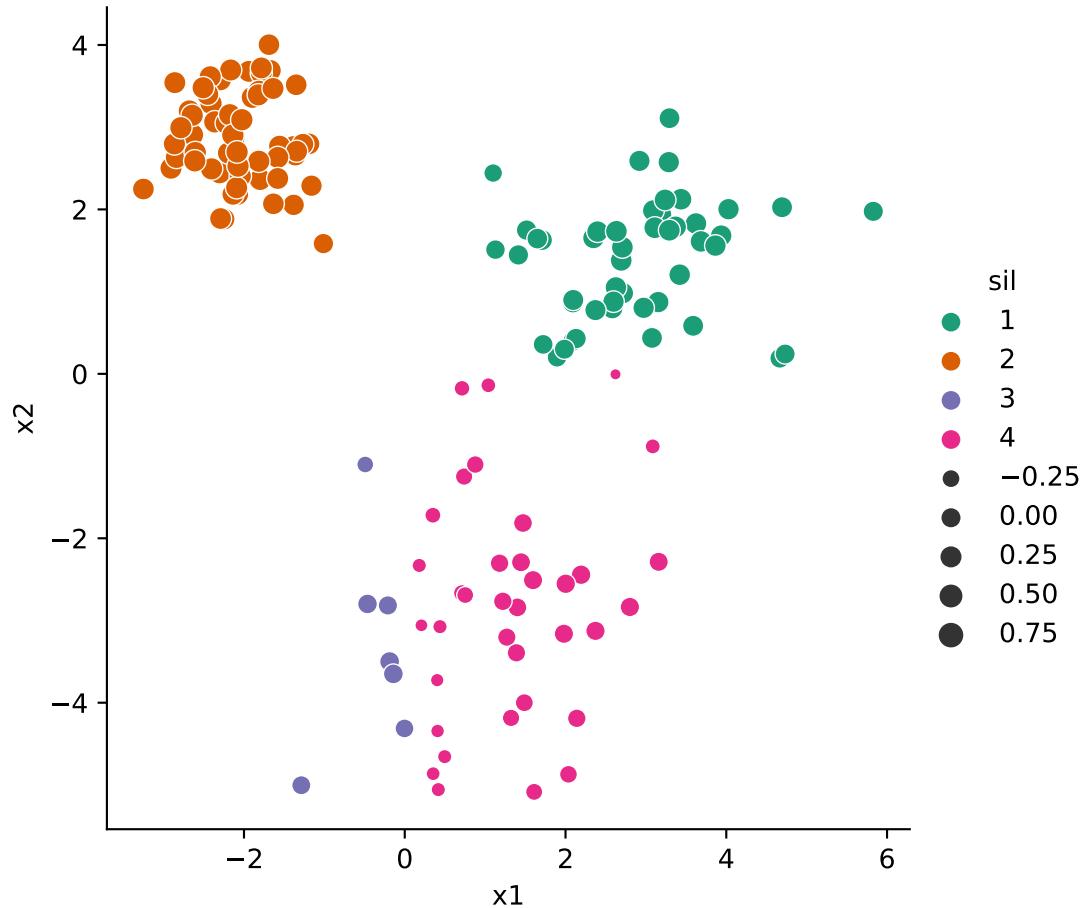
In the plot above, the size of each dot shows its silhouette coefficient. Those points which don't belong comfortably with their cluster have negative scores and the smallest dots. We can find the average score in each cluster through a grouped mean:

```
X.groupby(y)[["sil"]].median()
```

	sil
0	0.827518
1	0.692668
2	0.639722

These values are ordered as we would expect. We can also compute the silhouette scores for the quadrant-based clustering defined in Example 6.5:

```
X["sil"] = silhouette_samples(X, q)
sns.relplot(data=X,
    x="x1", y="x2",
    hue=q, size="sil", palette="Dark2"
);
```



```
X.groupby(q)["sil"].median()
```

	sil
1	0.700787
2	0.828918
3	0.390881
4	0.153479

The scores for clusters labelled as 1 and 2 are pretty good and almost the same as for the

original reference clustering. But the other two clusters score much more poorly separately than they did when they were considered a single cluster.

**Example 6.8.** `sklearn` has a well-known dataset that contains labeled handwritten digits. Let's extract the examples for just the numerals 4, 5, and 6:

[https://www.  
dropbox.com/  
s/  
icu3cmmi38cuzz7/  
Example6\\_7.  
mp4?raw=1](https://www.dropbox.com/s/icu3cmmi38cuzz7/Example6_7.mp4?raw=1)

```
digits = datasets.load_digits(as_frame=True) ["frame"]
keep = digits["target"].isin([4, 5, 6])
digits = digits[keep]
X = digits.drop("target", axis=1)
y = digits.target
y.value_counts()
```

target
5 182
4 181
6 181

We can check the silhouette scores for the reference labelling in order to set expectations for how well a clustering method might do:

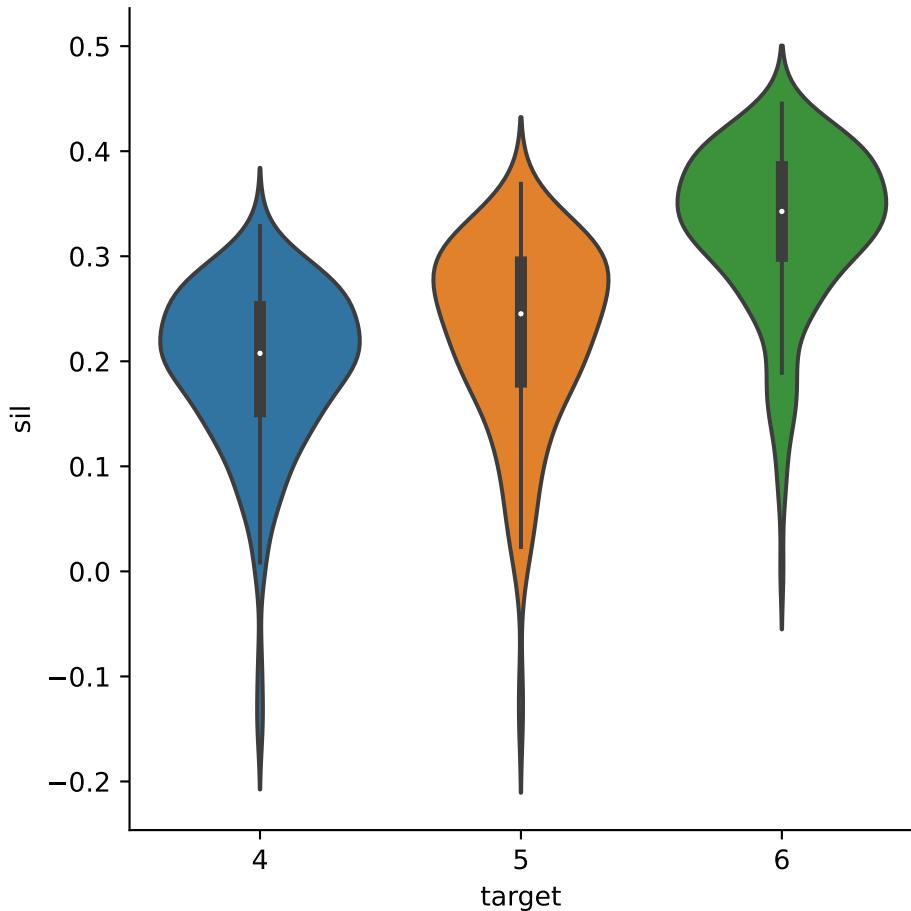
```
digits["sil"] = silhouette_samples(X,y)
digits.groupby("target")["sil"].median()
```

target
4 0.207595
5 0.245201
6 0.342689

As usual, means can tell us only so much. A look at the distributions of the values reveals more details:

```
sns.catplot(data=digits,
             x="target", y="sil",
             kind="violin");

```



The values are mostly positive, which indicates nearly all of the samples for a digit are at least somewhat closer to each other than to the other samples. The 6s are the most distinct. However, the existence of scores close to and below zero suggest that a clustering algorithm is unlikely to reproduce the true classification perfectly. To put it another way, the features chosen to represent the digits don't seem to create three clear, well-separated balls of points representing the three different types of digits.

Silhouette values are fairly easy to use. However, while we might consider a cluster to comprise points that are close only to other members of the cluster, silhouette values demand that the members of a cluster all be mutually close to one another. As a result, they favor compact, roughly spherical clusters.

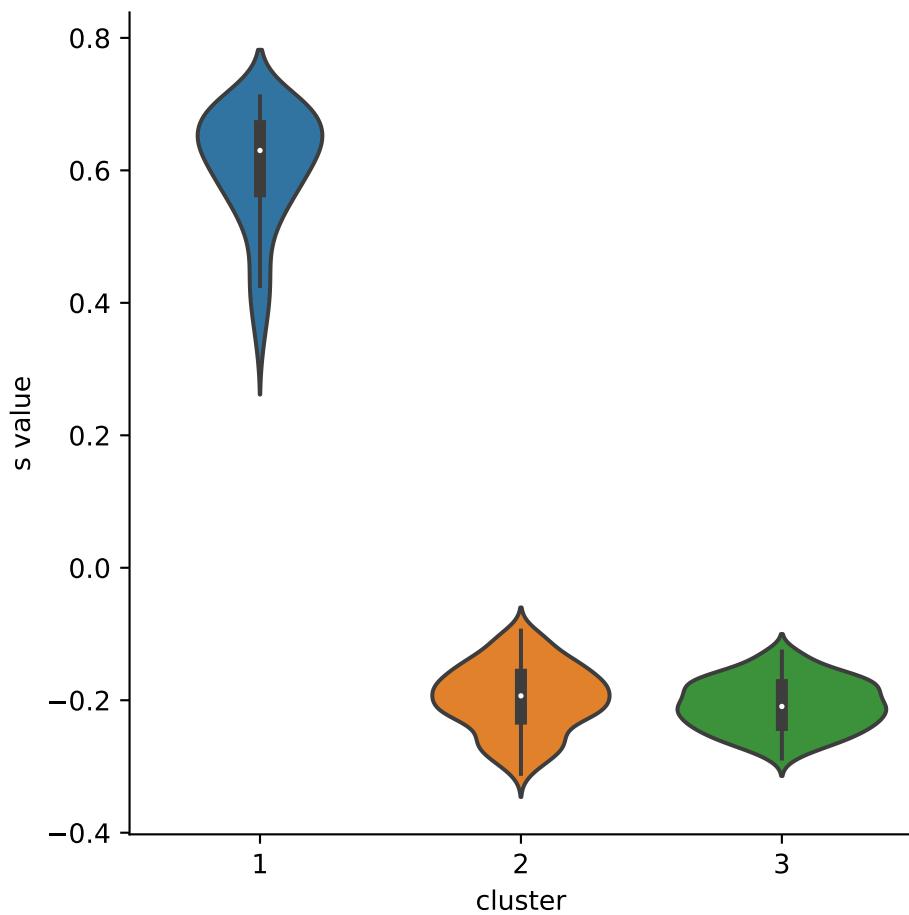
**Example 6.9.** The bullseye dataset has only one “cluster” that silhouette values approve of:

[https://www.  
dropbox.com/  
s/  
sn7gaapl6gc7sea/  
Example6\\_8.  
mp4?raw=1](https://www.dropbox.com/s/sn7gaapl6gc7sea/Example6_8.mp4?raw=1)

```

X, y = bullseye_data()
s = silhouette_samples(X, y)
clusters = pd.DataFrame({"cluster": y, "s value": s})
sns.catplot(data=clusters,
    x="cluster", y="s value",
    kind="violin"
)

```



The central blob is considered OK, but the rings score poorly.

[https://www.  
dropbox.com/  
s/  
f171zxlpriccdvq/  
Example6\\_9.  
mp4?raw=1](https://www.dropbox.com/s/f171zxlpriccdvq/Example6_9.mp4?raw=1)

## 6.3 k-means

The  **$k$ -means algorithm** is one of the best-known and most widely used clustering methods, although it has some significant limitations.

Given a sample matrix  $\mathbf{X}$  with  $n$  rows  $\mathbf{X}_i$ , the algorithm divides the sample points into disjoint sets  $C_1, \dots, C_k$ , where  $k$  is a hyperparameter. We need a pair of key definitions.

**Definition 6.5.** The **centroid** of cluster  $C_j$ , denoted by  $\mu_j$ , is the mean of the vectors in  $C_j$ :

$$\mu_j = \frac{1}{|C_j|} \sum_{\mathbf{x} \in C_j} \mathbf{x}.$$

The **inertia** of  $C_j$  is

$$I_j = \sum_{\mathbf{x} \in C_j} \|\mathbf{x} - \mu_j\|_2^2. \quad (6.2)$$

The goal of the  $k$ -means algorithm is to choose the clusters in order to minimize the total inertia,

$$I = \sum_{j=1}^k I_j.$$

[https://www.  
dropbox.com/  
s/  
9ay1njua14zlibj/  
Section6\\_3.  
mp4?raw=1](https://www.dropbox.com/s/9ay1njua14zlibj/Section6_3.mp4?raw=1)

**Example 6.10.** Given the samples  $-3, -2, -1, 2, 5, 7$ , find the inertia of the clustering

$$C_1 = \{-3, -2, -1\}, \quad C_2 = \{2, 5, 7\},$$

and of the clustering

$$C_1 = \{-3, -2, -1, 2\}, \quad C_2 = \{5, 7\}.$$

*Solution.* The first clustering has centroids  $\mu_1 = -2$ ,  $\mu_2 = 14/3$ . Thus, the total inertia of the first clustering is

$$[(-3 + 2)^2 + (-2 + 2)^2 + (-1 + 2)^2] + \left[ \left(2 - \frac{14}{3}\right)^2 + \left(5 - \frac{14}{3}\right)^2 + \left(7 - \frac{14}{3}\right)^2 \right] = 2 + \frac{124}{9} = 15.78.$$

The second clustering has centroids  $\mu_1 = -1$ ,  $\mu_2 = 6$ . Its total inertia is

$$[(-3 + 1)^2 + (-2 + 1)^2 + (-1 + 1)^2 + (2 + 1)^2] + [(5 - 6)^2 + (7 - 6)^2] = 14 + 2 = 16.$$

Finding the minimum inertia among all possible  $k$ -clusterings is an infeasible problem to solve exactly at any practical size. Instead, the approach is to iteratively improve on a starting clustering.

### 6.3.1 Lloyd's algorithm

The most-used iteration is known as **Lloyd's algorithm**. Starting with values for the  $k$  centroids, each iteration consists of two steps:

1. Each sample point is assigned to the cluster whose centroid is the nearest. (Ties are broken randomly.)
2. Recalculate the centroids based on the cluster assignments:

$$\mu_j^+ = \frac{1}{|C_j|} \sum_{\mathbf{x} \in C_j} \mathbf{x}.$$

[https://www.  
dropbox.com/  
s/  
d3l4ouuu6i4byxl/  
Section6\\_3\\_  
1.mp4?raw=1](https://www.dropbox.com/s/d3l4ouuu6i4byxl/Section6_3_1.mp4?raw=1)

These steps are repeated alternately until the assignment step does not change any of the clusters. In practice, this almost always happens quickly. Here is a demonstration:

[./\\_media/kmeans\\_demo.mp4](#)

If Lloyd's algorithm converges, it finds a local minimum of total inertia, in the sense that small changes to the cluster assignments cannot decrease it. But there is no guarantee of converging to the global minimum, and often, this does not happen.

### 6.3.2 Practical issues

- **Initialization.** The performance of  $k$ -means depends a great deal on the initial set of centroids. Traditionally, the centroids were chosen as random members of the sample set, but more reliable heuristics, such as *k-means++*, have since become dominant.
- **Multiple runs.** All the initialization methods include an element of randomness, and since the Lloyd algorithm usually converges quickly, it is often run with multiple instances of the initialization. The run with the lowest final inertia is kept.
- **Distance metric.** The Lloyd algorithm often fails to converge for norms other than the 2-norm and must be modified if another norm is preferred.

The algorithm treats  $k$  as a hyperparameter. Increasing  $k$  tends to lower the total inertia—at the extreme, taking  $k = n$  puts each sample in its own cluster, with a total inertia of zero. Hence, one should use silhouette scores or some other measure in order to optimize  $k$ . Occam's Razor dictates preferring smaller values to large ones. There are many suggestions on how to find the choice that gives the most “bang for the buck,” but none is foolproof.

Because of its dependence on the norm, the inertia criterion of  $k$ -means disfavors long, skinny clusters and clusters of unequal dispersion. Essentially, it wants to find spherical blobs of roughly equal size.

**Example 6.11.** We apply  $k$ -means to the *blobs* dataset that has 3 reference clusters, starting with  $k = 2$  clusters. For illustration, we instruct the algorithm to try 3 initializations and report on its progress:

```

from sklearn.cluster import KMeans

X, y = blobs_data()
km2 = KMeans(n_clusters=2, n_init=3, verbose=1, random_state=302)
km2.fit(X)

Initialization complete
Iteration 0, inertia 1059.6987886807747.
Iteration 1, inertia 731.1354331556078.
Iteration 2, inertia 710.0834960683221.
Converged at iteration 2: strict convergence.

Initialization complete
Iteration 0, inertia 991.8887143415382.
Iteration 1, inertia 742.3898509432538.
Iteration 2, inertia 710.0834960683221.
Converged at iteration 2: strict convergence.

Initialization complete
Iteration 0, inertia 1379.944166430521.
Iteration 1, inertia 917.3997019815874.
Iteration 2, inertia 903.6145145551955.
Iteration 3, inertia 845.0278508241745.
Iteration 4, inertia 736.5347784099483.
Iteration 5, inertia 710.0834960683221.
Converged at iteration 5: strict convergence.

KMeans(n_clusters=2, n_init=3, random_state=302, verbose=1)

```

The fitted clustering object can tell us the final inertia and cluster centroids:

```

print(f"final inertia: {km2.inertia_:.5g}")
print("cluster centroids:")
print(km2.cluster_centers_)

final inertia: 710.08
cluster centroids:
[[-2.01977462  2.86247767]
 [ 2.01902057 -0.69722874]]

```

There is a `predict` method that can make cluster assignments for arbitrary points in feature space. In  $k$ -means, this prediction simply tells you which centroid is closest:

```

Q = pd.DataFrame([ [-2,-1], [1,2] ], columns=X.columns)
km2.predict(Q)

array([0, 1], dtype=int32)

```

For the training samples, we don't need to call `predict` to get the cluster assignments. Every fitted clustering object has a `labels_` property that lists the cluster index values. We can use those labels to compute silhouette values:

```

def report(clustering):
    result = X.copy()
    y_hat = clustering.labels_      # cluster assignments
    result["cluster"] = y_hat
    result["sil"] = silhouette_samples(X, y_hat)
    print(f"inertia: {clustering.inertia_:.5g}")
    print("silhouette medians by cluster:")
    print( result.groupby("cluster")["sil"].median() )

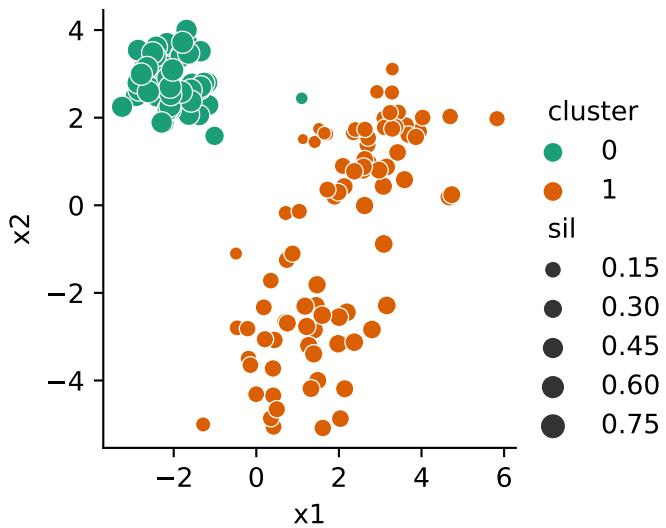
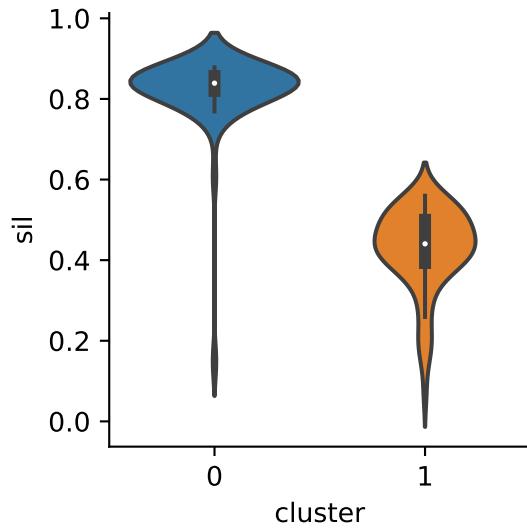
    sns.catplot(data=result,
                 x="cluster", y="sil",
                 kind="violin", height=3
                 );

    sns.relplot(data=result,
                 x="x1", y="x2",
                 hue="cluster", size="sil",
                 height=3, palette="Dark2"
                 );
    return result

report(km2);

inertia: 710.08
silhouette medians by cluster:
cluster
0      0.839082
1      0.440536
Name: sil, dtype: float64

```



It's clear in both plots that one cluster is more tightly packed than the other. Let's repeat the computation for  $k = 3$  clusters:

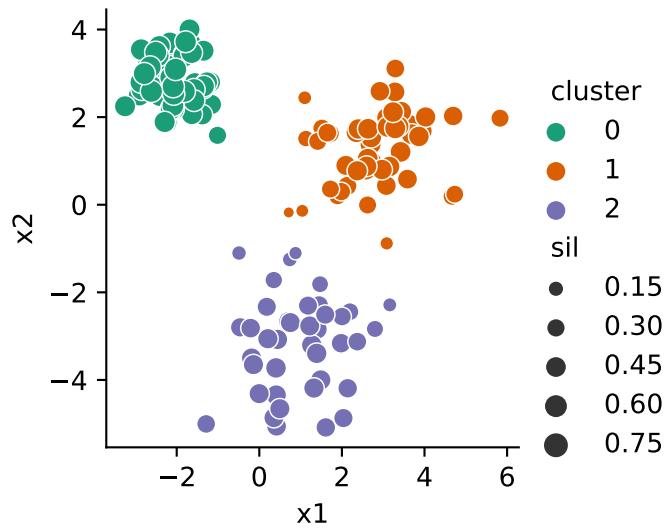
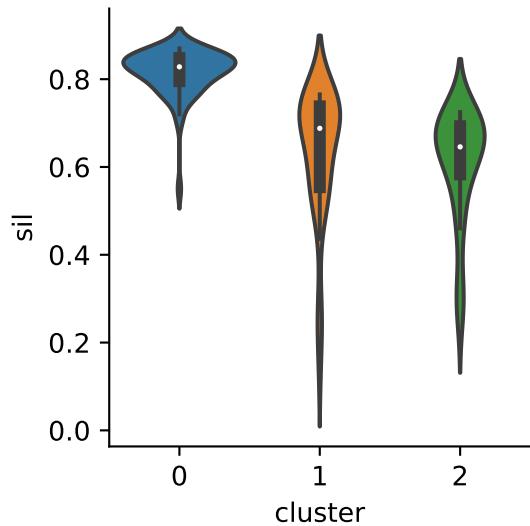
```
km3 = KMeans(n_clusters=3, n_init=1, random_state=302)
km3.fit(X)
report(km3);
```

```
inertia: 203.3
silhouette medians by cluster:
```

```

cluster
0    0.828146
1    0.688014
2    0.645840
Name: sil, dtype: float64

```

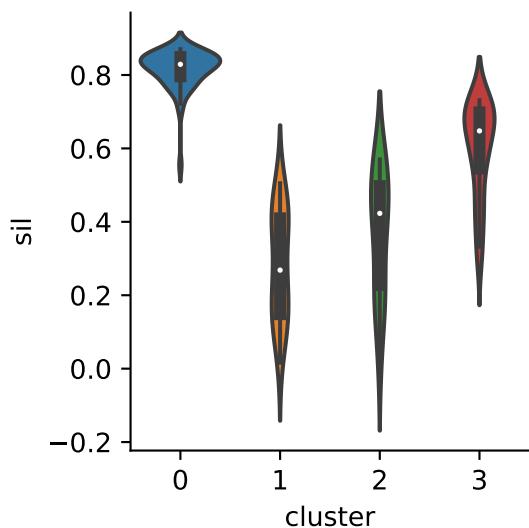


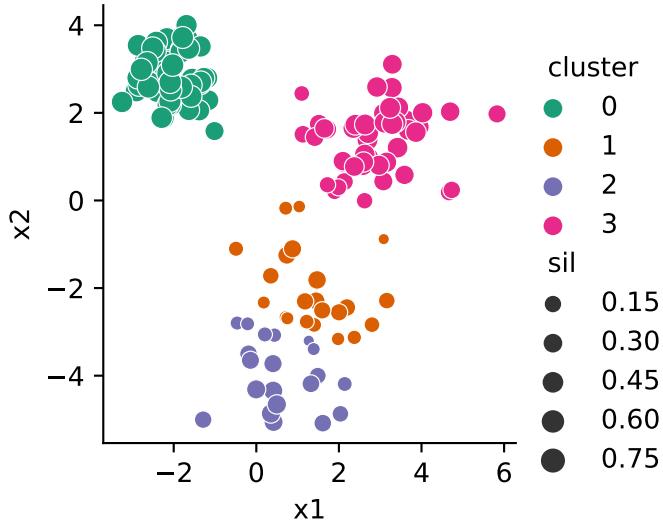
This result shows a modest reduction in silhouette scores for original good cluster, but improvement for the problematic one.

Moving on to  $k = 4$  clusters shows clear degradation of the silhouette scores:

```
km4 = KMeans(n_clusters=4, n_init=1, random_state=302)
km4.fit(X)
report(km4);
```

```
inertia: 166.02
silhouette medians by cluster:
cluster
0    0.829120
1    0.268430
2    0.423066
3    0.647724
Name: sil, dtype: float64
```



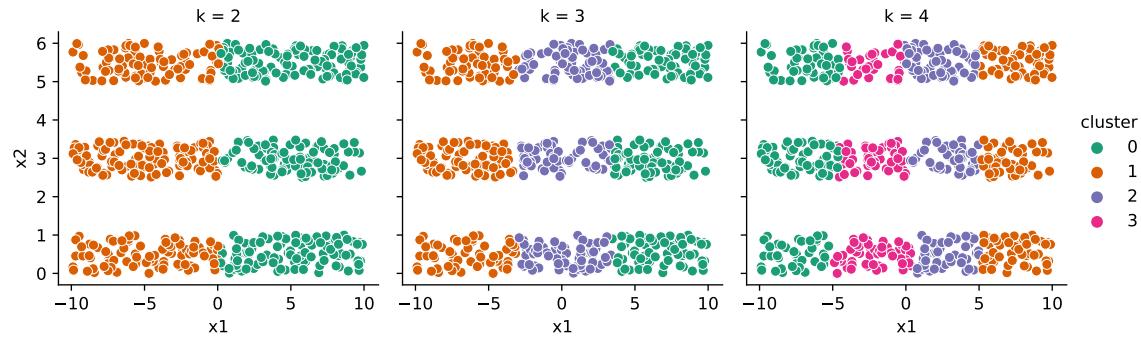


Based on silhouette scores, we would be justified to stop at  $k = 3$  clusters.

**Example 6.12.** The *stripes* dataset does not conform to compact clusters of equal size, and  $k$ -means performs poorly on it:

```
X, y = stripes_data()
results = pd.DataFrame()
for k in [2, 3, 4]:
    km = KMeans(n_clusters=k, n_init=3, random_state=302)
    km.fit(X)
    results = pd.concat(
        results,
        pd.DataFrame( {
            "x1": X["x1"], "x2": X["x2"],
            "cluster": km.labels_,
            "k": k
        })
    )
sns.relplot(data=results,
             x="x1", y="x2",
             hue="cluster", col="k",
             height=3, palette="Dark2"
            );
```

[https://www.dropbox.com/s/fw3had5030zediv/Example6\\_11.mp4?raw=1](https://www.dropbox.com/s/fw3had5030zediv/Example6_11.mp4?raw=1)

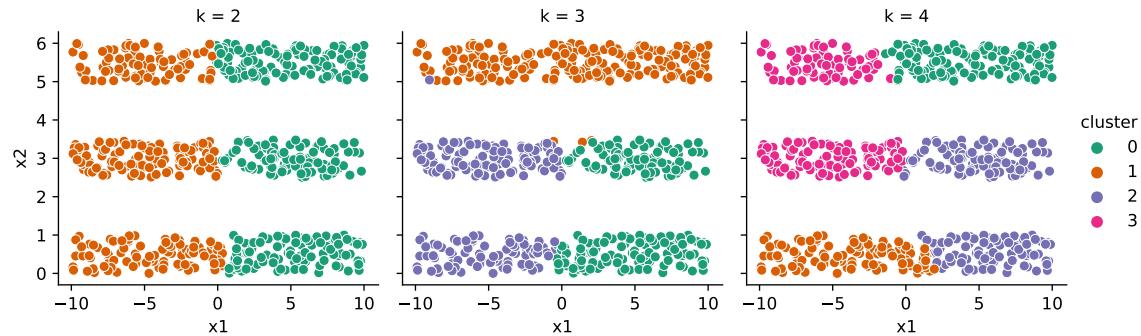


It's usually a good idea to standardize the data before clustering. But that's not much help here:

```

results = pd.DataFrame()
for k in [2, 3, 4]:
    km = make_pipeline(
        StandardScaler(),
        KMeans(n_clusters=k, n_init=3, random_state=302)
    )
    km.fit(X)
    results = pd.concat( (
        results,
        pd.DataFrame( {
            "x1": X["x1"], "x2": X["x2"],
            "cluster": km[1].labels_,
            "k": k
        })
    ) )
sns.relplot(data=results,
    x="x1", y="x2",
    hue="cluster", col="k",
    height=3, palette="Dark2"
);

```



Clustering is hard!

**Example 6.13.** We return to the handwriting recognition dataset. Again we keep only the samples labeled 4, 5, or 6:

```

digits = datasets.load_digits(as_frame=True)[["frame"]]
keep = digits["target"].isin([4, 5, 6])
digits = digits[keep]

X = digits.drop("target", axis="columns")
y = digits["target"]

```

We first fit 3 clusters to the feature matrix:

```

km = make_pipeline(StandardScaler(), KMeans(n_clusters=3, n_init=3, random_state=302))
km.fit(X)
y_hat = km[1].labels_
digits[["cluster number"]] = y_hat
digits[["target", "cluster number"]].head(9)

```

target	cluster number
4	2
5	0
6	1
14	2
15	0
16	1
24	2
25	0
26	1

The adjusted Rand index suggests that we have reproduced the classification very well:

```
ARI = adjusted_rand_score(y, y_hat)
print(f"ARI: {ARI:.4f}")
```

```
ARI: 0.9457
```

However, that conclusion benefits from our prior knowledge of the classification. What if we have no such reference to check against? Let's look over a range of  $k$  choices, recording the median silhouette values for each:

```
results = []
kvals = range(2,7)
for k in kvals:
    km = KMeans(n_clusters=k, n_init=3, random_state=19716)
    km.fit(X)
    y_hat = km.labels_

    sil = np.median( silhouette_samples(X, y_hat) )
    results.append(sil)

pd.Series(results, index=kvals)
```

	0
2	0.237310
3	0.258975
4	0.257826
5	0.249603
6	0.204824

The silhouette score is maximized at  $k = 3$ , which could be considered a reason to choose 3 clusters. While the score for 4 clusters is close to it, we should prefer the less complex model.

## 6.4 Hierarchical clustering

The idea behind hierarchical clustering is to organize all the sample points into a tree structure called a **dendrogram**. At the root of the tree is the entire sample set, while each leaf of the tree is a single sample. Groups of similar samples are connected as nearby relatives in the tree, with less-similar groups located as more distant relatives.

[https://www.dropbox.com/s/kvppujcih39j8xa/Example6\\_13.mp4?raw=1](https://www.dropbox.com/s/kvppujcih39j8xa/Example6_13.mp4?raw=1)

[https://www.dropbox.com/s/kybi7xus8j9b3le/Section6\\_4a.mp4?raw=1](https://www.dropbox.com/s/kybi7xus8j9b3le/Section6_4a.mp4?raw=1)

Dendograms can be found by starting with the root and recursively splitting, or by starting at the leaves and recursively merging. We will describe the latter approach, known as **agglomerative clustering**.

The algorithm begins with  $n$  singleton clusters, i.e.,  $C_i = \{\mathbf{X}_i\}$  for  $i = 1, \dots, n$ . Then, the similarity or distance between each pair of clusters is determined. The pair with the minimum distance is merged, and the process repeats. Effectively, we end up with an entire family of clusterings; we can stop at any number of clusters between  $n$  and 1.

Common ways to define the distance between two clusters  $C_i$  and  $C_j$  are:

- **single linkage** (also called *minimum linkage*)

$$\min_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \{\|\mathbf{x} - \mathbf{y}\|\} \quad (6.3)$$

[https://www.  
dropbox.com/  
s/  
c75mj4898gfmu4o/  
Section6\\_4b.  
mp4?raw=1](https://www.dropbox.com/s/c75mj4898gfmu4o/Section6_4b.mp4?raw=1)

- **complete linkage** (also called *maximum linkage*)

$$\max_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \{\|\mathbf{x} - \mathbf{y}\|\} \quad (6.4)$$

- **average linkage**

$$\frac{1}{|C_i||C_j|} \sum_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \|\mathbf{x} - \mathbf{y}\| \quad (6.5)$$

- **Ward linkage** The increase in inertia resulting from merging  $C_i$  and  $C_j$ , which is equal to

$$\frac{|C_i||C_j|}{|C_i| + |C_j|} \left\| \mu_i - \mu_j \right\|_2^2, \quad (6.6)$$

where  $\mu_i$  and  $\mu_j$  are the centroids of  $C_i$  and  $C_j$ .

Single linkage only pays attention to the gaps between clusters, not the size or spread of them. Complete linkage, on the other hand, wants to keep every cluster packed tightly together. Average linkage is a compromise between these extremes. Agglomerative clustering with Ward linkage amounts to trying to minimize the increase of inertia with each merger. In that sense, it has the same objective as  $k$ -means, but it is usually not as successful at minimizing inertia.

Unlike  $k$ -means, which has to be able to compute the distances between the samples and changing centroid locations, agglomerative clustering with single, complete, or average linkage needs only the pairwise distances between samples and can be found from a distance matrix in place of the samples themselves.

**Example 6.14.** Given clusters  $C_1 = \{-3, -2, -1\}$  and  $C_2 = \{3, 4, 5\}$ , find the different linkages between them.

*Solution.* **Ward.** The centroids of the clusters are  $-2$  and  $4$ . So the linkage is

$$\frac{3 \cdot 3}{3 + 3} 6^2 = 54.$$

**Single.** The pairwise distances between members of  $C_1$  and  $C_2$  form a  $3 \times 3$  matrix:

	-3	-2	-1
3	6	5	4
4	7	6	5
5	8	7	6

(Note that this is a submatrix of the full  $6 \times 6$  distance matrix.) The single linkage is therefore 4.

**Complete.** The maximum of the matrix above is 8.

**Average.** The average value of the matrix entries is  $54/9$ , which is 6.

**Example 6.15.** Let's use 5 sample points in the plane, and agglomerate them by single linkage. The `pairwise_distances` function converts sample points into a distance matrix:

```
X = np.array( [[-2,-1] , [-2,-2] , [1,0.5] , [0,2] , [-1,1]] )
D = pairwise_distances(X, metric="euclidean")
D

array([[0.          , 1.          , 3.35410197, 3.60555128, 2.23606798],
       [1.          , 0.          , 3.90512484, 4.47213595, 3.16227766],
       [3.35410197, 3.90512484, 0.          , 1.80277564, 2.06155281],
       [3.60555128, 4.47213595, 1.80277564, 0.          , 1.41421356],
       [2.23606798, 3.16227766, 2.06155281, 1.41421356, 0.        ]])
```

[https://www.dropbox.com/s/wn5frbym4g9j1m/Example6\\_14b.mp4?raw=1](https://www.dropbox.com/s/wn5frbym4g9j1m/Example6_14b.mp4?raw=1)

(video example is different from the text)

The minimum value in the upper triangle of the distance matrix is in row 0, column 1. So our first merge results in the cluster  $C_1 = \{\mathbf{X}_0, \mathbf{X}_1\}$ . The next-smallest entry in the upper triangle is at position  $(3,4)$ , so we want to merge those samples together next, resulting in

$$C_1 = \{\mathbf{X}_0, \mathbf{X}_1\}, C_2 = \{\mathbf{X}_3, \mathbf{X}_4\}, C_3 = \{\mathbf{X}_2\}.$$

The next-smallest element in the matrix is at  $(2,3)$ , resulting in

$$C_1 = \{\mathbf{X}_0, \mathbf{X}_1\}, C_2 = \{\mathbf{X}_2, \mathbf{X}_3, \mathbf{X}_4\}.$$

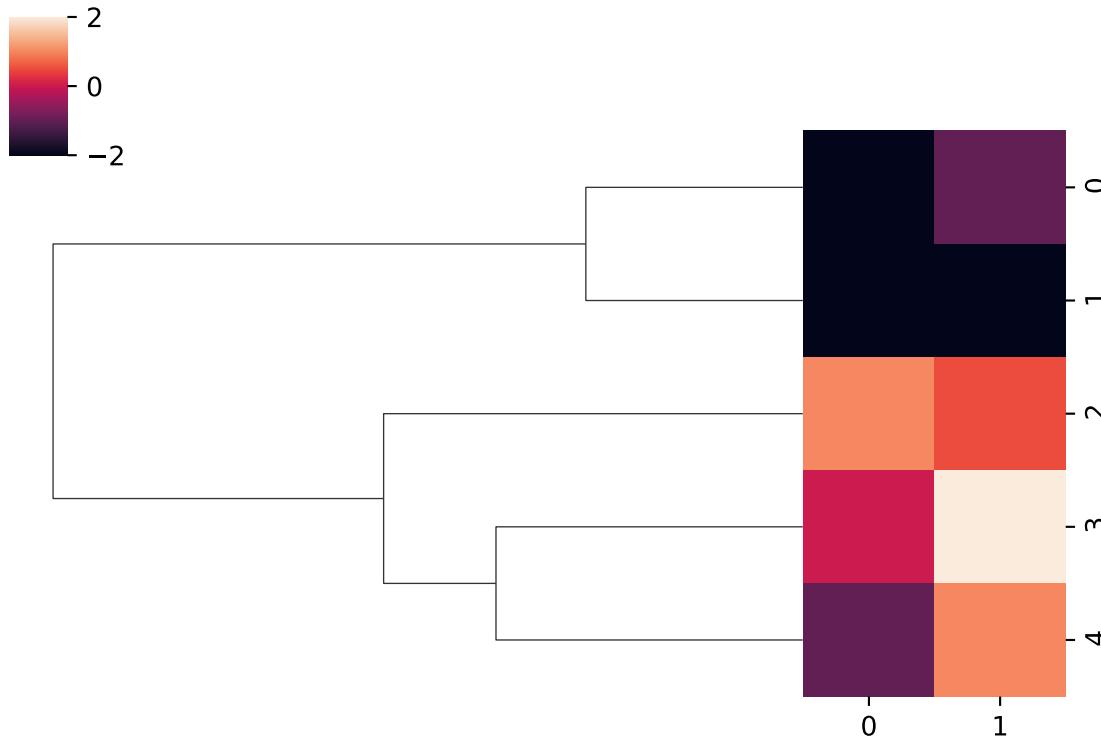
The final merge is to combine these.

The entire dendrogram can be visualized with seaborn's `clustermap`:

```

sns.clustermap(X,
    col_cluster=False,
    dendrogram_ratio=(.75,.15),
    figsize=(6,4)
);

```



The root of the tree above is at the left, with the leaves at the right. The leaves have been ordered so that the lines in the dendrogram don't ever cross. The two colored columns show the values of the features of the samples.

The horizontal position in the dendrogram indicates the linkage value, which is largest at the left. Working from right to left, we first see the merger of samples 0 and 1 at the minimum linkage in the entire distance matrix. The next merger is between  $\mathbf{X}_3$  and  $\mathbf{X}_4$ , and so on.

We can choose to stop at any horizontal position (linkage value). Based on the distance matrix, if we stop at a linkage value of 2.0, then we perform only the first 2 merges and get 3 clusters. The largest gap between merges is in the merge from 2 clusters to 1, which could be used to justify  $k = 2$  as the best number of clusters.

The `AgglomerativeClustering` class in `sklearn.cluster` performs the clustering process to create a learner object.

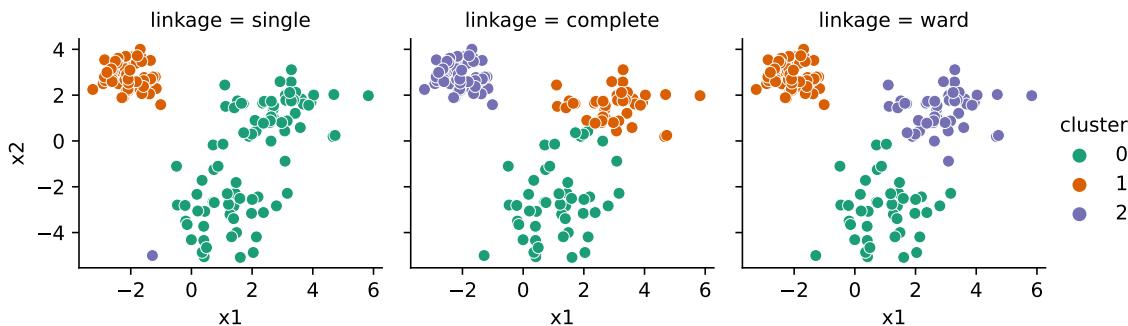
**Example 6.16.** We define a function that allows us to run all three linkages for a given feature matrix:

```
from sklearn.cluster import AgglomerativeClustering

def run_experiment(X):
    results = pd.DataFrame()
    data = X.copy()
    for linkage in ["single", "complete", "ward"]:
        agg = AgglomerativeClustering(n_clusters=3, linkage=linkage)
        agg.fit( X )
        data["cluster"] = agg.labels_
        data["linkage"] = linkage
    results = pd.concat( (results, data) )
    return results
```

We first try the *blobs* dataset:

```
X, y = blobs_data()
results = run_experiment(X)
sns.relplot(data=results,
            x="x1", y="x2",
            hue="cluster", col="linkage",
            height=2.5, palette="Dark2"
            );
```



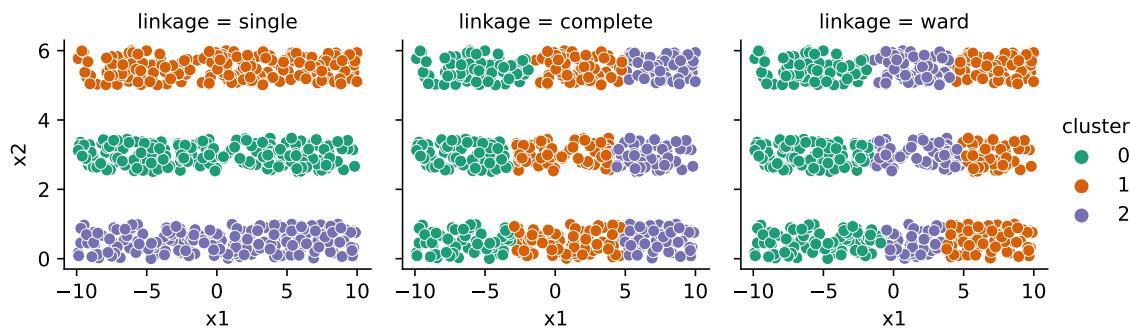
As you can see, the simple linkage was badly confused by the two blobs that nearly run together. The others are good at reproducing the ball-like clusters.

Next, we try the *stripes* data:

```

X, y = stripes_data()
results = run_experiment(X)
sns.relplot(data=results,
            x="x1", y="x2",
            hue="cluster", col="linkage",
            height=2.5, palette="Dark2"
            );

```



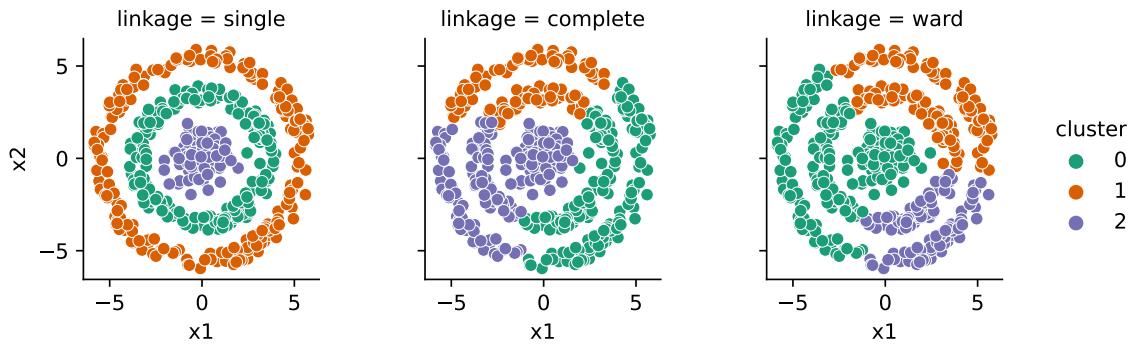
Now the tendency of complete and Ward linkages to find compact, roughly spherical clusters is a bug, not a feature. They group together points across stripes rather than clusters extending lengthwise. The geometric flexibility of single linkage pays off here.

Finally, we try the most demanding test, *bullseye*:

```

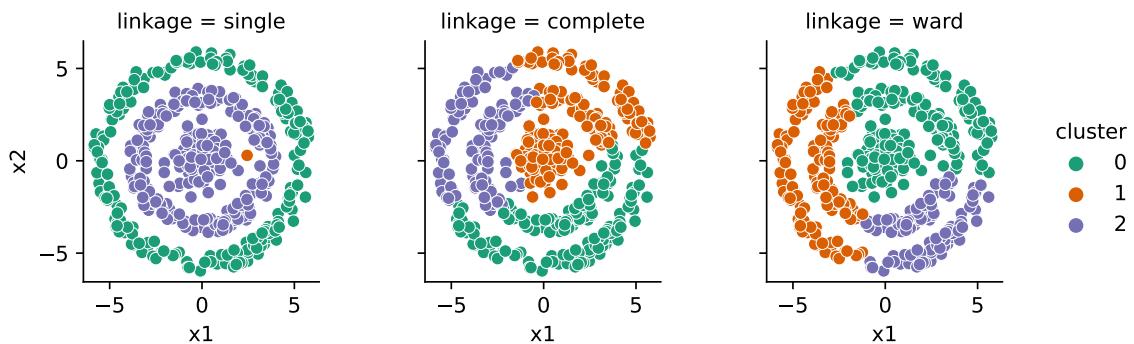
X ,y = bullseye_data()
results = run_experiment(X)
fig = sns.relplot(data=results,
                   x="x1", y="x2",
                   hue="cluster", col="linkage",
                   height=2.5, palette="Dark2"
                   )
fig.set(aspect=1);

```



The single linkage is the only one to cluster the rings properly. However, it's a delicate situation, and it can be sensitive to individual samples. Here, for instance, we add just one sample to the dataset and get a major change:

```
X = pd.concat( ( X, pd.DataFrame({"x1": [0], "x2": [2.25]}) ) )
results = run_experiment(X)
fig = sns.relplot(data=results,
                    x="x1", y="x2",
                    hue="cluster", col="linkage",
                    height=2.5, palette="Dark2"
                  )
fig.set(aspect=1);
```



That instability can make single linkage difficult to work with.

**Example 6.17.** Let's try agglomerative clustering to “discover” the species of the penguins, pretending we don't know them in advance. First, let's recall how many of each species we actually do have:

[https://www.dropbox.com/s/h4as5ck4sufn45a/Example6\\_16.mp4?raw=1](https://www.dropbox.com/s/h4as5ck4sufn45a/Example6_16.mp4?raw=1)

```

penguins = sns.load_dataset("penguins").dropna()
features = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]
X = penguins[features]
y = penguins["species"]
y.value_counts()

```

species
Adelie
Gentoo
Chinstrap

Our first attempt is single linkage. Because 2-norm distances are involved, we use standardization in a pipeline with the clustering method. After fitting, the `labels_` property of the cluster object is a vector of cluster assignments.

```

from sklearn.cluster import AgglomerativeClustering

single = AgglomerativeClustering(n_clusters=3, linkage="single")
pipe = make_pipeline(StandardScaler(), single)
pipe.fit(X)
y_hat = single.labels_           # cluster assignments
penguins["single"] = y_hat
penguins.loc[:, ["species", "single"]]

```

	species	single
0	Adelie	0
31	Adelie	0
58	Adelie	0
84	Adelie	0
110	Adelie	0
136	Adelie	0
162	Chinstrap	0
188	Chinstrap	0
214	Chinstrap	0
240	Gentoo	2
267	Gentoo	2
294	Gentoo	2
320	Gentoo	2

Perhaps *Gentoo* is associated with cluster number 2, but the situation with the other species is less clear. Here are the value counts:

```
print("single linkage results:")
penguins["single"].value_counts()
```

```
single linkage results:
```

single	
0	213
2	119
1	1

As we saw with the *bullseye* dataset in Example 6.16, single linkage is susceptible to declaring one isolated point to be a cluster, while grouping together other points we would like to separate. Here is the ARI for this clustering, compared to the true classification:

```
from sklearn.metrics import adjusted_rand_score
ARI = adjusted_rand_score(y, y_hat)
print(f"single linkage ARI: {ARI:.4f}")
```

```
single linkage ARI: 0.6506
```

Now let's try Ward linkage (which is the default):

```
ward = AgglomerativeClustering(n_clusters=3, linkage="ward")
pipe = make_pipeline(StandardScaler(), ward)
pipe.fit(X)
y_hat = ward.labels_
penguins["ward"] = y_hat

print("Ward linkage results:")
print(penguins["ward"].value_counts())
```

```
Ward linkage results:
```

1	157
0	119
2	57

Name: ward, dtype: int64

This result looks more promising. The ARI confirms that hunch:

```

ARI = adjusted_rand_score(y, penguins["ward"])
print(f"Ward linkage ARI: {ARI:.4f}")

```

Ward linkage ARI: 0.9132

If we guess at the likely correspondence between the cluster numbers and the different species, then we can find the confusion matrix:

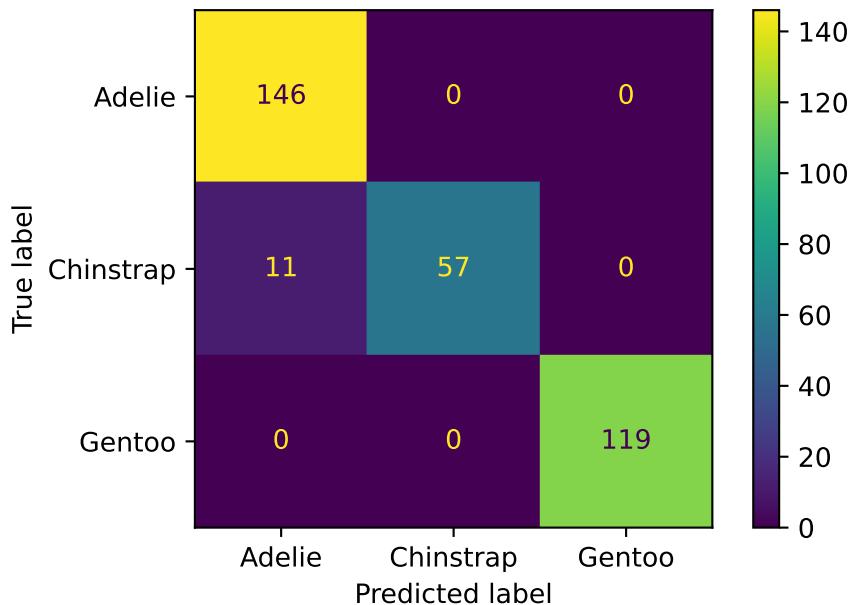
```

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Convert cluster numbers into labels:
replacements = {1: "Adelie", 0: "Gentoo", 2: "Chinstrap"}
y_hat = penguins["ward"].replace(replacements)
C = confusion_matrix(y, y_hat)

ConfusionMatrixDisplay(C, display_labels=y.unique()).plot();

```



While the performance of clustering as a classifier in Example 6.17 is inferior to supervised methods we used for that purpose, the clustering process reveals that the Gentoo penguins are the most dissimilar from the other species. By inspecting the mislabeled cases in the confusion matrix, further insight might be gained about what particular traits can make the other species difficult to distinguish. In clustering, the goal is often more about insight than prediction.

[https://www.dropbox.com/s/uzu954euu5twqxh/Example6\\_17.mp4?raw=1](https://www.dropbox.com/s/uzu954euu5twqxh/Example6_17.mp4?raw=1)

## Exercises

**Exercise 6.1.** (6.1) Prove that the angular distance between any nonzero vector and itself is zero.

**Exercise 6.2.** (6.1) Find an example for which the cosine distance does not satisfy the triangle inequality. That is, find three vectors  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mathbf{z}$  such that

$$\text{dist}(\mathbf{x}, \mathbf{y}) > \text{dist}(\mathbf{x}, \mathbf{z}) + \text{dist}(\mathbf{z}, \mathbf{y}).$$

(Hint: it's enough to play around with some simple vectors in two dimensions. Use vectors for which the angle between is a multiple of  $\pi/6$  or  $\pi/4$ .)

**Exercise 6.3.** (6.1) Here is a feature matrix.

$$\mathbf{X} = \begin{bmatrix} -1 & -1 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & -2 \\ 1 & 3 & 1 \end{bmatrix}.$$

- (a) Find the associated distance matrix using the 1-norm.
- (b) Find the associated distance matrix using the infinity-norm.

**Exercise 6.4.** (6.2) Here are two clusterings on 6 samples.

$$\begin{aligned} A : C_1 &= \{\mathbf{X}_1, \mathbf{X}_4\}, C_2 = \{\mathbf{X}_2, \mathbf{X}_3, \mathbf{X}_5, \mathbf{X}_6\} \\ B : C_1 &= \{\mathbf{X}_1, \mathbf{X}_2\}, C_2 = \{\mathbf{X}_3, \mathbf{X}_4\}, C_3 = \{\mathbf{X}_5, \mathbf{X}_6\} \end{aligned}$$

- (a) Find the Rand index between these clusterings.
- (b) Find the silhouette values for all samples in clustering A.

**Exercise 6.5.** (6.3) Let  $z$  be a positive number, and consider the 12 planar samples

$$\begin{aligned} [z, -3], [z, -2], [z, -1], [z, 1], [z, 2], [z, 3], \\ [-z, -3], [-z, -2], [-z, -1], [-z, 1], [-z, 2], [-z, 3]. \end{aligned}$$

- (a) Using the 2-norm, calculate the inertia, as a function of  $z$ , of the clustering

$$C_1 = \{[z, j] : j = -3, -2, -1, 1, 2, 3\}, \quad C_2 = \{[-z, j] : j = -3, -2, -1, 1, 2, 3\}.$$

(This divides the sample points by the  $x$ -axis.)

(b) Using the 2-norm, calculate the inertia, as a function of  $z$ , of the clustering

$$C_1 = \{[-z, j] \text{ and } [z, j] : j = -3, -2, -1\}, \quad C_2 = \{[-z, j] \text{ and } [z, j] : j = 1, 2, 3\}.$$

(This divides the sample points by the  $y$ -axis.)

(c) For which values of  $z$ , if any, does clustering from part (a) have less inertia than the clustering from part (b)?

**Exercise 6.6.** (6.4) Here is a distance matrix for samples  $\mathbf{X}_1, \dots, \mathbf{X}_5$ :

$$\begin{bmatrix} 0 & 2 & 3.5 & 5 & 6 \\ 2 & 0 & 2.5 & 3 & 4 \\ 3.5 & 2.5 & 0 & 1 & 1.5 \\ 5 & 3 & 1 & 0 & 1 \\ 6 & 4 & 1.5 & 1 & 0 \end{bmatrix}$$

(a) Compute the average linkage between the clusters  $C_1 = \{\mathbf{X}_1, \mathbf{X}_3\}$  and  $C_2 = \{\mathbf{X}_2, \mathbf{X}_4, \mathbf{X}_5\}$ .

(b) Compute the single linkage between the clusters  $C_1 = \{\mathbf{X}_1, \mathbf{X}_3, \mathbf{X}_4\}$  and  $C_2 = \{\mathbf{X}_2, \mathbf{X}_5\}$ .

**Exercise 6.7.** (6.4) Find the agglomerative clustering, using complete linkage, for the samples given in Exercise 6.6. (This means finding four individual merge steps based on the distance matrix.)

# 7 Networks

```
import numpy as np
from numpy.random import default_rng
import pandas as pd
import seaborn as sns
from numpy.random import default_rng
import matplotlib as mpl
mpl.rcParams["figure.figsize"] = [4.8, 3.6]
```

Many phenomena have a natural network structure. Obvious examples are social networks, transportation networks, and the Web, but other examples include cellular protein interactions, scientific citations, ecological predation, and many others.

## 7.1 Graphs

In mathematics, a network is represented as a **graph**.

**Definition 7.1.** A graph is a collection of **nodes** (also called *vertices*) and **edges** that connect pairs of nodes.

In an **undirected graph**, the edge  $(a, b)$  is identical to  $(b, a)$ , while in a **directed graph** or **digraph**,  $(a, b)$  and  $(b, a)$  are different potential edges.

In either type of graph, each edge might be labeled with a numerical value, which results in a **weighted graph**.

Undirected, unweighted graphs will give us plenty to handle, and we will not go beyond them. We also will not allow graphs that have an edge from a node to itself.

We will use the NetworkX package to work with graphs.

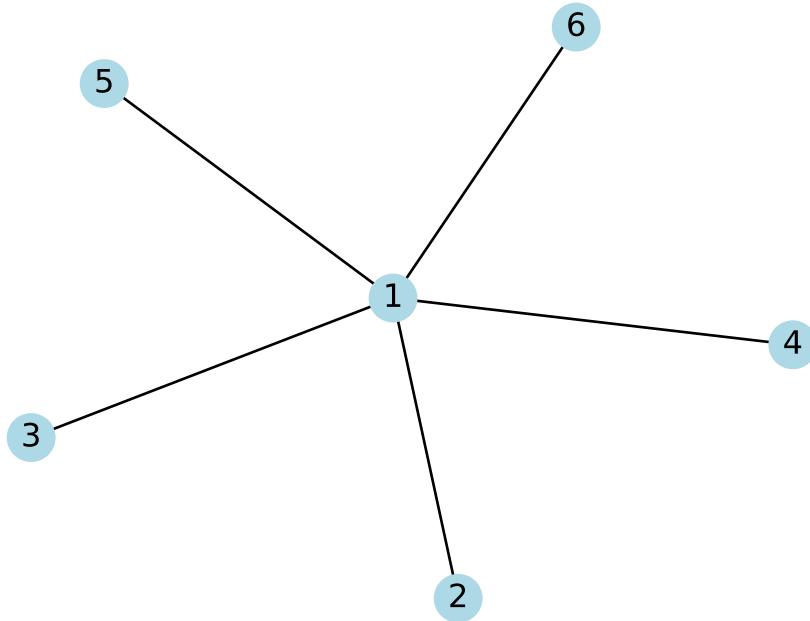
```
import networkx as nx
```

**Example 7.1.** One way to create a graph is from a list of edges:

```

star = nx.Graph( [ (1,2),(1,3),(1,4),(1,5),(1,6) ] )
nx.draw(star, with_labels=True, node_color="lightblue")

```



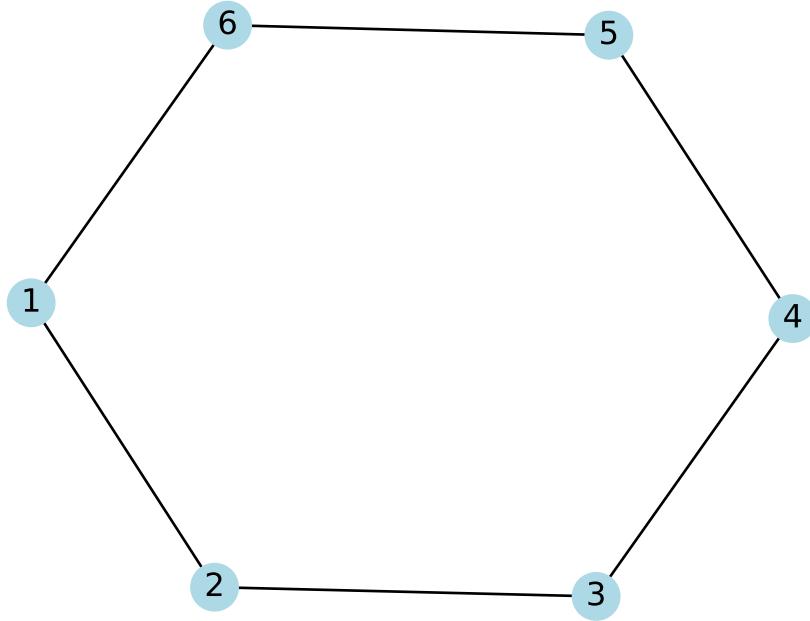
Another way to create a graph is to give the start and end nodes of the edges as columns in a data frame:

```

network = pd.DataFrame(
    {'from': [1,2,3,4,5,6], 'to': [2,3,4,5,6,1]}
)
print(network)
H = nx.from_pandas_edgelist(network, 'from', 'to')
nx.draw(H, with_labels=True, node_color="lightblue")

```

	from	to
0	1	2
1	2	3
2	3	4
3	4	5
4	5	6
5	6	1



**Example 7.2.** We can deconstruct a graph object into its constituent nodes and edges. The results have special types that can be converted into sets, lists, or other objects.

```

print("Nodes as a list:")
print( list(star.nodes) )
print("\nNodes as an Index:")
print( pd.Index(star.nodes) )

```

```

Nodes as a list:
[1, 2, 3, 4, 5, 6]

```

```

Nodes as an Index:
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')

```

It's also easy to get a list of any node's neighbors using bracket indexing:

```

print( "Neighbors of node 3 in graph H:", list(H[3]) )

```

```

Neighbors of node 3 in graph H: [2, 4]

```

**Example 7.3.** There are many ways to read graphs from, and write them to, files. For example, here is a friend network among Twitch users:

```
twitch = nx.read_edgelist(  
    "_datasets/musae_edges.csv",  
    delimiter=',',  
    nodetype=int  
)
```

The file just imported has a pair of node labels separated by commas on each line, representing one edge. The node labels can be any strings, but we overrode that above to interpret them as integers.

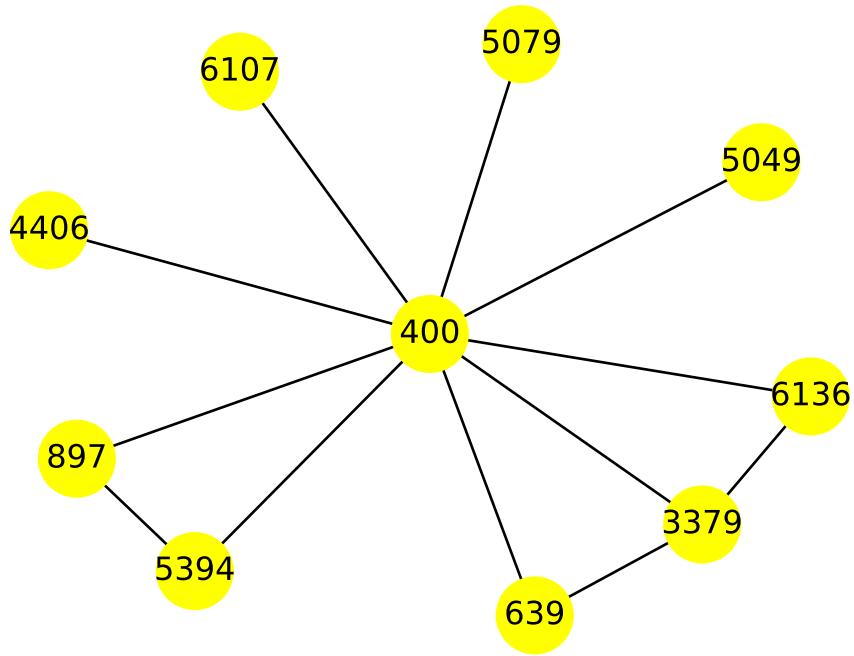
We will explore many functions that tell us facts about a graph. Here are two of the most fundamental:

```
print("Twitch network has",  
      twitch.number_of_nodes(),  
      "nodes and",  
      twitch.number_of_edges(),  
      "edges"  
)
```

```
Twitch network has 7126 nodes and 35324 edges
```

Due to its large size, this graph is difficult to draw in its entirety. We can zoom in on a subset by selecting a node and its **ego graph**, which includes its neighbors along with all edges between the captured nodes:

```
ego = nx.ego_graph(twitch, 400)  
nx.draw(ego, with_labels=True, node_size=800, node_color="yellow")
```

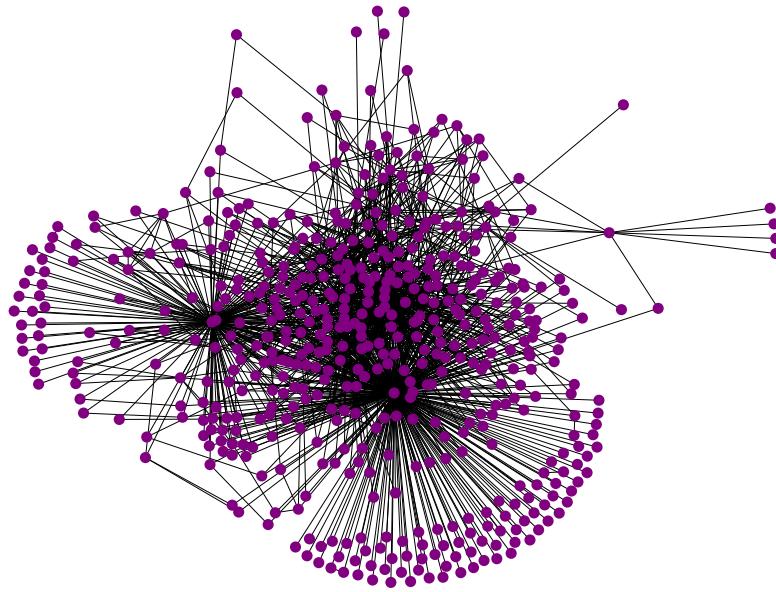


Notice that the nodes of the ego network have the same labels as they did in the graph that it was taken from. We can widen the ego graph to include the ego graphs of all the neighbors:

```
big_ego = nx.ego_graph(twitch, 400, radius=2)
print(big_ego.number_of_nodes(), "nodes and",
      big_ego.number_of_edges(), "edges")

pos = nx.spring_layout(big_ego, iterations=60)
nx.draw(big_ego,
        pos=pos, width=0.2, node_size=10, node_color="purple")
```

528 nodes and 1567 edges



The reason for the two-step process in making the drawing above is that computing the node positions via springs takes a hidden computational iteration. By calling that iteration explicitly, we were able to stop it early and save time.

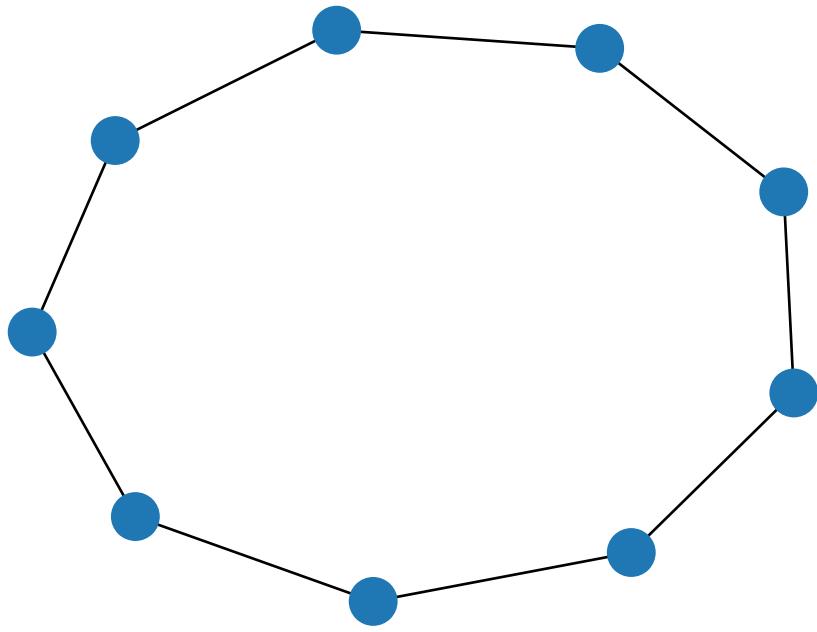
**i** Note

As pointed out [in its documentation](#), the drawing tools provided by NetworkX are bare-bones, and you should look into alternatives if you will be using them heavily.

### 7.1.1 A graph menagerie

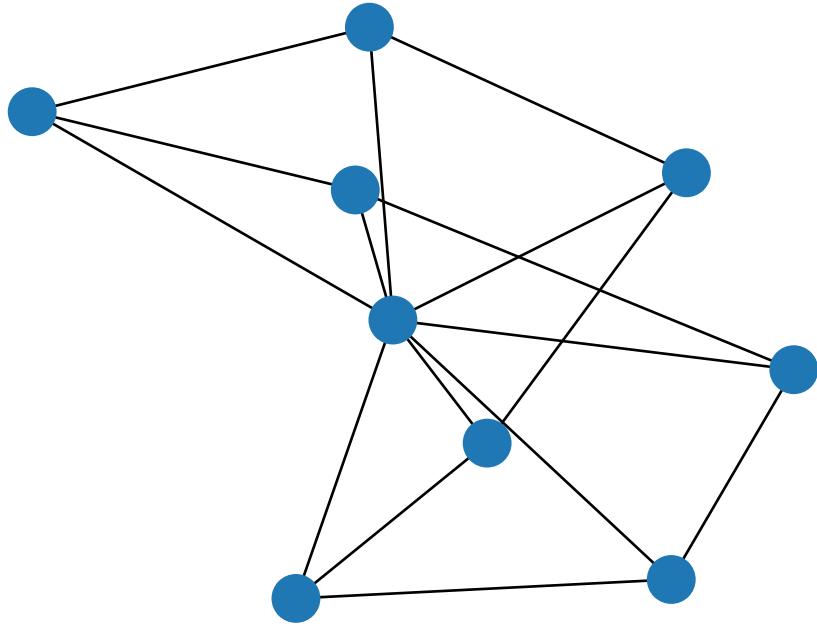
There are functions that generate different well-studied types of graphs. The first graph constructed above is a **star graph**, and the graph H above is a **cycle graph**.

```
nx.draw(nx.cycle_graph(9))
```



A cross between the star and the cycle is a **wheel graph**.

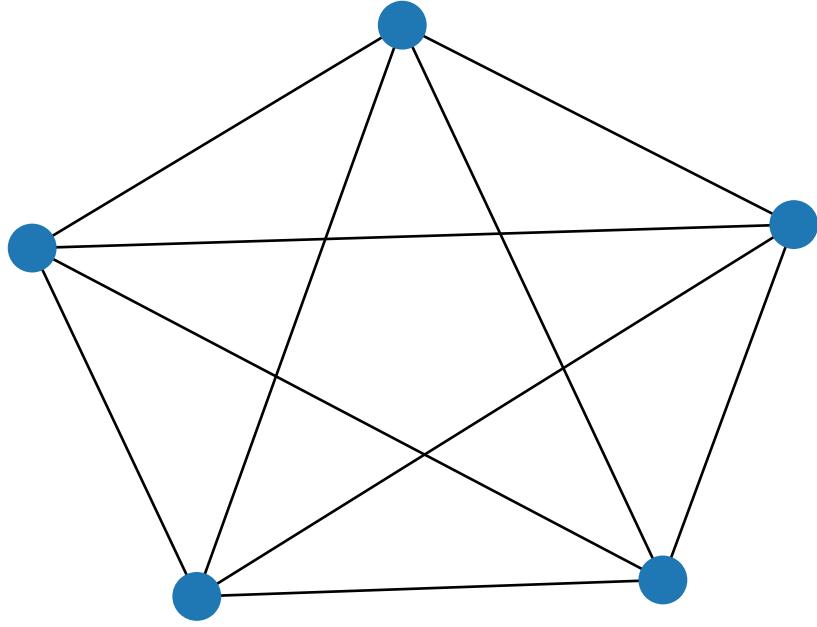
```
nx.draw(nx.wheel_graph(9))
```



A **complete graph** is one that has every possible edge.

```
K5 = nx.complete_graph(5)
print("5 nodes, ", nx.number_of_edges(K5), "edges")
nx.draw(K5)
```

5 nodes, 10 edges



In a graph on  $n$  nodes, there are

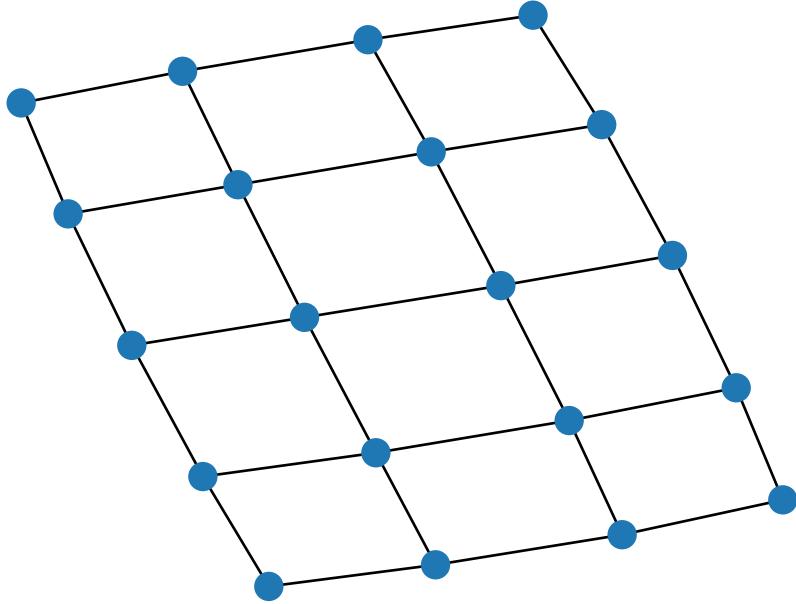
$$\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$$

unique pairs of distinct nodes. Hence, there are  $\binom{n}{2}$  edges in the undirected complete graph on  $n$  nodes.

A **lattice graph** has a regular structure, like graph paper.

```
lat = nx.grid_graph( (5,4) )
print(lat.number_of_nodes(), "nodes,", lat.number_of_edges(), "edges")
nx.draw(lat, node_size=100)
```

20 nodes, 31 edges



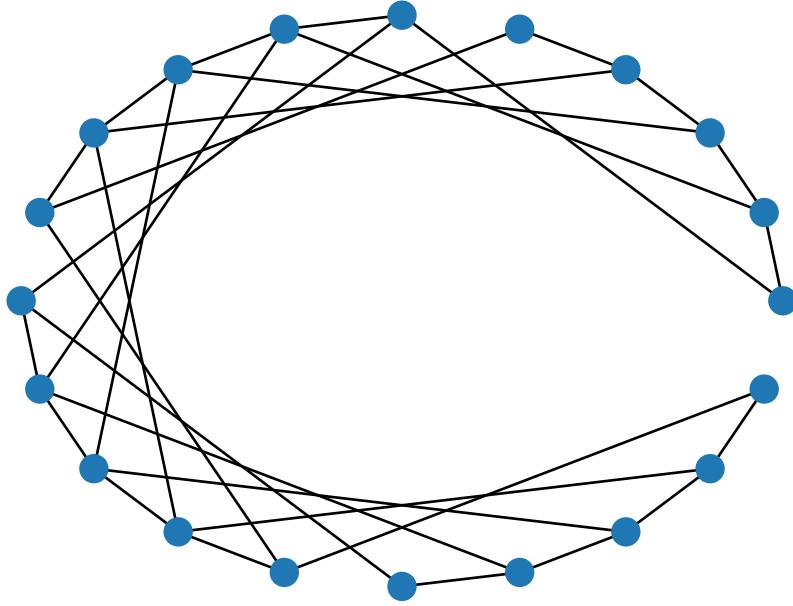
In an  $m \times n$  lattice graph, there are  $m - 1$  edges in one direction repeated  $n$  times, plus  $n - 1$  edges in the other direction, repeated  $m$  times. Thus there are

$$(m - 1)n + (n - 1)m = 2mn - (m + n)$$

edges altogether.

There are different ways to draw a particular graph in the plane, as determined by the positions of the nodes. The default is to imagine that the edges are springs pulling on the nodes. But there are alternatives that may be useful at times.

```
nx.draw_circular(lat, node_size=100)
```



As you can see, it's not easy to tell how similar two graphs are by comparing renderings of them.

### 7.1.2 Adjacency

The most salient feature of a graph is how each node connects to others.

**Definition 7.2.** Nodes are said to be **adjacent** if they share an edge. The **neighbors** of a node are the nodes that are adjacent to it.

We can represent all the adjacency relationships at once using a matrix.

**Definition 7.3.** If we number the nodes of a graph from 0 to  $n - 1$ , then its **adjacency matrix** is the  $n \times n$  matrix whose entries are

$$A_{ij} = \begin{cases} 1, & \text{if } (i, j) \text{ is an edge,} \\ 0, & \text{otherwise.} \end{cases}$$

**Example 7.4.** The star graph defined in Section 7.1.1 has the following adjacency matrix:

```

A = nx.adjacency_matrix(star)
A

<6x6 sparse matrix of type '<class 'numpy.int64'>'  

with 10 stored elements in Compressed Sparse Row format>

```

The matrix  $A$  is not stored in the format we have been used to. In a large network we would expect most of its entries to be zero, so it makes more sense to store it as a *sparse matrix*, where we keep track of only the nonzero entries.

```
print(A)
```

```
(0, 1)    1  

(0, 2)    1  

(0, 3)    1  

(0, 4)    1  

(0, 5)    1  

(1, 0)    1  

(2, 0)    1  

(3, 0)    1  

(4, 0)    1  

(5, 0)    1
```

We can easily convert  $A$  to a standard array, if it is not too large to fit in memory.

```
A.toarray()
```

```
array([[0, 1, 1, 1, 1, 1],  

       [1, 0, 0, 0, 0, 0],  

       [1, 0, 0, 0, 0, 0],  

       [1, 0, 0, 0, 0, 0],  

       [1, 0, 0, 0, 0, 0],  

       [1, 0, 0, 0, 0, 0]])
```

In an undirected graph, we have

$$A_{ij} = A_{ji}$$

for all valid indices, and we say that  $A$  is **symmetric**.

### 7.1.3 Degree

**Definition 7.4.** The **degree** of a node is the number of neighbors it has. The **average degree** of a graph is the mean of the degrees of all of its nodes.

It's common to use the notation  $\bar{k}$  for the average degree of a graph.

The **degree** property of a NetworkX graph gives a dictionary-style object of all nodes with their degrees.

**Example 7.5.** Continuing with the graphs from Example 7.3:

```
ego.degree
```

```
DegreeView({897: 2, 400: 9, 5394: 2, 3379: 3, 4406: 1, 5079: 1, 6136: 2, 5049: 1, 6107: 1, 639: 2})
```

The result here can be a bit awkward to work with; it's actually a *generator* of a list, rather than the list itself. (This “lazy” attitude is useful when dealing with very large networks.) So, for instance, we can collect it into a list of ordered tuples:

```
list(ego.degree)
```

```
[(897, 2),  
(400, 9),  
(5394, 2),  
(3379, 3),  
(4406, 1),  
(5079, 1),  
(6136, 2),  
(5049, 1),  
(6107, 1),  
(639, 2)]
```

It can be convenient to use a series or frame to keep track of quantities like degree that are associated with nodes:

```
nodes = pd.Index(ego.nodes)  
degrees = pd.Series(dict(ego.degree), index=nodes)  
degrees.head()
```

	0
897	2
400	9
5394	2
3379	3
4406	1

We can then do familiar calculations:

```
print("average degree of ego graph:", degrees.mean())
```

```
average degree of ego graph: 2.4
```

As we are about to see, though, there's an easier way to compute the average degree.

Here is our first nontrivial fact about graphs.

**Theorem 7.1.** *Suppose a graph has  $n$  nodes and  $m$  edges. Then its average degree satisfies*

$$\bar{k} = \frac{2m}{n},$$

*Proof.* If we sum the degrees of all the nodes, we must get  $2m$ , because each edge will have been contributed twice to the summation. Dividing by  $n$  gives the mean.

□

## 7.2 Random graphs

One way of understanding a real-world network is by comparing it to ones that are constructed randomly, but according to relatively simple rules. The idea is that if the real network behaves similarly to members of some random family, then perhaps it is constructed according to similar principles.

### 7.2.1 Erdős-Rényi graphs

Our first construction gives every potential edge an equal chance to exist.

Start with  $n$  nodes and no edges. Suppose you have a weighted coin that comes up heads  $(100p)\%$  of the time. For each pair of nodes, you toss the coin, and if it comes up heads, you add the edge between those nodes. This is known as an *ER graph*.

**Definition 7.5.** An **Erdős-Rényi graph** (ER graph) is a graph in which each potential edge occurs with a fixed probability  $p$ .

Since there are  $\binom{n}{2}$  unique unordered pairs among  $n$  nodes, the mean number of edges in an ER graph is

$$p \binom{n}{2} = \frac{pn(n-1)}{2}.$$

This fact is usually stated in terms of the average node degree.

**Theorem 7.2.** *The average degree  $\bar{k}$  satisfies*

$$\mathbb{E} [\bar{k}] = \frac{1}{n} pn(n-1) = p(n-1),$$

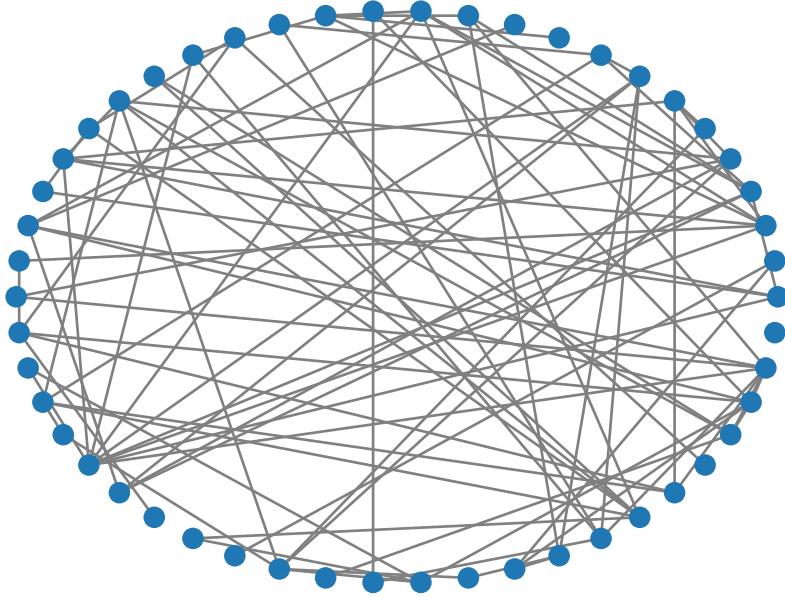
where the expectation operation is taken over all realizations of ER graphs on  $n$  nodes.

There are two senses of averaging going on in Theorem 7.2:. We take the average (expectation operation) over all random instances of the average degree over all nodes within the instance.

**Example 7.6.**

```
n, p = 50, 0.08
ER = nx.erdos_renyi_graph(n, p, seed=2)
print(ER.number_of_nodes(),"nodes",ER.number_of_edges(),"edges")
nx.draw_circular(ER,node_size=50,edge_color="gray")
```

50 nodes, 91 edges

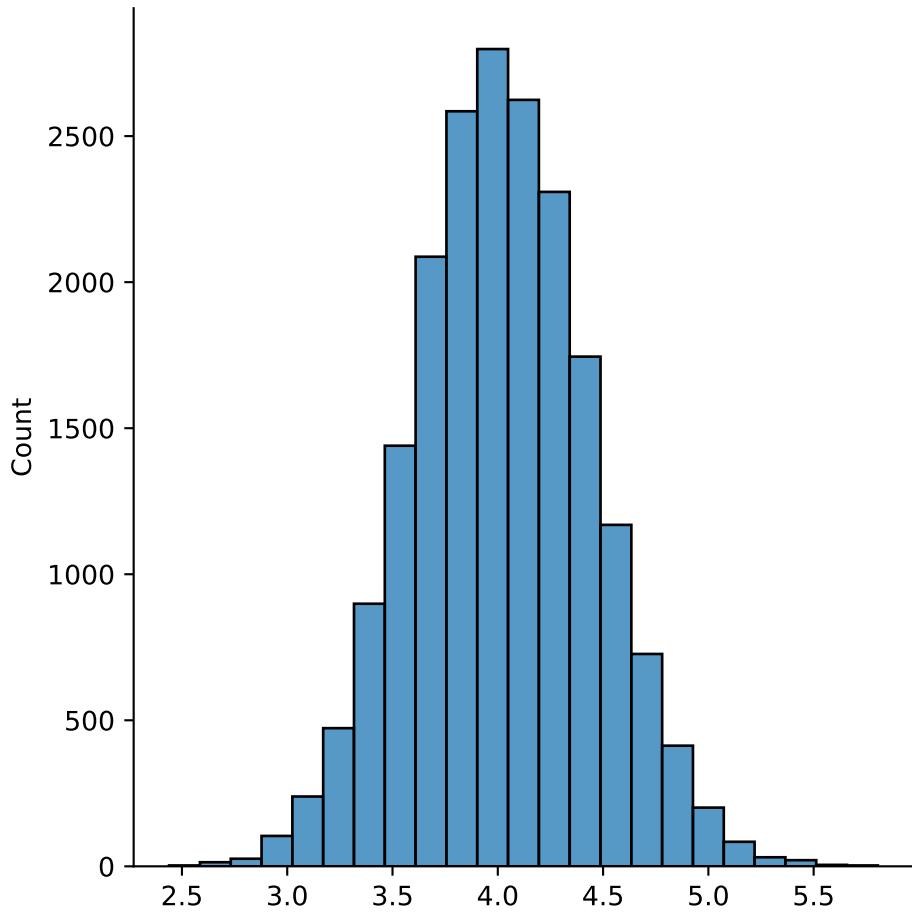


Here is the distribution of  $\bar{k}$  over 20000 ER instances:

```
n, p = 41, 0.1
def average_degree(G):
    return 2*G.number_of_edges() / G.number_of_nodes()

kbar = []
for iter in range(20000):
    ER = nx.erdos_renyi_graph(n, p, seed=iter + 302)
    kbar.append( average_degree(ER) )

sns.distplot( x=kbar, bins=23 );
```



The mean of this distribution converges to  $p(n - 1) = 4$  as the number of random instances goes to infinity.

Theorem 7.1 states that the average degree in a graph with  $n$  nodes and  $m$  edges is  $2m/n$ . According to Theorem 7.2, an ER graph will have the same expected average degree if

$$\frac{2m}{n} = p(n - 1), \quad (7.1)$$

or  $p = 2m/(n^2 - n)$ .

In Example 7.3, for example, we found that our Twitch network has  $n = 7126$  nodes and  $m = 35324$  edges, which suggests an ER equivalent of

$$p = \frac{2(35324)}{(7126)(7125)} \approx 0.001391.$$

That is, if connections in the Twitch network were made completely at random, they would have to occur at a probability of about 0.14%.

### 7.2.2 Watts–Strogatz graphs

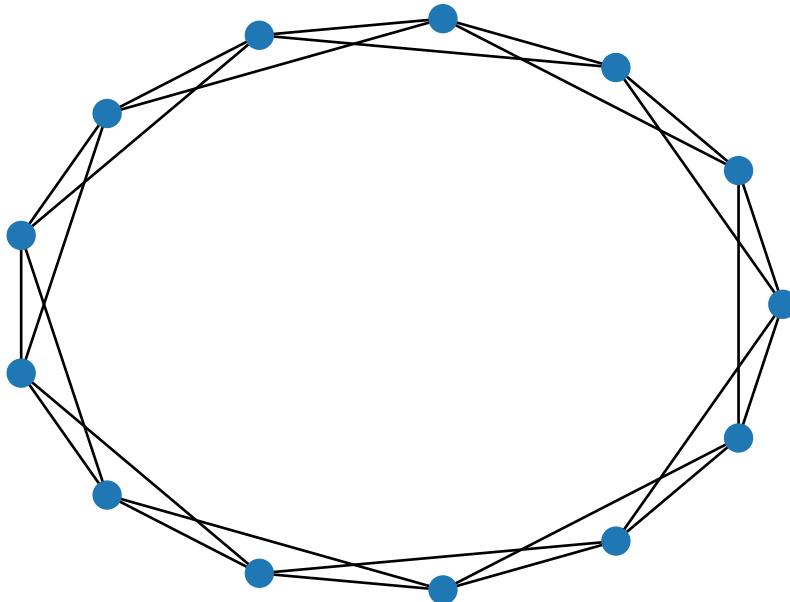
A **Watts–Strogatz graph** (WS graph) is a crude model of human social relationships. A WS graph has three parameters:  $n$ , an even integer  $k$ , and a probability  $q$ .

Imagine  $n$  nodes arranged in a circle. In the first phase, we connect each node with an edge to all of its  $k/2$  left neighbors and  $k/2$  right neighbors. This setup, called a **ring lattice**, represents the idea that people have a local network of friends.

In the next phase, we rewire some of the local relationships into far-flung ones. We visit each node  $i$  in turn. For each edge from  $i$  to a neighbor, with probability  $q$  we replace it with an edge between  $i$  and a node chosen at random from all the nodes  $i$  currently not adjacent to  $i$ .

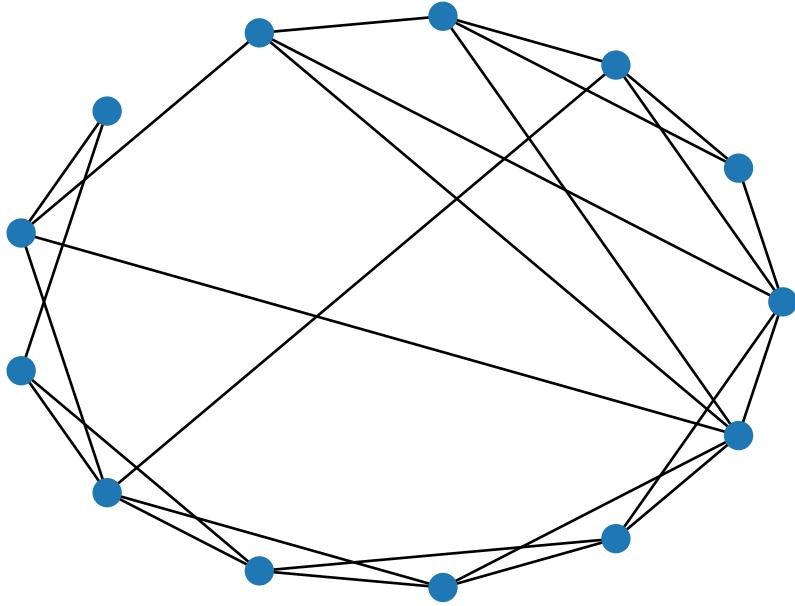
**Example 7.7.** If we set  $q = 0$ , we disable the rewiring phase and get a regular structure:

```
WS = nx.watts_strogatz_graph(13, 4, 0, seed=302)
nx.draw_circular(WS, node_size=100)
```



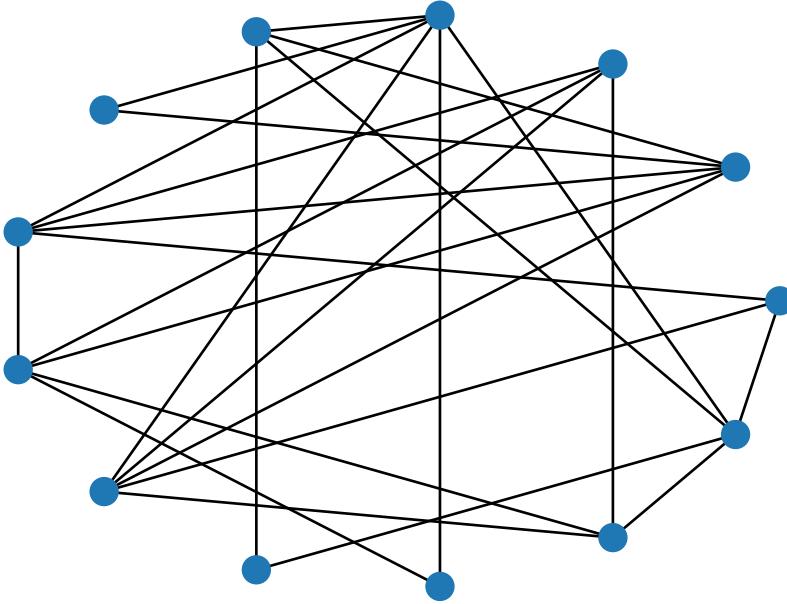
As  $q$  increases, we introduce more randomness:

```
WS = nx.watts_strogatz_graph(13, 4, 0.25, seed=302)
nx.draw_circular(WS, node_size=100)
```



Taken to the limit  $q = 1$ , we end up with something that loses the mostly-local structure:

```
WS = nx.watts_strogatz_graph(13, 4, 1.0, seed=302)
nx.draw_circular(WS, node_size=100)
```



In the initial setup phase, every node in the WS graph has degree equal to  $k$ . The rewiring phase only changes edges—it does not insert nor delete them—so the average degree remains  $k$ .

**Theorem 7.3.** *A WS graph of type  $(n, k, q)$  has average node degree equal to  $k$ .*

Because of this result and Theorem 7.1, we have

$$k = \frac{2m}{n}$$

to get a WS graph of the same average degree as any other graph with  $n$  nodes and  $m$  edges. Keep in mind that we are constrained to use an even integer for  $k$ , so this relation usually can hold only approximately.

## 7.3 Clustering

### i Note

The term *clustering* has a meaning in network analysis that has virtually nothing to do with our earlier use of *clustering* for samples in a feature matrix.

In your own social networks, your friends are probably more likely to be friends with each other than pure randomness would imply. There are various ways to quantify this effect precisely, but here is one of the simplest.

**Definition 7.6.** The **local clustering coefficient** for node  $i$  is

$$C(i) = \frac{2T(i)}{d_i(d_i - 1)},$$

where  $d_i$  is the degree of the node and  $T(i)$  is the number of edges between node  $i$ 's neighbors. If  $d_i = 0$  or  $d_i = 1$ , we set  $C(i) = 0$ .

Equivalently, the value of  $T(i)$  is the number of triangles in the graph that include node  $i$ .

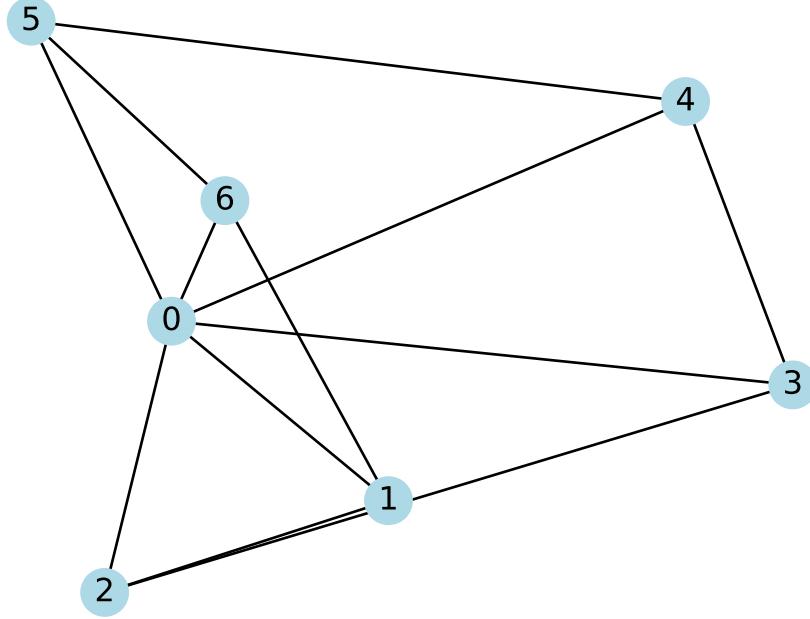
Because the subgraph of the neighbors has

$$\binom{d_i}{2}$$

possible edges, the value of  $C(i)$  is between 0 and 1.

**Example 7.8.** Let's find the clustering coefficient for each node in this wheel graph:

```
W = nx.wheel_graph(7)
nx.draw(W, with_labels=True, node_color="lightblue")
```



Node 0 is adjacent to 6 other nodes, and there are 6 triangles that include it. Thus, its clustering coefficient is

$$C(0) = \frac{6}{6 \cdot 5/2} = \frac{2}{5}.$$

Every other node has 3 neighbors and 2 triangles, so they each have

$$C(i) = \frac{2}{3 \cdot 2/2} = \frac{2}{3}, \quad i \neq 0.$$

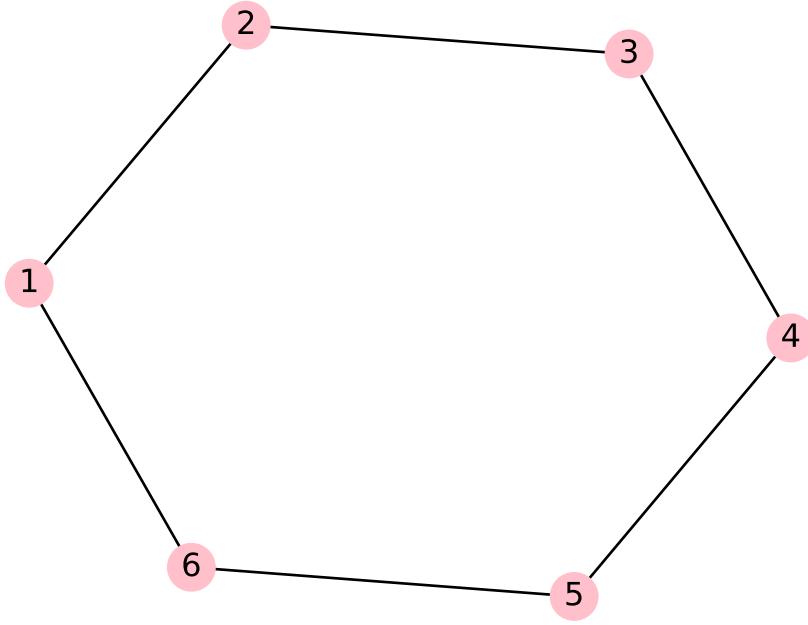
In NetworkX, we can manually count the number of edges among neighbors of node 0 by examining its ego subgraph:

```

nbrhood = W.subgraph( W[0] )      # does not include node 0 itself
print(nbrhood.number_of_edges(), "edges among neighbors of node 0")
nx.draw(nbrhood, with_labels=True, node_color="pink")

```

6 edges among neighbors of node 0



More directly, though, the `clustering` function in NetworkX computes  $C(i)$  for any single node, or for all the nodes in a graph.

```
print("node 0 clustering =", nx.clustering(W,0))
print("\nclustering at each node:")
print( pd.Series(nx.clustering(W), index=W.nodes) )
```

```
node 0 clustering = 0.4
```

```
clustering at each node:
0    0.400000
1    0.666667
2    0.666667
3    0.666667
4    0.666667
5    0.666667
6    0.666667
dtype: float64
```

In addition, the `average_clustering` function will take the average over all nodes of the local clustering values. This is just for convenience when you want a single summary value;

if you already computed the clustering values of all the nodes, then there is little point in using this function to take their mean.

```
print("average clustering =", nx.average_clustering(W))
```

```
average clustering = 0.6285714285714284
```

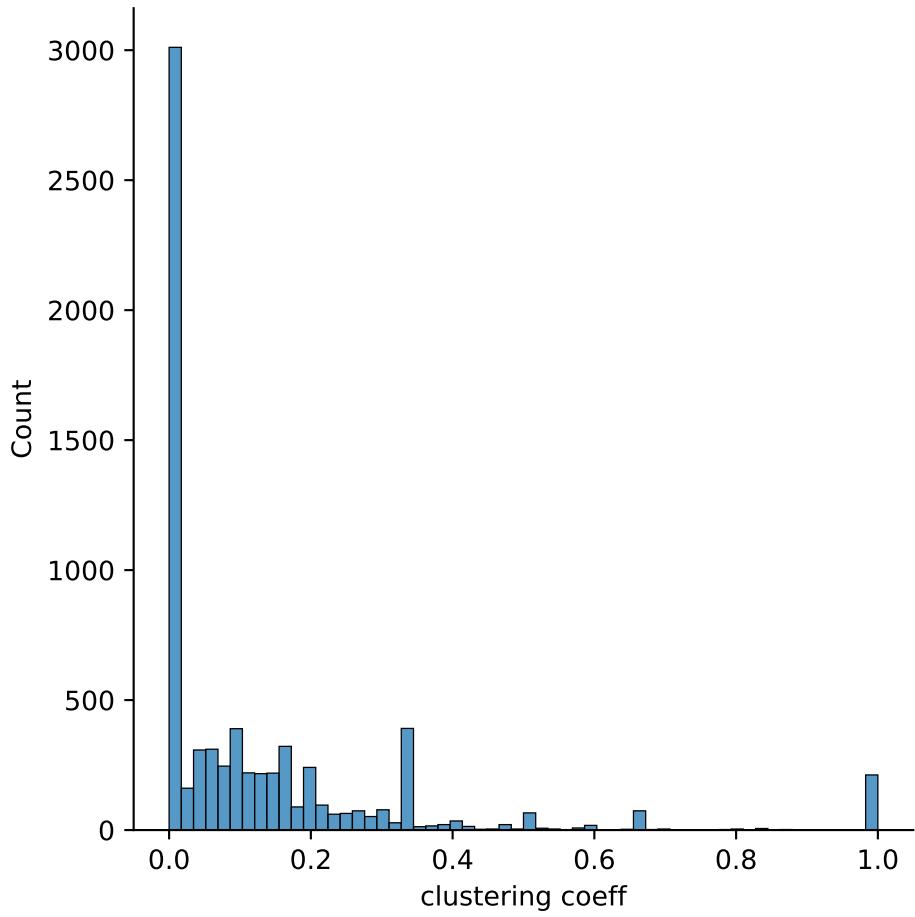
**Example 7.9.** Let's examine clustering within the Twitch network. Here is the distribution of the node cluster values:

```
twitch = nx.read_edgelist("_datasets/musae_edges.csv", delimiter=',', nodetype=int)

n, m = twitch.number_of_nodes(), twitch.number_of_edges()
kbar = 2 * m / n
print(f"{n} nodes, {m} edges, and average degree {kbar:.3f}")

cluster = pd.Series(
    nx.clustering(twitch),
    index=twitch.nodes,
    name="clustering coeff"
)
sns.displot(data=cluster);
```

```
7126 nodes, 35324 edges, and average degree 9.914
```



The average clustering coefficient is:

```
print( f"average Twitch clustering is {cluster.mean():.4f}" )
```

```
average Twitch clustering is 0.1309
```

### 7.3.1 ER graphs

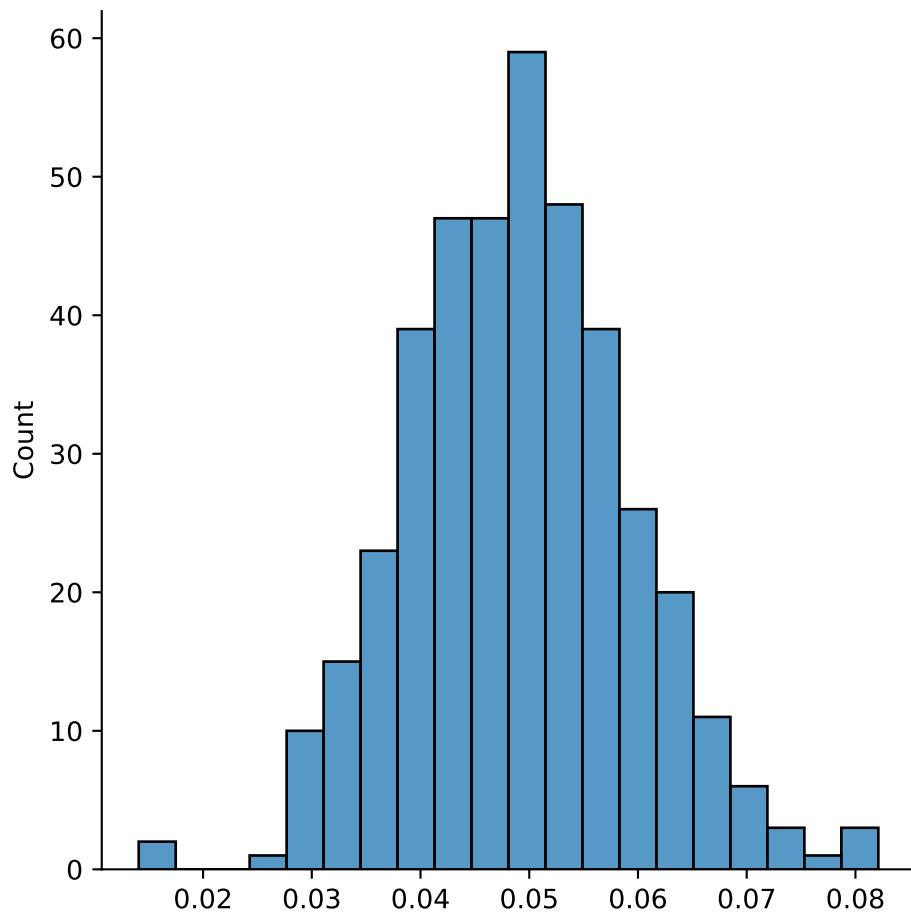
**Theorem 7.4.** *The expected value of the average clustering coefficient in ER graphs of type  $(n, p)$  is  $p$ .*

A formal proof of this theorem is largely superfluous. Considering that each edge in the graph has a probability  $p$  of inclusion, then  $p$  is also the expected fraction of edges that appear within the neighborhood subgraph of any node.

**Example 7.10.** Let's compute average clustering within multiple ER random graphs.

```
n,p = 121,1/20
results = []
for iter in range(400):
    ER = nx.erdos_renyi_graph(n, p, seed=iter+5000)
    results.append( nx.average_clustering(ER) )

sns.displot(x=results);
```



The distribution above can't be normal, because there are hard bounds at 0 and 1, but it looks similar to a normal distribution. The peak is at the value of  $p$  used in the simulation, in accordance with Theorem 7.4.

In Example 7.9, we saw that the Twitch network has  $n = 7126$  and mean clustering equal to about 0.131. If an ER graph is expected to have that same mean clustering, then we

must take  $p = 0.131$ , according to Theorem 7.4. But by Theorem 7.2, this would imply an average node degree of  $p(n-1) \approx 933$ , which is almost 10 times larger than what is actually observed! Thus, clustering strongly suggests that the Twitch network is not much like an ER graph.

### 7.3.2 WS graphs

The initial setup of a WS graph of type  $(n, k, q)$  is highly clustered by design (see Exercise 7.9). If  $q$  is close to zero, the final graph will retain much of this initial clustering. However, the average clustering coefficient tends to decrease as  $q$  increases.

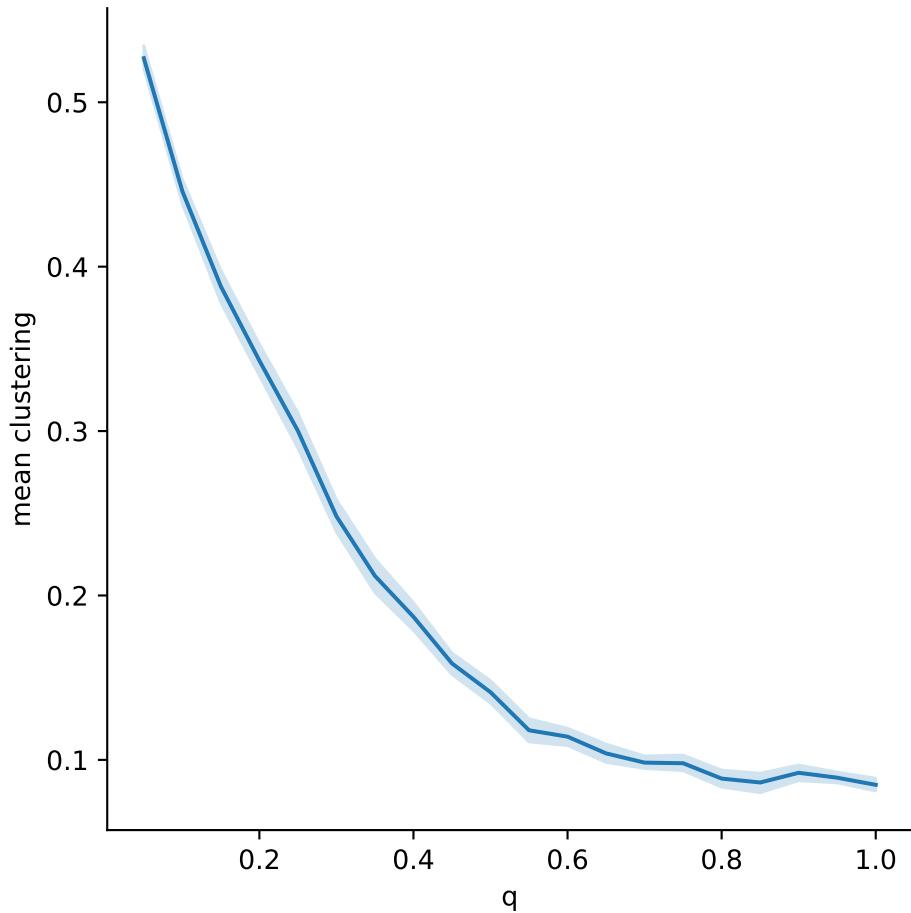
**Example 7.11.** Here is an experiment to observe how the average clustering in a WS graph depends on the rewiring probability  $q$ :

```
n, k = 60, 6
results = []
seed = 302
for q in np.arange(0.05, 1.05, 0.05):
    for iter in range(40):
        WS = nx.watts_strogatz_graph(n, k, q, seed=seed)
        results.append( (q, nx.average_clustering(WS)) )
        seed += 1

results = pd.DataFrame( results, columns=["q", "mean clustering"] )

print("Mean clustering in WS graphs on 60 nodes:")
sns.relplot(data=results,
             x="q", y="mean clustering",
             kind="line")
);
```

Mean clustering in WS graphs on 60 nodes:



Let's scale the experiment above up to the size of the Twitch network. Conveniently, the average degree in that network is roughly 10, which is the value we will use for  $k$  in the WS construction. To save computation time, we use just one WS realization at each value of  $q$ .

```

n, k = twitch.number_of_nodes(), 10
seed = 19716
results = []
for q in np.arange(0.15, 0.51, 0.05):
    WS = nx.watts_strogatz_graph(n, k, q, seed=seed)
    results.append( (q, nx.average_clustering(WS)) )
    seed += 1

print( pd.DataFrame( results, columns=["q", "mean clustering"] ) )

q  mean clustering

```

0	0.15	0.415477
1	0.20	0.347262
2	0.25	0.286947
3	0.30	0.232891
4	0.35	0.184559
5	0.40	0.146123
6	0.45	0.112606
7	0.50	0.085669

The mean clustering for the Twitch network was found to be about 0.131 in Example 7.9. From the above, it appears that the WS graphs will have a similar mean clustering at around  $q = 0.42$ . Let's freeze  $q$  there and try more WS realizations to check:

```
seed = 3383
n, k, q = twitch.number_of_nodes(), 10, 0.42
cbar = []
for iter in range(10):
    WS = nx.watts_strogatz_graph(n, k, q, seed=seed)
    cbar.append(nx.average_clustering(WS))
    seed += 10
print(f"avg WS clustering at q = 0.42 is {np.mean(cbar):.4f}")
```

avg WS clustering at q = 0.42 is 0.1327

So far, the WS construction gives a plausible way to reconstruct the clustering observed in the Twitch network. However, there are many other graph properties left to examine!

## 7.4 Distance

The *small-world phenomenon* is, broadly speaking, the observation that any two people in a large group can be connected by a surprisingly short path of acquaintances. This concept appears, for instance, in the *Bacon number game*, where actors are nodes, appearing in the same movie creates an edge between them, and one tries to find the distance between Kevin Bacon and some other designated actor.

**Definition 7.7.** A **path** from node  $x$  to node  $y$  is a sequence of edges in the graph, either the singleton sequence  $(x, y)$ , or

$$(x, z_1), (z_1, z_2) (z_2, z_3) \dots, (z_{t-1}, z_t) (z_t, y),$$

for some  $t \geq 1$ . The path above has **length**  $t + 1$ , which is the number of edges in the path. The **distance** between two nodes in a graph is the length of the shortest path between them. The maximum distance over all pairs of nodes in a graph is called its **diameter**.

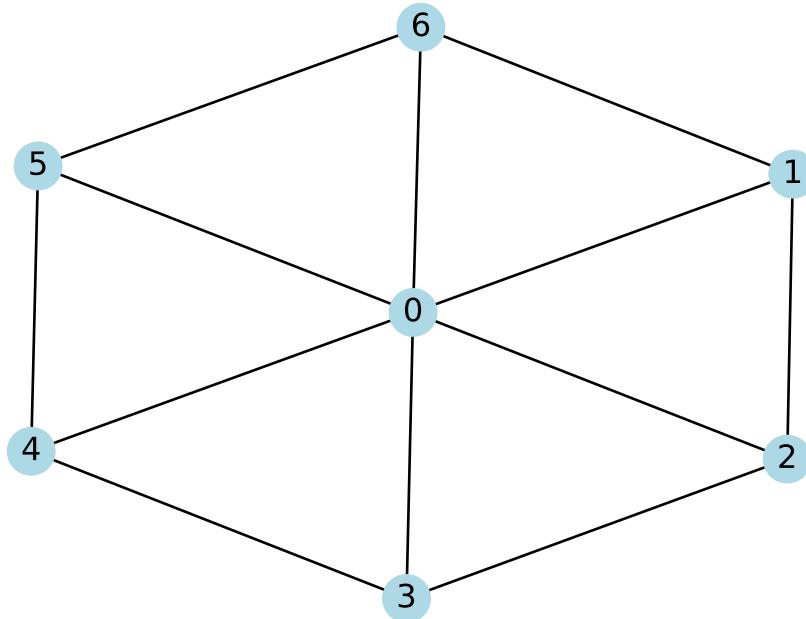
For example, in a complete graph, the distance between any pair of distinct nodes is 1, since all possible pairs are connected by an edge.

Because the diameter depends on the extreme outlier of the distance distribution, it's often more useful to consider the **average distance** by taking the mean. In practice, computing pairwise distances can be computationally expensive, so the average distance might be estimated by taking a sample of all possible pairs.

It could happen that there is a pair of nodes that is not connected by any path of edges. In that case, the distance between them is either undefined or infinite, and we say the graph is **disconnected**.

**Example 7.12.** Here is a small wheel graph:

```
W = nx.wheel_graph(7)
nx.draw(W, with_labels=True, node_color="lightblue")
```



No node is more than two hops away from another, so the diameter of this graph is 2. This graph has so few nodes that we can easily compute the entire matrix of pairwise distances:

```
distance = dict( nx.all_pairs_shortest_path_length(W) )
D = np.array( [ [distance[i][j] for j in W.nodes] for i in W.nodes] )
```

```
print(D)

[[0 1 1 1 1 1]
 [1 0 1 2 2 2]
 [1 1 0 1 2 2]
 [1 2 1 0 1 2]
 [1 2 2 1 0 1]
 [1 2 2 2 1 0]
 [1 1 2 2 2 1]]
```

If all we want is the average distance, though, there is a convenience function for computing it directly:

```
mean_dist = nx.average_shortest_path_length(W)
print(f"The average distance is {mean_dist:.5f}")
```

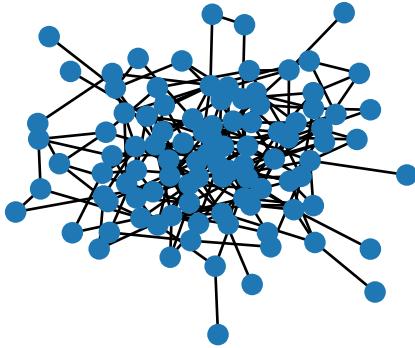
The average distance is 1.42857

It does become quite slow as the number of nodes grows.

### 7.4.1 ER graphs

If we want to compute distances within ER random graphs, we quickly run into a problem, because an ER graph may be disconnected. NetworkX gives an error if we try to compute the average distance in a disconnected graph.

```
n, p = 101, 1/25
ER = nx.erdos_renyi_graph(n, p, seed=0)
nx.draw(ER, node_size=50)
```



```
nx.average_shortest_path_length(ER)
```

NetworkXError: Graph is not connected.

One way to cope with this eventuality is to decompose the graph into **connected components**, a disjoint separation of the nodes into connected subgraphs. We can use `nx.connected_components` to get node sets for each component:

```
[ len(cc) for cc in nx.connected_components(ER) ]
```

[100, 1]

The result above tells us that removing the lone unconnected node in the ER graph leaves us with a connected component. We can always get the largest component with the following idiom:

```
ER_sub = ER.subgraph( max(nx.connected_components(ER), key=len) )
print(ER_sub.number_of_nodes(), "nodes in largest component")
```

```
100 nodes in largest component
```

We can then use this largest component as a proxy for the full graph:

```
nx.average_shortest_path_length(ER_sub)
```

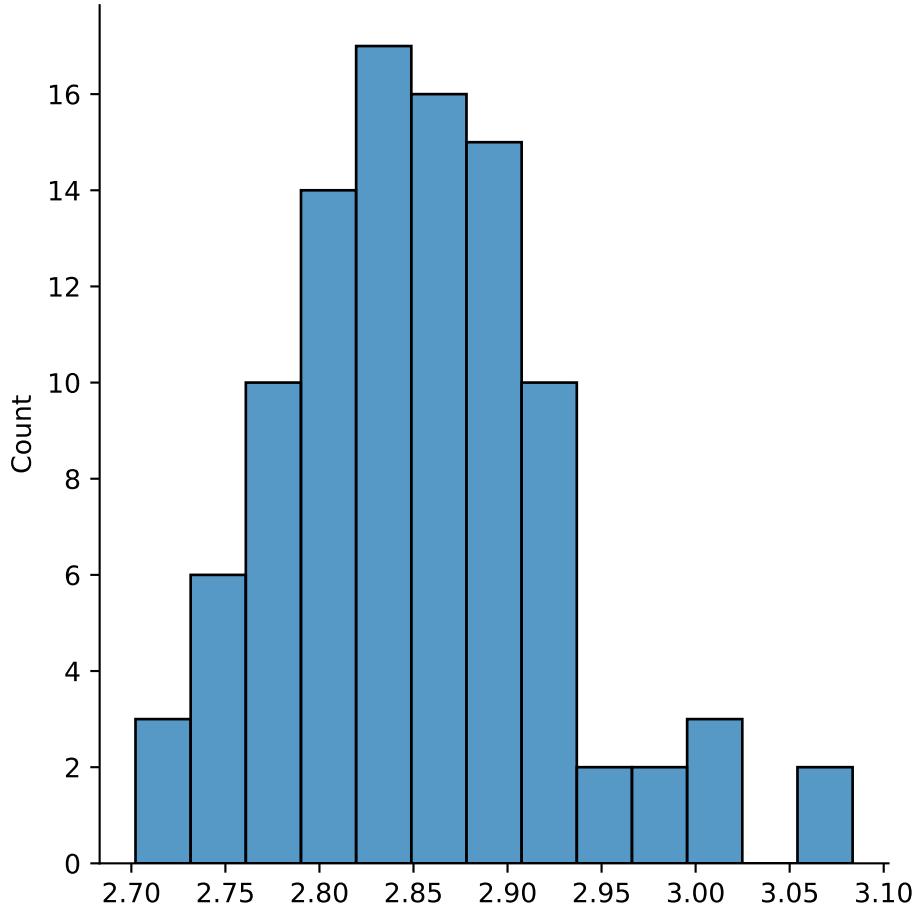
```
3.30828282828283
```

**Example 7.13.** Let's examine average distances within ER graphs of a fixed type.

```
n, p = 121, 1/20
dbar = []
for iter in range(100):
    ER = nx.erdos_renyi_graph(n, p, seed=iter+5000)
    ER_sub = ER.subgraph( max(nx.connected_components(ER), key=len) )
    dbar.append( nx.average_shortest_path_length(ER_sub) )

print("average distance in the big component of ER graphs:")
sns.displot(x=dbar, bins=13);
```

```
average distance in the big component of ER graphs:
```



The chances are good, therefore, that any message could be passed along in three hops or fewer within the big component.

Theory states that as  $n \rightarrow \infty$ , the mean distance in ER graphs is expected to be approximately

$$\frac{\ln(n)}{\ln(\bar{k})}. \quad (7.2)$$

For Example 7.13, we have  $n = 121$  and  $\bar{k} = 6$ , which gives about 2.68 in this formula.

#### 7.4.2 WS graphs

The Watts–Strogatz model was originally proposed to demonstrate small-world networks. The initial ring-lattice structure of the construction exhibits both large clustering and large mean distance:

```

# q=0 ==> initial ring lattice
G = nx.watts_strogatz_graph(400, 6, 0)
C0 = nx.average_clustering(G)
L0 = nx.average_shortest_path_length(G)
print(f"Ring lattice has average clustering {C0:.4f}")
print(f"and average shortest path length {L0:.2f}")

```

Ring lattice has average clustering 0.6000  
and average shortest path length 33.75

At the other extreme of  $q = 1$ , we get small clustering and small average distance. The most interesting aspect of WS graphs is the transition between these extremes as  $q$  varies.

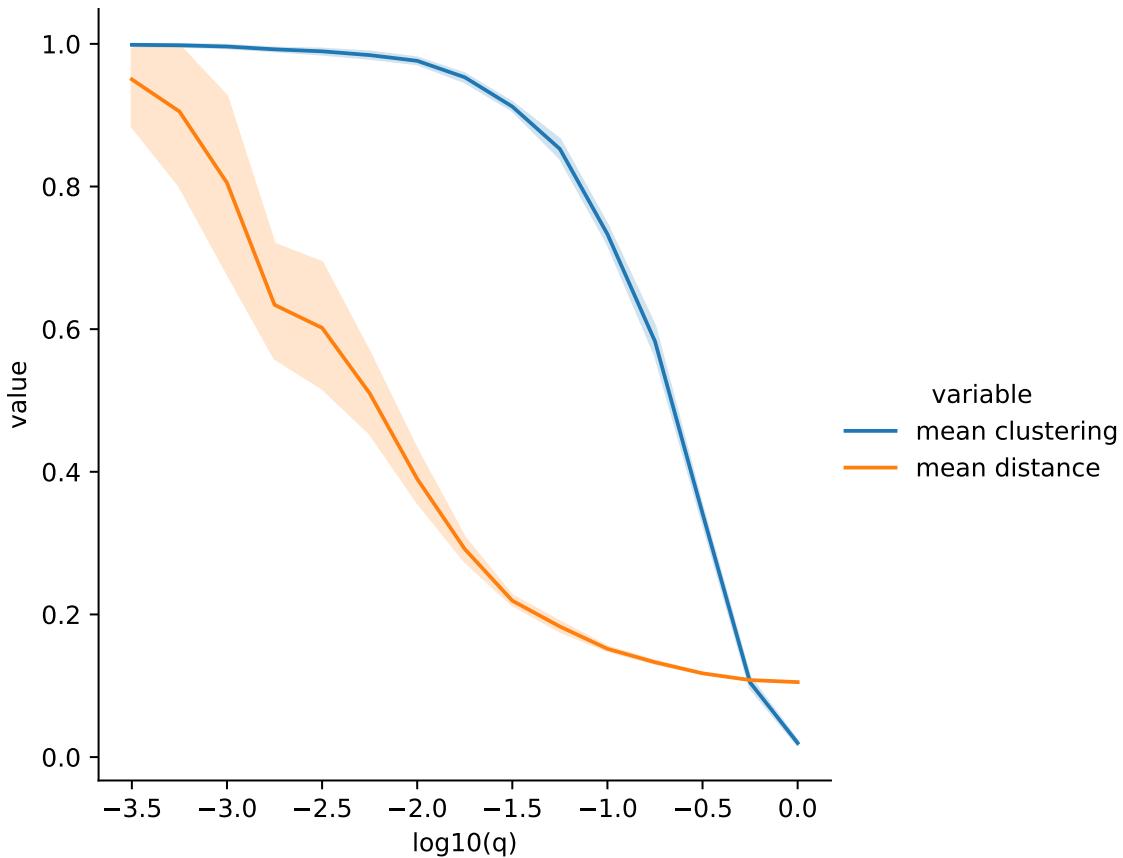
```

results = []
for lq in np.arange(-3.5, 0.01, 0.25):
    for iter in range(8):
        G = nx.watts_strogatz_graph(400, 6, 10**lq, seed=302 + 10*iter)
        C = nx.average_clustering(G) / C0
        L = nx.average_shortest_path_length(G) / L0
        results.append( (lq, C, L) )

results = pd.DataFrame( results,
    columns=[ "log10(q)", "mean clustering", "mean distance" ]
)

sns.relplot(data=pd.melt(results, id_vars="log10(q)") ,
    x="log10(q)", y="value",
    hue="variable", kind="line"
);

```



The  $y$ -axis shows the average clustering and distance relative to their values at  $q = 0$ , computed above as  $C_0$  and  $L_0$ . Watts and Strogatz raised awareness of the fact that for quite small values of  $q$ , i.e., relatively few nonlocal connections, there are networks with both large clustering and small average distance—in short, the small-world effect.

### 7.4.3 Twitch network

Now we consider distances within the Twitch network.

```
twitch = nx.read_edgelist("_datasets/musae_edges.csv", delimiter=',', nodetype=int)
n, e = twitch.number_of_nodes(), twitch.number_of_edges()
kbar = 2*e/n
print(n, "nodes and", e, "edges")
print(f"average degree is {kbar:.3f}")
```

7126 nodes and 35324 edges

```
average degree is 9.914
```

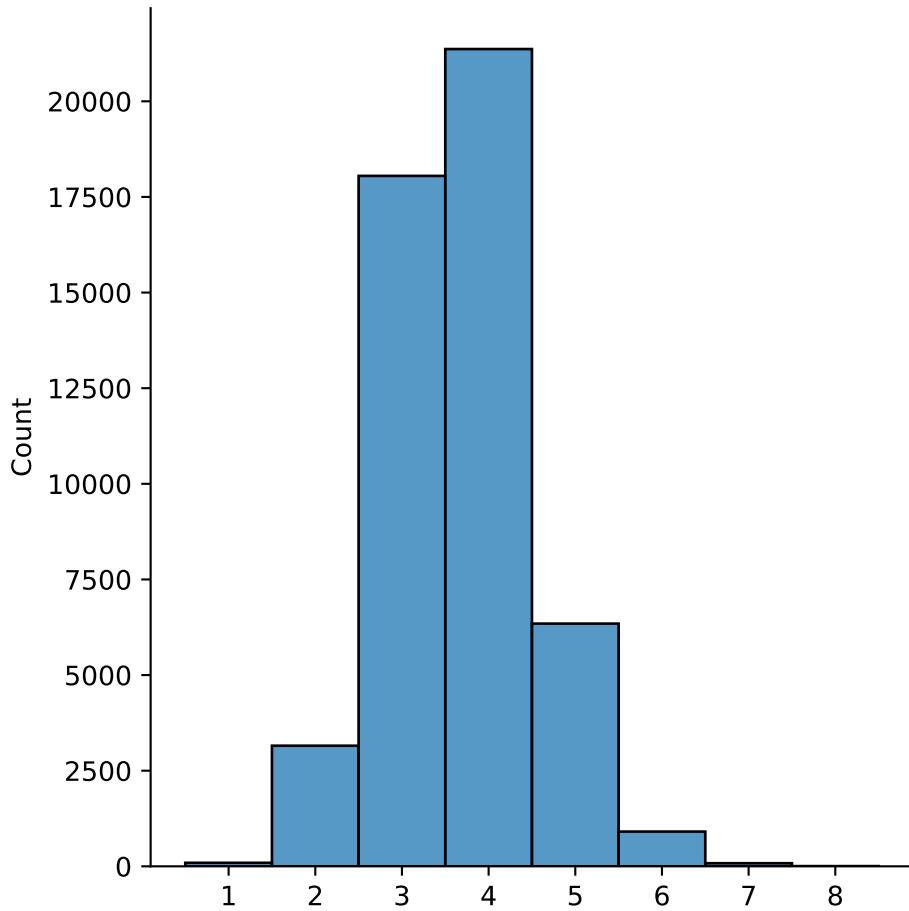
Computing the distances between all pairs of nodes in this graph would take a long time, so we sample pairs randomly to get an estimate.

```
rng = default_rng(19716)

# Compute the distance between a random pair of distinct nodes:
def pairdist(G):
    n = nx.number_of_nodes(G)
    i = j = rng.integers(0,n)
    while i==j: j=rng.integers(0,n)    # get distinct nodes
    return nx.shortest_path_length(G,source=i,target=j)

distances = [ pairdist(twitch) for _ in range(50000) ]
print("Pairwise distances in Twitch graph:")
sns.displot(x=distances, discrete=True)
print( "estimated mean =", np.mean(distances) )
```

```
Pairwise distances in Twitch graph:
estimated mean = 3.67584
```



Next we explore WS graphs with the same  $n$  as the Twitch network and  $k = 10$  to get a similar average degree.

```

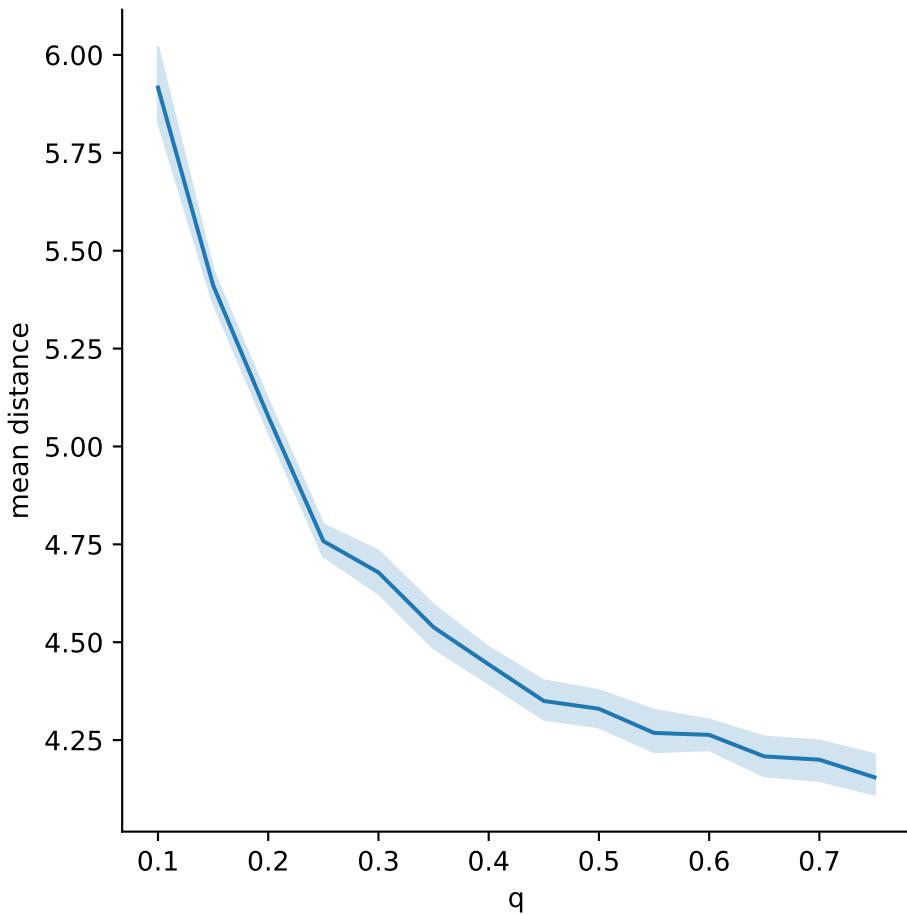
results = []
seed = 44044
n, k = twitch.number_of_nodes(), 10
for q in np.arange(0.1, 0.76, 0.05):
    for iter in range(10):
        WS = nx.watts_strogatz_graph(n, k, q, seed=seed)
        dbar = sum(pairdist(WS) for _ in range(60))/60
        results.append( (q,dbar) )
        seed += 7

results = pd.DataFrame( results, columns=["q", "mean distance"] )
print("Pairwise distances in WS graphs:")

```

```
sns.relplot(data=results, x="q", y="mean distance", kind="line");
```

Pairwise distances in WS graphs:



The decrease with  $q$  here is less pronounced than we saw for the smaller WS graphs in Section 7.4.2.

In Example 7.11, we found that  $q = 0.42$  reproduces the same average clustering as in the Twitch network. In the graph above, that corresponds to a mean distance of about 4.5, which is a bit above the estimated Twitch mean distance of 3.7, but not dramatically so. Thus, Watts-Strogatz could still be considered a plausible model for the Twitch network. Or to put it another way, the Twitch network seems to be comparable to a small-world network model.

In the next section, however, we will see that there are major differences in another respect.

## 7.5 Degree distributions

As we well know, means of distributions do not always tell the entire story. For example, the distribution of the degrees of all the nodes in our Twitch network has some surprising features:

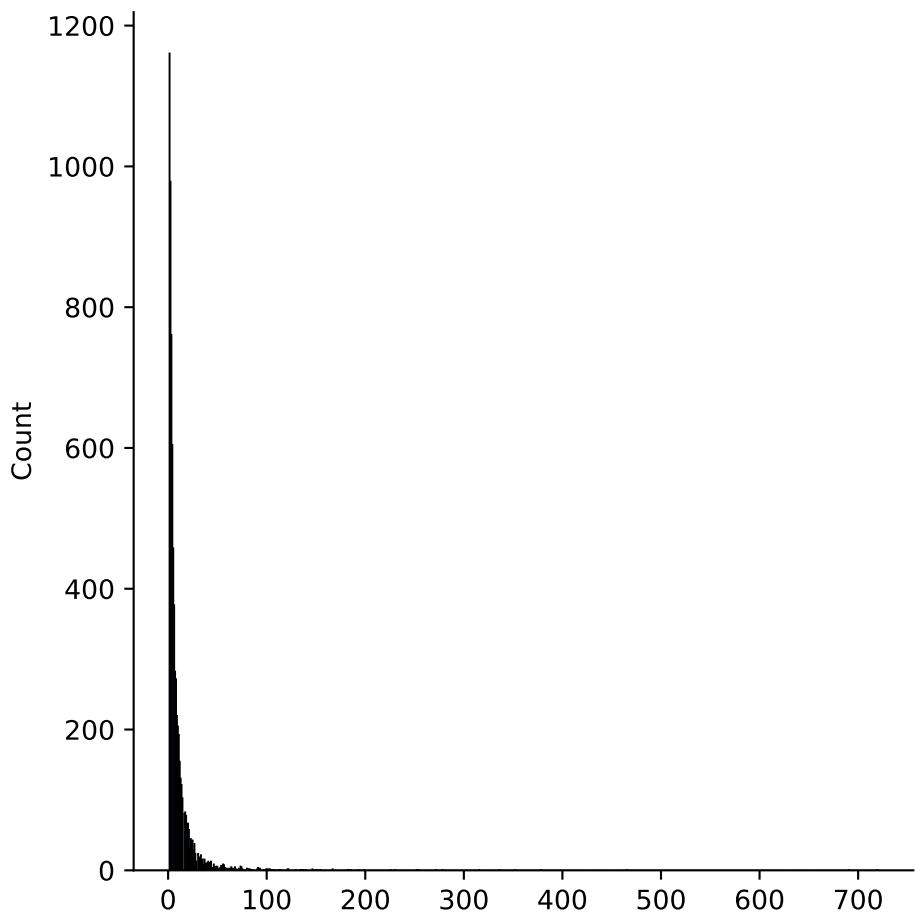
```
twitch = nx.read_edgelist("_datasets/musae_edges.csv", delimiter=',', nodetype=int)
twitch_degrees = pd.Series( dict(twitch.degree), index=twitch.nodes )
twitch_degrees.describe()
```

	0
count	7126.000000
mean	9.914117
std	22.190263
min	1.000000
25%	2.000000
50%	5.000000
75%	11.000000
max	720.000000

Observe above that that there is a significant disparity between the mean and median values of the degree distribution, and that the standard deviation is much larger than the mean. A histogram plot confirms that the degree distribution is widely dispersed:

```
print("Twitch network degree distribution:")
sns.displot(twitch_degrees);
```

Twitch network degree distribution:



A few nodes in the network have hundreds of friends:

```
friend_counts = twitch_degrees.value_counts() # histogram heights
friend_counts.sort_index(ascending=False).head(10)
```

	0
720	1
691	1
465	1
378	1
352	1
336	1
316	1
278	1
272	1
254	1

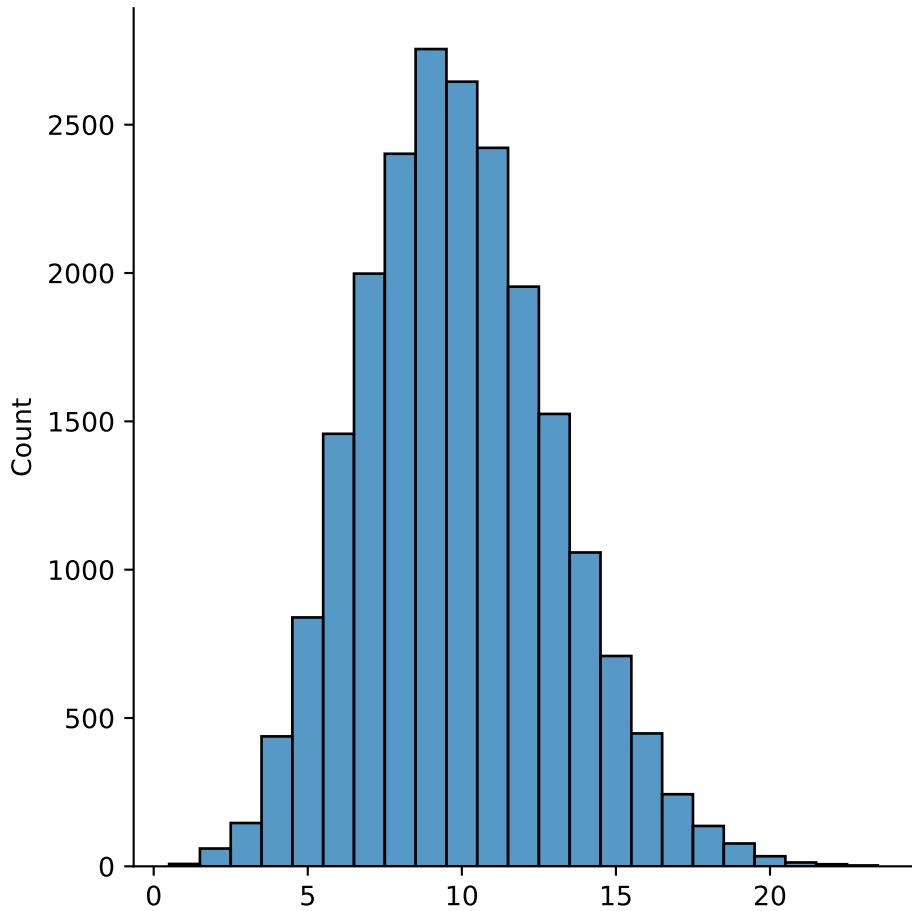
These “gregarious nodes” or *hubs* are characteristic of many social and other real-world networks.

We can compare the above distribution to that in a collection of ER graphs with the same size and expected average degree:

```
n, e = twitch.number_of_nodes(), twitch.number_of_edges()
kbar = 2*e/n
p = kbar/(n-1)
degrees = []
for iter in range(3):
    ER = nx.erdos_renyi_graph(n, p, seed=111+iter)
    degrees.extend( [ER.degree(i) for i in ER.nodes] )

print("ER graphs degree distribution:")
sns.displot(degrees, discrete=True);
```

ER graphs degree distribution:



Theory proves that the plot above converges to a *binomial distribution*. This is yet another indicator that the ER model does not explain the Twitch network well. A WS graph has a similar distribution:

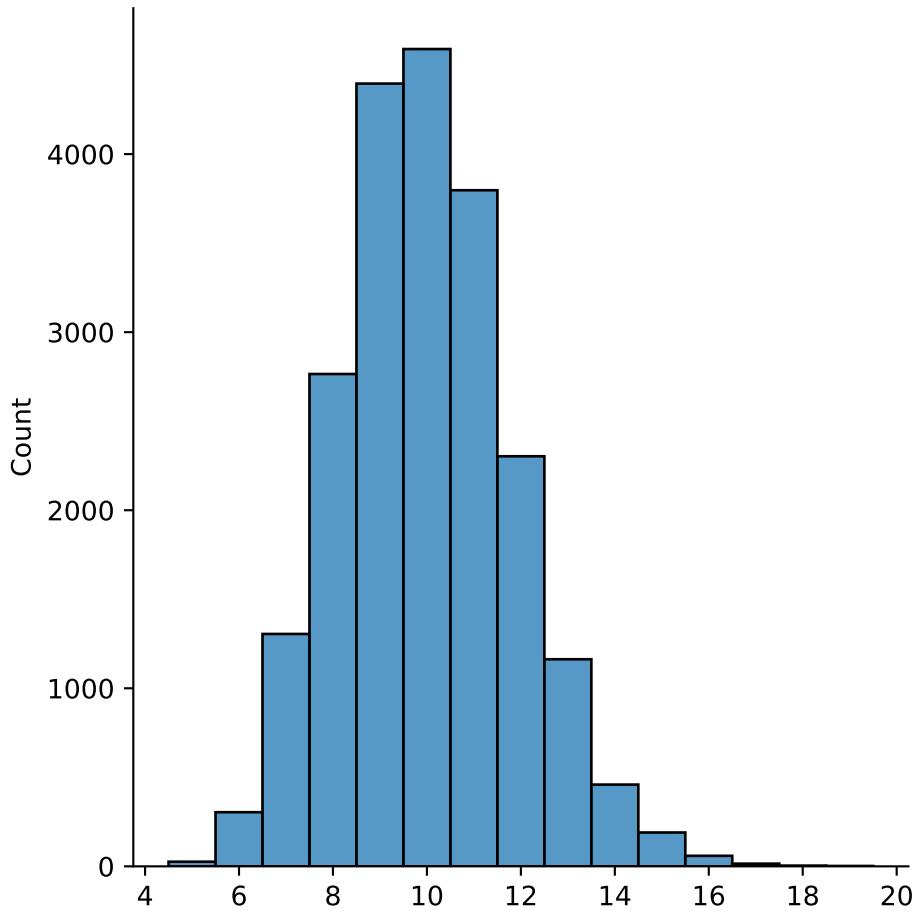
```

k,q = 10, 0.42
degrees = []
for iter in range(3):
    WS = nx.watts_strogatz_graph(n, k, q, seed=222+iter)
    degrees.extend( [WS.degree(i) for i in WS.nodes] )

print("WS graphs degree distribution:")
sns.displot(degrees, discrete=True);

```

WS graphs degree distribution:

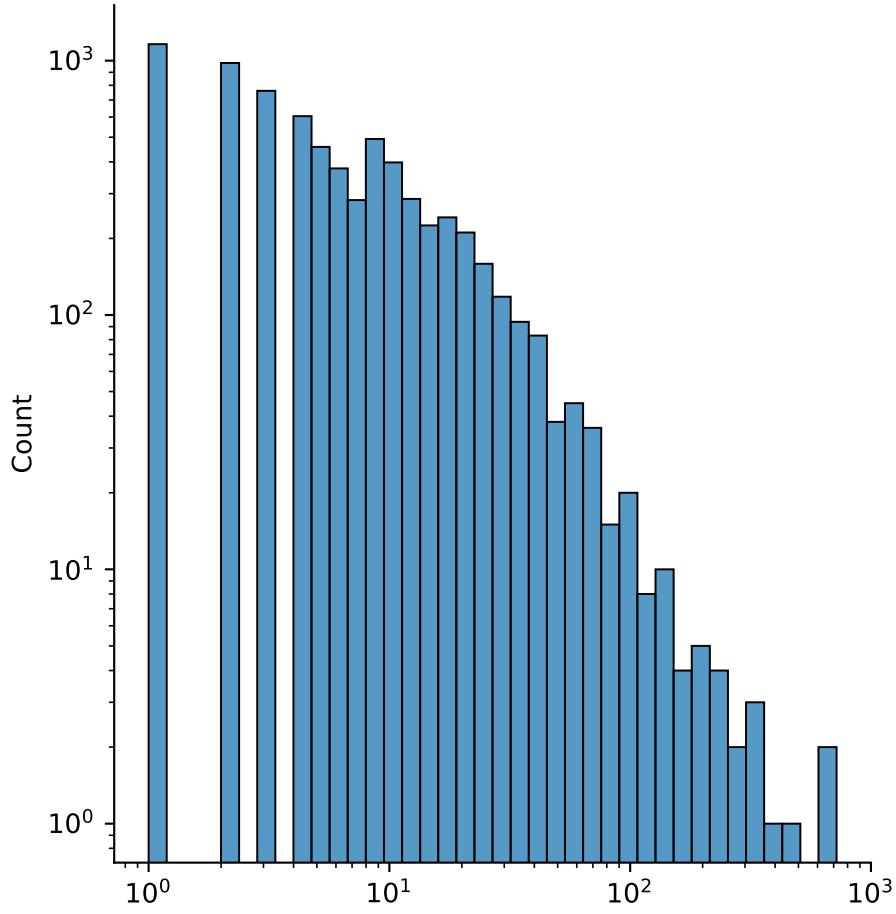


Neither ER nor WS graphs show degree distributions even remotely like that of the Twitch network.

### 7.5.1 Power-law distribution

The behavior of the Twitch degree distribution gets very interesting when the axes are transformed to use log scales:

```
hist = sns.displot(data=twitch_degrees, log_scale=True)
hist.axes[0,0].set_yscale("log")
```



For degrees between 10 and several hundred, the counts lie nearly on a straight line. That is, if  $x$  is degree and  $y$  is the node count at that degree, then

$$\log(y) \approx -a \cdot \log(x) + b,$$

i.e.,

$$y \approx Bx^{-a},$$

for some  $a > 0$ . This relationship is known as a **power law**, a relationship often seen in physics. Many social networks seem to follow a power-law distribution of node degrees, to some extent. (The precise extent is a subject of heated debate.) The decay of  $x^{-a}$  to zero as  $x \rightarrow \infty$  is much slower than the normal distribution's  $e^{-x^2/2}$ , for example, and we could say that the power-law network has a *heavy-tailed distribution*.

We can get a fair estimate of the constants  $B$  and  $a$  in the power law by doing a least-squares fit on the logs of  $x$  and  $y$ . First, we need the counts:

```

y = twitch_degrees.value_counts()
counts = pd.DataFrame( {"degree": y.index, "count": y.values} )
counts = counts[ (counts["degree"] > 10) & (counts["degree"] < 200) ];
counts.head(6)

```

	degree	count
10	11	193
11	12	155
12	13	131
13	14	122
14	15	103
15	17	83

Now we will get additional columns by log transformations. (Note: the `np.log` function is the natural logarithm.)

```
logcounts = counts.transform(np.log)
```

Now we use `sklearn` for a linear regression.

```

from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(logcounts[["degree"]], logcounts["count"])
lm.coef_[0], lm.intercept_

```

```
(-1.9941272617745713, 9.7094067609447)
```

The first value, which is both the slope of the line and the exponent of  $x$  in the power law, is the most interesting part. Hence, we conclude that the degree counts vary like  $Bx^{-2}$  over a wide range of degrees.

### 7.5.2 Barabási–Albert graphs

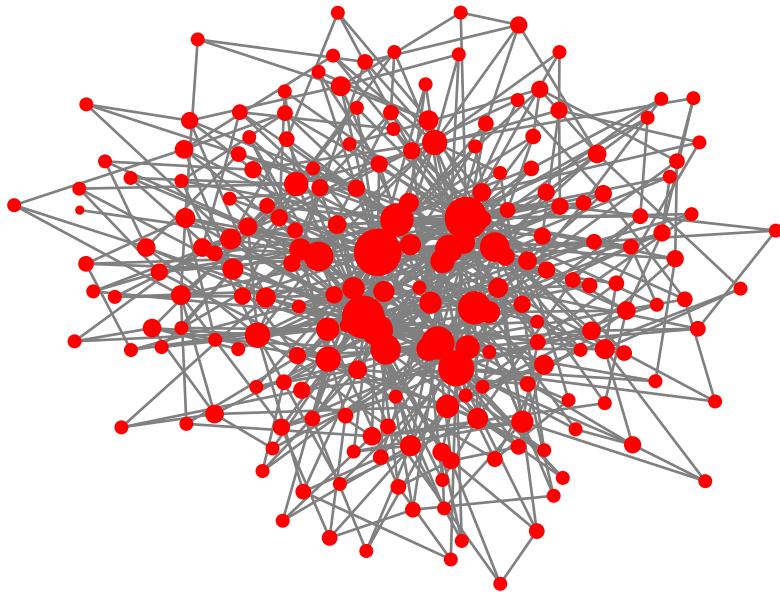
A random **Barabási–Albert** graph (BA graph) is a standard way to model power-law networks.

The construction starts with a small seed graph. One by one, new nodes are added with  $m$  edges connecting it to existing nodes. The new edges are chosen randomly, but higher probability is given for connections to nodes that already have higher degree (i.e., are more “popular”). This is a the-rich-get-richer concept known as *preferential attachment*.

Because of preferential attachment, there is a strong tendency to develop a hubs of very high degree.

**Example 7.14.** Here is a BA graph with 200 nodes and  $m = 3$  connections per new node:

```
BA = nx.barabasi_albert_graph(200, 3, seed=302)
BA_degrees = pd.Series( dict(BA.degree), index=BA.nodes )
nx.draw(
    BA,
    node_size=6 * BA_degrees,
    node_color="red", edge_color="gray"
)
```



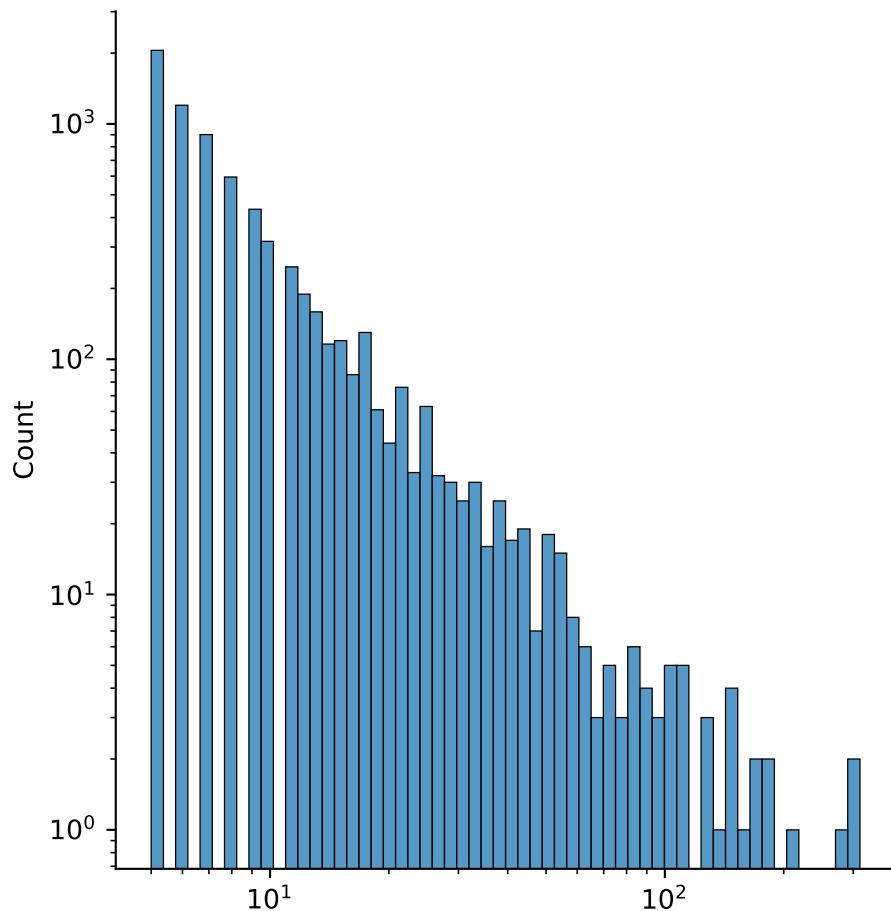
The size of each node is proportional to its degree. In the default rendering, the large hubs crowd into the center of the picture, while the lowest-degree nodes are pushed to the periphery.

Probability theory predicts that a BA graph has a power-law degree distribution with exponent  $-3$  as  $n \rightarrow \infty$ .

Except for the initial seed stage, each node of a BA graph introduces  $m$  new edges, for a total edge count of about  $mn$ . By Theorem 7.1, the average degree of the graph is roughly  $\bar{k} = 2mn/n = 2m$ .

**Example 7.15.** For the Twitch network,  $\bar{k}$  is almost 10, so we use  $m = 5$  to get a comparable BA graph. Here we construct such a graph with the same number of nodes as the Twitch network.

```
m = round(kbar/2)
BA = nx.barabasi_albert_graph(n, m, seed=5)
BA_degrees = pd.Series( dict(BA.degree), index=BA.nodes )
hist = sns.displot(data=BA_degrees, log_scale=True)
hist.axes[0,0].set_yscale("log")
```



The straight-line relationship on log-log scales is the hallmark of a power law. As with the Twitch network, we can estimate the exponent by a linear regression:

```
y = BA_degrees.value_counts()
counts = pd.DataFrame( {"degree":y.index, "count":y.values} )
```

```

counts = counts[ (counts["degree"] > 5) & (counts["degree"] < 80) ]
logcounts = counts.transform(np.log)
lm.fit( logcounts[["degree"]], logcounts["count"] )
print( "exponent of power law:", lm.coef_[0] )

```

exponent of power law: -2.873136852062997

While BA graphs capture the heavy-tailed degree distribution of the Twitch network rather well, they are way off when it comes to clustering. For the Twitch network, we have:

```

print( "Mean clustering in Twitch graph:", nx.average_clustering(twitch) )

```

Mean clustering in Twitch graph: 0.13092821901472096

And for BA, we get:

```

cbar = []
seed = 59
for iter in range(20):
    BA = nx.barabasi_albert_graph(n, m, seed=seed)
    cbar.append( nx.average_clustering(BA) )
    seed += 1

print( "Mean clustering in BA graphs:", np.mean(cbar) )

```

Mean clustering in BA graphs: 0.009219743245252128

The Twitch network has characteristics of both small-world (high clustering with small distance) and power-law (superstar hub nodes) networks. Possibly, some combination of the WS and BA constructions would get closer to the Twitch network than either one does alone.

## 7.6 Centrality

In some applications, we want to know which nodes of a network are the most important in some sense. For instance, we might want to find the most informative website, the most influential members of a social network, or nodes that are critical for efficient transfer within the network. These traits go under the general name of **centrality**.

### 7.6.1 Degree centrality

A clear candidate for measuring the centrality of a node is its degree.

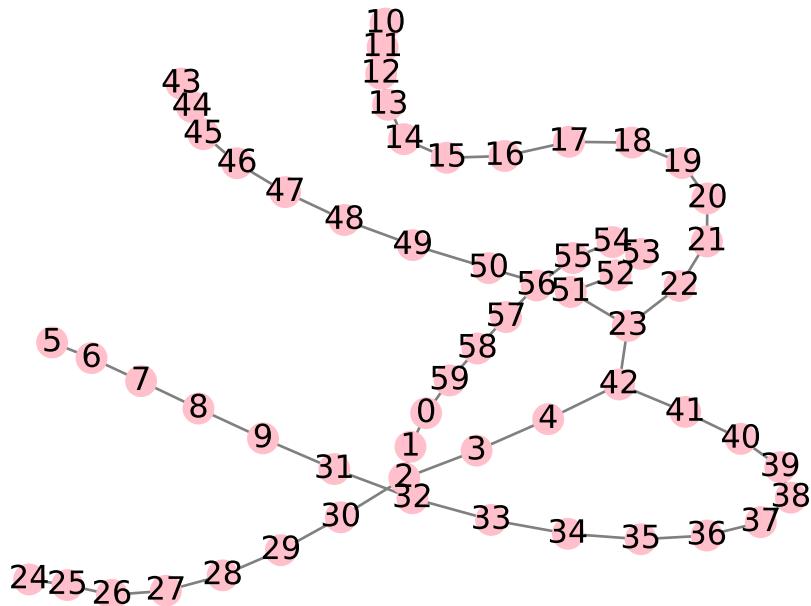
**Definition 7.8.** The **degree centrality** of a node is its degree, divided by the maximum degree over all nodes in the graph.

NetworkX has a function `degree_centrality` to compute this quantity.

While degree centrality yields useful information in some networks, it is not always a reliable measuring stick. For example, a search engine for the Web that uses degree centrality would be easily fooled by creating a large collection of sites that all link to each other, even if few outside sources also link to the collection. Other aspects of network layout also argue against relying on degree centrality.

**Example 7.16.** Consider the following Watts–Strogatz graph:

```
G = nx.watts_strogatz_graph(60, 2, .1, seed=6)
pos = nx.spring_layout(G, seed=1)
style = dict(pos=pos, edge_color="gray", node_color="pink", with_labels=True)
nx.draw(G, **style, node_size=120)
```



There is little variation in the degrees of the nodes. In fact, there are only 3 unique values of the degree centrality. From the rendering of the graph, though, it's clear that the nodes with degree equal to 2 are not all roughly equivalent. If we remove node 6, for instance, then node 5 is the only other node that will be affected. But if we remove node 22, we cut off a large branch from the rest of the network.

### 7.6.2 Betweenness centrality

A different way to measure centrality is to consider the paths between nodes. If  $i$  and  $j$  are nodes in a graph, then there is at least one path between them whose length is the shortest possible, that is, whose length is the distance between  $i$  and  $j$ . If there is only one such path, then removing it would increase the distance between these nodes. But if there are multiple paths of shortest length between the nodes, then the graph has more redundancy or connectedness built-in.

**Definition 7.9.** For a graph on  $n$  nodes, let  $\sigma(i, j)$  denote the number of shortest paths between nodes  $i$  and  $j$ . For any other node  $k$ , let  $\sigma(i, j|k)$  be the number of such shortest paths that pass through node  $k$ . Then the **betweenness centrality** of node  $k$  is

$$c_B(k) = \frac{1}{\binom{n-1}{2}} \sum_{\substack{\text{all pairs } i,j \\ i \neq k, j \neq k}} \frac{\sigma(i, j|k)}{\sigma(i, j)}.$$

Each term in the sum is less than or equal to 1, and the number of terms in the sum is  $\binom{n-1}{2}$ , so  $0 \leq c_B(k) \leq 1$  for any node, and  $c_B(k) = 0$  only if node  $k$  has degree zero; i. e., is disconnected from the graph.

**Example 7.17.** We will find the betweenness centrality of the following *barbell graph*:

Let's begin with node  $k = 3$ , in the middle. We can divide the nodes into the right-side triangle  $\{0, 1, 2\}$  and the left-side triangle  $\{4, 5, 6\}$ . If we take both  $i$  and  $j$  from a single triangle, then no shortest paths between  $i$  and  $j$  go through node 3. Thus, these pairs contribute nothing to the sum in the formula for  $c_B(3)$ . On the other hand, if  $i$  and  $j$  are in different triangles, then there is a unique shortest path between them that does pass through node 3, and  $\sigma(i, j) = \sigma(i, j|3) = 1$ . We count  $3 \cdot 3 = 9$  such pairings. Finally, since  $n = 7$ , we obtain

$$c_B(3) = \frac{9}{15} = \frac{3}{5}.$$

Next, consider node  $k = 2$ . The reasoning is similar to the above if we divide the other nodes into the sets  $\{0, 1\}$  and  $\{3, 4, 5, 6\}$ . Hence,

$$c_B(2) = \frac{1}{15} \cdot (2 \cdot 4) = \frac{8}{15}.$$

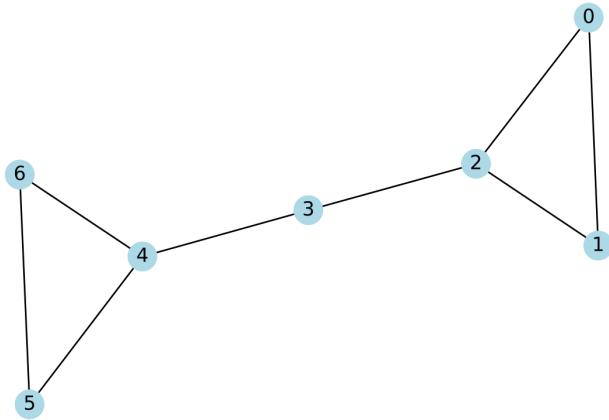


Figure 7.1: Barbell graph

We get the same value for  $c_B(4)$ .

All the other nodes appear in no shortest paths. For instance, any path passing through node 0 can be replaced with a shorter one that follows the edge between nodes 1 and 2. Hence  $c_B$  is zero on these nodes.

The `betweenness_centrality` function returns a dictionary with nodes as keys and  $c_B$  as values. (However, look at both of the next two examples.)

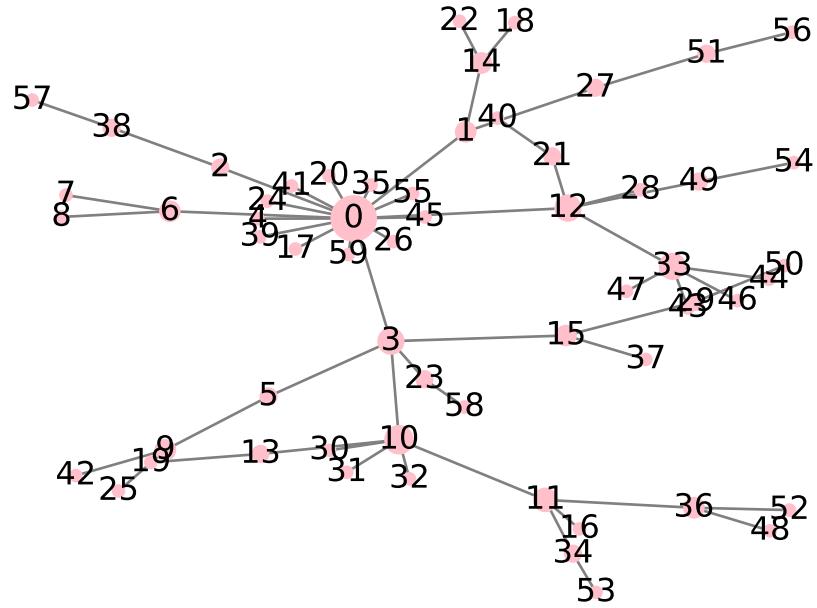
**Example 7.18.** Let's take a look at centrality measures for a power-law graph. Degree centrality points out a dominant hub:

```

P = nx.barabasi_albert_graph(60, 1, seed=2)
cdeg = pd.Series( nx.degree_centrality(P), index=P.nodes )

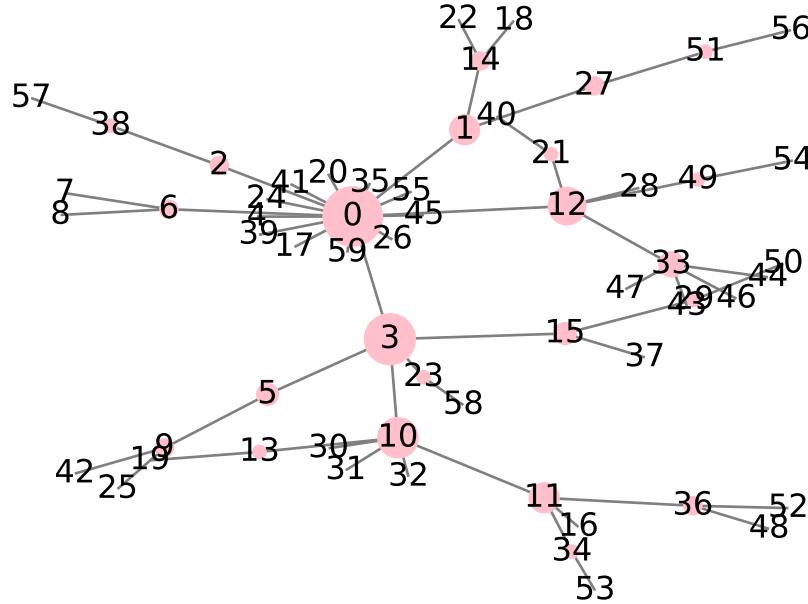
style = dict(edge_color="gray", node_color="pink", with_labels=True)
style["pos"] = nx.spring_layout(P, seed=3)
nx.draw(P, node_size=1000*cdeg, **style)

```



However, betweenness centrality also highlights some secondary hubs:

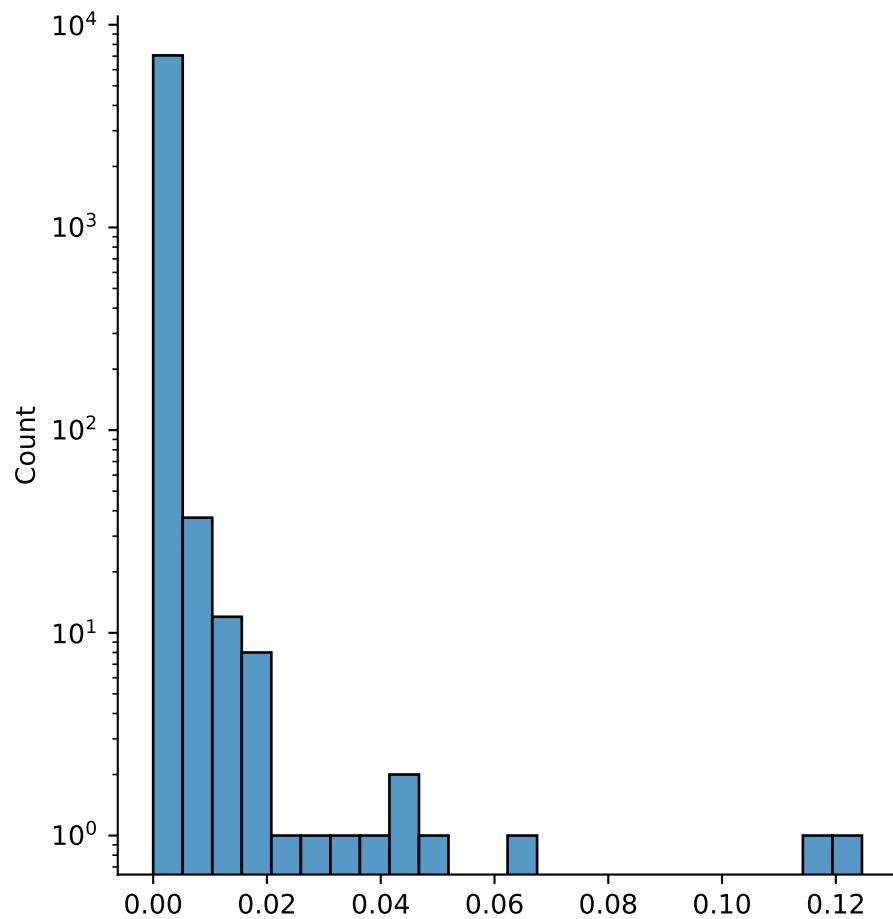
```
cbet = pd.Series(nx.betweenness_centrality(P), index=P.nodes)
nx.draw(P, node_size=600*cbet, **style)
```



The definition of  $c_B$  requires an expensive computation if the number of nodes is more than a few hundred. In practice, the  $\sigma$  values are often estimated by sampling, which is also offered by the `betweenness_centrality` function.

**Example 7.19.** With  $n > 7000$ , the Twitch network is much too large for calculating betweenness centrality exactly. So we provide a parameter  $k$  to use estimation by sampling:

```
tcb = pd.Series(  
    nx.betweenness_centrality(twitch, k=400, seed=302),  
    index=twitch.nodes  
)  
  
hist = sns.displot(x=tcb, bins=24);  
hist.axes[0,0].set_yscale("log");
```



As you can see above, only a few nodes have a significant amount of betweenness centrality:

```
tcb.sort_values().iloc[-10:]
```

	0
2732	0.024132
1103	0.030459
2447	0.034628
166	0.038451
6136	0.042333
1924	0.043946
5842	0.047202
3401	0.063231
1773	0.116461
4949	0.124590

### 7.6.3 Eigenvector centrality

A different way of distinguishing nodes of high degree is to suppose that not all links are equally valuable. By analogy with ranking sports teams, where wins over good teams should count for more than wins over bad teams, we should assign more importance to nodes that link to other important nodes.

We can try to turn this idea into an algorithm as follows. Suppose we initially assign uniform centrality scores  $x_1, \dots, x_n$  to all of the nodes. Now we can update the scores by looking at the current scores for all the neighbors. Specifically, the new scores are

$$x_i^+ = \sum_{j \text{ adjacent to } i} x_j = \sum_{j=1}^n A_{ij} x_j, \quad i = 1, \dots, n,$$

where  $A_{ij}$  are entries of the adjacency matrix. Once we have updated the scores, we can repeat the process to update them again, and so on. If the scores were to converge, in the sense that  $x_i^+$  approaches  $x_i$ , then we would have a solution of the equation

$$x_i \stackrel{?}{=} \sum_{j=1}^n A_{ij} x_j, \quad i = 1, \dots, n.$$

In fact, since the sums are all inner products across rows of  $\mathbf{A}$ , this is simply

$$\mathbf{x} \stackrel{?}{=} \mathbf{Ax}.$$

Except for  $\mathbf{x}$  equal to the zero vector, this equation does not have a solution in general. However, if we relax it just a bit, we get somewhere important. Instead of equality, let's look for *proportionality*, i.e.,

$$\lambda \mathbf{x} = \mathbf{Ax}$$

for a number  $\lambda$ . This is an **eigenvalue equation**, one of the fundamental problems in linear algebra.

**Example 7.20.** Consider the complete graph  $K_3$ , which is just a triangle. Its adjacency matrix is

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

We would expect to discover that all three vertices are ranked equally. In fact, if we define  $\mathbf{x} = \frac{1}{3}[1, 1, 1]$ , then

$$\mathbf{Ax} = \left[ \frac{2}{3}, \frac{2}{3}, \frac{2}{3} \right] = 2\mathbf{x},$$

`da=2$` is an eigenvalue to go with eigenvector  $\mathbf{x}$ . Note that any (nonzero) multiple of  $\mathbf{x}$  would work just as well:

$$\mathbf{A}(c\mathbf{x}) = \left[\frac{2}{3}c, \frac{2}{3}c, \frac{2}{3}c\right] = 2(c\mathbf{x}),$$

so that  $c\mathbf{x}$  is also an eigenvector. All that the eigenvector gives us, then, is *relative* centrality of the nodes, though it would be natural to normalize it so that its elements sum to 1.

Every  $n \times n$  matrix has at least one nonzero solution to the eigenvalue equation, although complex numbers might be involved. For an adjacency matrix, the *Perron–Frobenius theorem* guarantees a real solution for some  $\lambda > 0$  and for which the  $x_i$  all have the same sign. That last property allows us to interpret the  $x_i$  as relative importance or centrality of the nodes.

**Definition 7.10.** The **eigenvector centrality** of node  $i$  is the value of  $x_i$ , where  $\mathbf{x}$  is a positive eigenvector of the adjacency matrix.

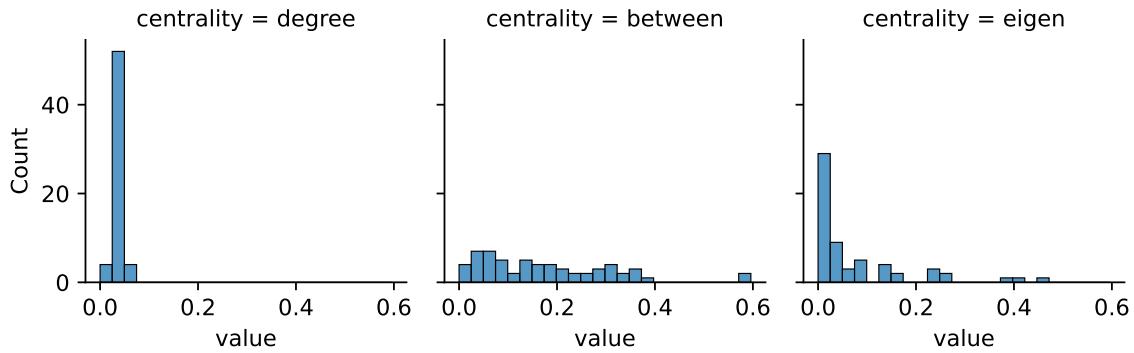
### Note

With a small tweak to avoid overestimating the neighbors of a major hub, eigenvector centrality becomes the *PageRank* algorithm, which launched Google.

NetworkX has two functions for computing eigenvector centrality. One that relies on numpy to solve the eigenvalue problem and is called `eigenvector_centrality_numpy`. As with betweenness centrality, the return value is a dictionary with nodes as the keys.

**Example 7.21.** Continuing with the small-world graph  $G$  from Example 7.16, we look at all three centrality measures side by side:

```
centrality = pd.DataFrame( {"degree": nx.degree_centrality(G)}, index=G.nodes )
centrality["between"] = nx.betweenness_centrality(G)
centrality["eigen"] = nx.eigenvector_centrality_numpy(G)
sns.displot(data=pd.melt(centrality, var_name="centrality"),
            x="value", col="centrality",
            height=2.4, bins=24
)
```



Of the three, eigenvector centrality does the most to create a relatively small distinguished group. Correlation coefficients suggest that while the three centrality measures are related, they are far from redundant:

```
centrality.corr()
```

	degree	between	eigen
degree	1.000000	0.630732	0.601884
between	0.630732	1.000000	0.736732
eigen	0.601884	0.736732	1.000000

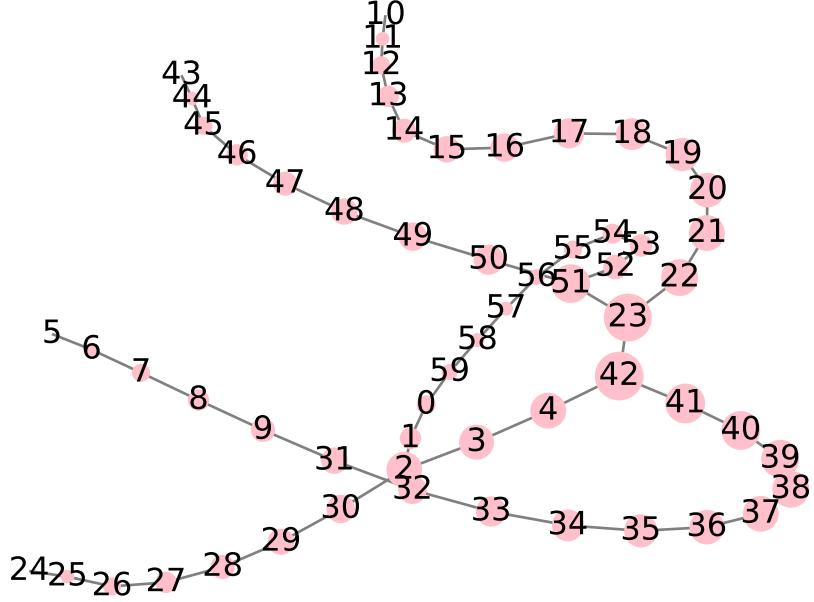
Here is a ranking based on betweenness:

```
centrality.sort_values(by="between", ascending=False).head(8)
```

	degree	between	eigen
42	0.050847	0.595850	0.405831
23	0.050847	0.576856	0.458056
41	0.033898	0.385739	0.232910
40	0.033898	0.368206	0.133669
51	0.050847	0.363822	0.392303
39	0.033898	0.349503	0.076714
38	0.033898	0.329632	0.044027
22	0.033898	0.329632	0.262883

As you can see, the top two are quite clear. A drawing of the graph supports the case that they are indeed central:

```
style["pos"] = nx.spring_layout(G, seed=1)
nx.draw(G, node_size=500*centrality["between"], **style)
```



A drawback, though, is that there are many secondary nodes whose values taper off only slowly as we enter the remote branches.

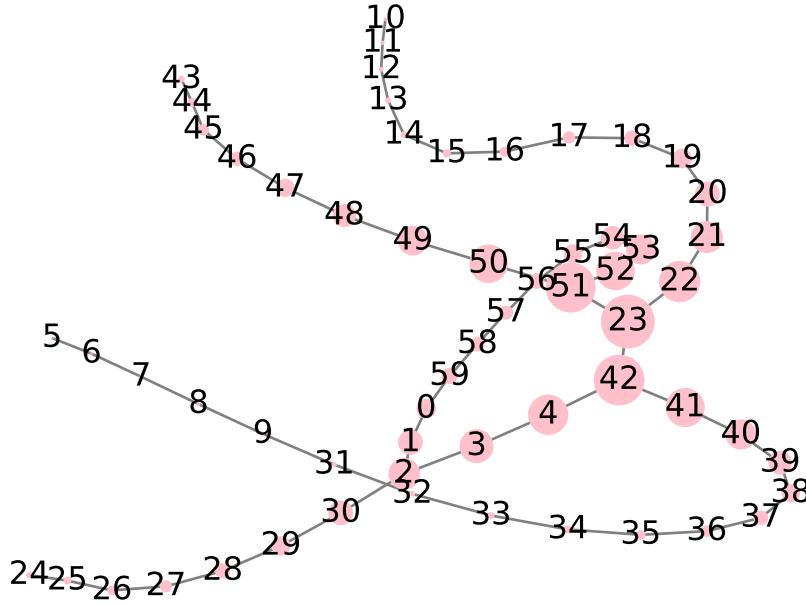
Here is a ranking according to eigenvector centrality:

```
centrality.sort_values(by="eigen", ascending=False).head(8)
```

	degree	between	eigen
23	0.050847	0.576856	0.458056
42	0.050847	0.595850	0.405831
51	0.050847	0.363822	0.392303
22	0.033898	0.329632	0.262883
4	0.033898	0.311806	0.249077
41	0.033898	0.385739	0.232910
52	0.033898	0.134717	0.225527
50	0.033898	0.212741	0.225125

This ranking has a clear top choice, followed by two that are nearly identical. Here we visualize the scores on the rendering:

```
nx.draw(G, node_size=800*centrality["eigen"], **style)
```



Eigenvector centrality identifies a more compact and distinct center.

## 7.7 Friendship paradox

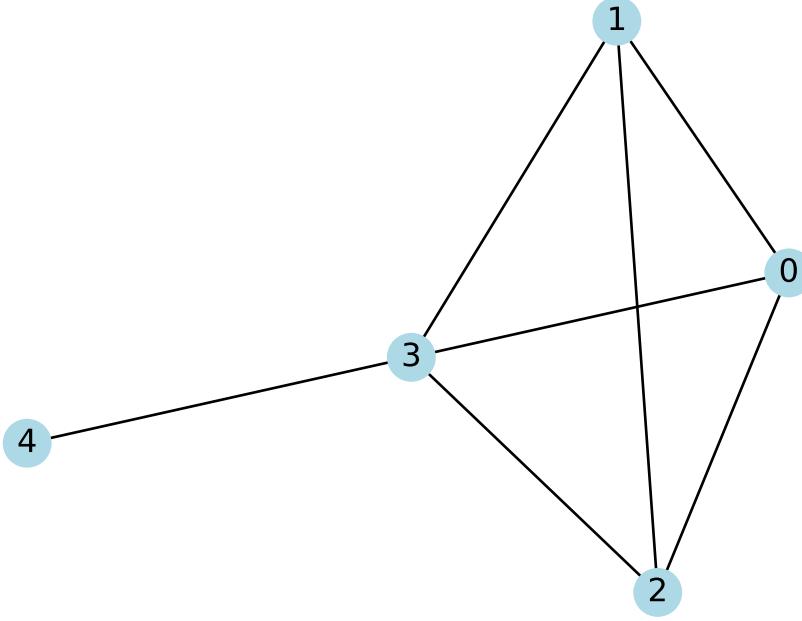
A surprising fact about social networks is that on average, your friends have more friends than you do, a fact that is called the **friendship paradox**. Let  $\mathbf{d}$  be an  $n$ -vector whose components are the degrees of the nodes in the network. On average, the number of “friends” (i.e., adjacent nodes) is the average degree, which is equal to

$$\frac{\|\mathbf{d}\|_1}{n}.$$

Now imagine that we create a list as follows: for each node  $i$ , add to the list the number of friends of each of  $i$ 's friends. The mean value of this list is the average number of “friends of friends.”

For example, consider the following graph:

```
L = nx.lollipop_graph(4, 1)  
nx.draw(L, with_labels=True, node_color="lightblue")
```



The average degree is  $(3+3+3+4+1)/5 = 14/5$ . Here are the entries in our friends-of-friends list contributed by each node:

- Node 0: 3 (from node 1), 3 (from node 2), 4 (from node 3)
- Node 1: 3 (from node 0), 3 (from node 2), 4 (from node 3)
- Node 2: 3 (from node 0), 3 (from node 1), 4 (from node 3)
- Node 3: 3 (from node 0), 3 (from node 1), 3 (from node 2), 1 (from node 4)
- Node 4: 4 (from node 3)

The average value of this list, i.e., the average number of friends' friends, is  $44/14 = 3.143$ , which is indeed larger than the average degree.

There is an easy way to calculate this value in general. Node  $i$  contributes  $d_i$  terms to the list, so the total number of terms is  $\|\mathbf{d}\|_1$ . We observe that node  $i$  appears  $d_i$  times in the list, each time contributing the value  $d_i$ , so the sum of the entire list must be

$$\sum_{i=1}^n d_i^2 = \|\mathbf{d}\|_2^2 = \mathbf{d}^T \mathbf{d}.$$

Hence the mathematical statement of the friendship paradox is

$$\frac{\|\mathbf{d}\|_1}{n} \leq \frac{\mathbf{d}^T \mathbf{d}}{\|\mathbf{d}\|_1}. \quad (7.3)$$

You are asked to prove this inequality in the exercises. Here is a verification for the BA graph above:

```
n = G.number_of_nodes()
d = pd.Series(dict(G.degree), index=G.nodes)
dbar = d.mean()
dbar_friends = np.dot(d,d) / d.sum()

print(dbar, "is less than", dbar_friends)
```

2.0 is less than 2.0666666666666667

The friendship paradox generalizes to eigenvector centrality: the average centrality of all nodes is less than the average of the centrality of all nodes' friends. The mathematical statement is

$$\frac{\|\mathbf{x}\|_1}{n} \leq \frac{\mathbf{x}^T \mathbf{d}}{\|\mathbf{d}\|_1}, \quad (7.4)$$

where  $\mathbf{x}$  is the eigenvector defining centrality of the nodes.

```
x = centrality["eigen"]
xbar = x.mean()
xbar_friends = np.dot(x,d) / sum(d)
print(xbar, "is less than", xbar_friends)
```

0.07365890048370116 is less than 0.08530969685141405

In fact, the friendship paradox inequality for any vector  $\mathbf{x}$  is equivalent to  $\mathbf{x}$  having nonnegative correlation with the degree vector.

## 7.8 Communities

In applications, one may want to identify *communities* within a network. There are many ways to define this concept precisely. We will choose a **random-walk** model.

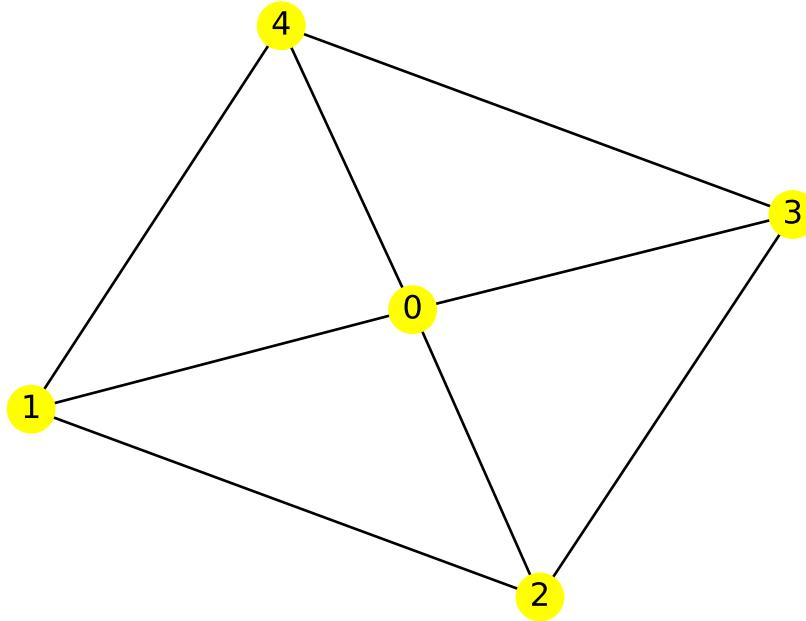
Imagine that a bunny sits on node  $i$ . In one second, the bunny hops to one of  $i$ 's neighbors, chosen randomly. In the next second, the bunny hops to another node chosen randomly from the neighbors of the one it is sitting on, etc. This is a random walk on the nodes of the graph.

Now imagine that we place another bunny on node  $i$  and track its path as it hops around the graph. Then we place another bunny, etc., so that we have an ensemble of walks. We

can now reason about the *probability* of the location of the walk after any number of hops. Initially, the probability of node  $i$  is 100%. If  $i$  has  $m$  neighbors, then each of them will have probability  $1/m$  after one hop, and all the other nodes (including  $i$  itself) have zero probability.

Let's keep track of the probabilities for this simple wheel graph:

```
G = nx.wheel_graph(5)
nx.draw(G, node_size=300, with_labels=True, node_color="yellow")
```



We start at node 4. This corresponds to the probability vector

$$\mathbf{p} = [0, 0, 0, 0, 1].$$

On the first hop, we are equally likely to visit each of the nodes 0, 1, or 3. This implies the probability distribution

$$\mathbf{q} = \left[ \frac{1}{3}, \frac{1}{3}, 0, \frac{1}{3}, 0 \right].$$

Let's now find the probability of standing on node 0 after the next hop. The two possible histories are 4-1-0 and 4-3-0, with total probability

$$\frac{1}{3} \cdot \frac{1}{3} + \frac{1}{3} \cdot \frac{1}{3} = \frac{2}{9}.$$

What about node 2 after two hops? The viable paths are 4-0-2, 4-1-2, and 4-3-2. Keeping in mind that node 0 has 4 neighbors, we get

$$\frac{1}{3} \cdot \frac{1}{4} + \frac{1}{3} \cdot \frac{1}{3} + \frac{1}{3} \cdot \frac{1}{3} = \frac{11}{36}.$$

This quantity is actually an inner product between the vector  $\mathbf{q}$  (probabilities of the prior location) and

$$\mathbf{w}_2 = [\frac{1}{4}, \frac{1}{3}, 0, \frac{1}{3}, 0],$$

which encodes the chance of hopping directly to node 2 from anywhere. In fact, the entire next vector of probabilities is just

$$[\mathbf{w}_1^T \mathbf{q}, \mathbf{w}_2^T \mathbf{q}, \mathbf{w}_3^T \mathbf{q}, \mathbf{w}_4^T \mathbf{q}, \mathbf{w}_5^T \mathbf{q}] = \mathbf{W}\mathbf{q},$$

where  $\mathbf{W}$  is the  $n \times n$  matrix whose rows are  $\mathbf{w}_1, \mathbf{w}_2, \dots$ . In terms of matrix-vector multiplications, we have the easy statement that the probability vectors after each hop are

$$\mathbf{p}, \mathbf{W}\mathbf{p}, \mathbf{W}(\mathbf{W}\mathbf{p}), \dots.$$

Explicitly, the matrix  $\mathbf{W}$  is

$$\mathbf{W} = \begin{bmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{3} & 0 & \frac{1}{3} & 0 \\ \frac{1}{4} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{3} & 0 & \frac{1}{3} & 0 \end{bmatrix}.$$

This has a lot of resemblance to the adjacency matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

The only difference is that each column has to be normalized by the number of options outgoing at that node, i.e., the degree of the node. Thus,

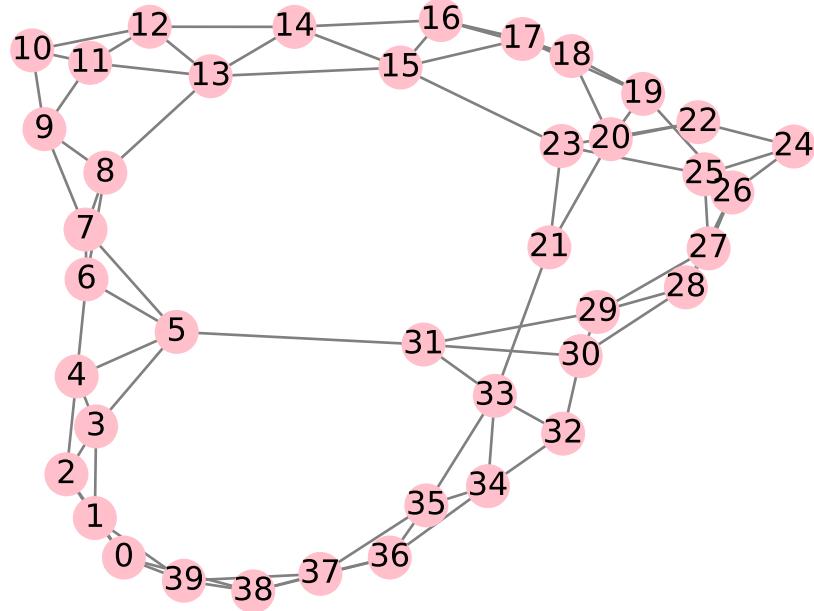
$$W_{ij} = \frac{1}{\deg(j)} A_{ij}.$$

### 7.8.1 Simulating the random walk

Let's do a simulation for a more interesting graph:

```
WS = nx.watts_strogatz_graph(40, 4, 0.04, seed=11)
pos = nx.spring_layout(WS, k=0.25, seed=1, iterations=200)
style = dict(pos=pos, with_labels=True, node_color="pink", edge_color="gray")

nx.draw(WS, node_size=240, **style)
```



First, we construct the random-walk matrix  $\mathbf{W}$ .

```
n = WS.number_of_nodes()
A = nx.adjacency_matrix(WS).astype(float)
```

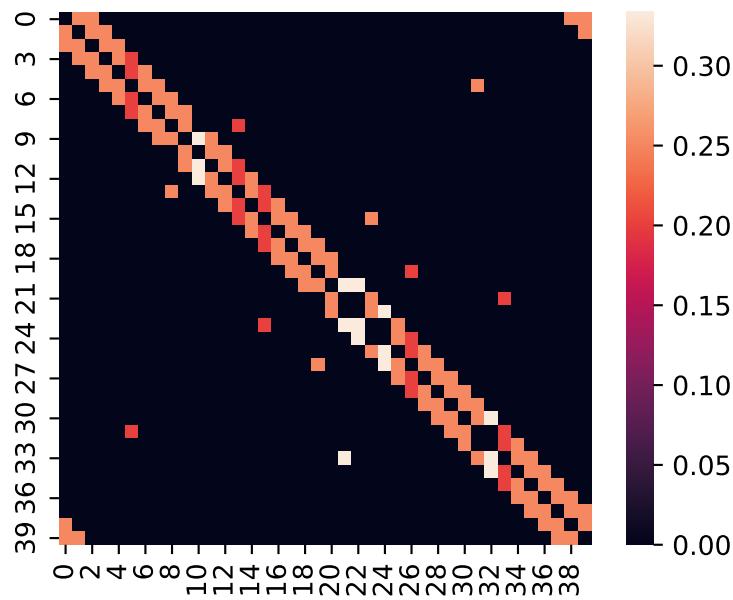
```

degree = [ WS.degree[i] for i in WS.nodes ]

W = A.copy()
for j in range(n):
    W[:,j] /= degree[j]

sns.heatmap(W.toarray()).set_aspect(1);

```



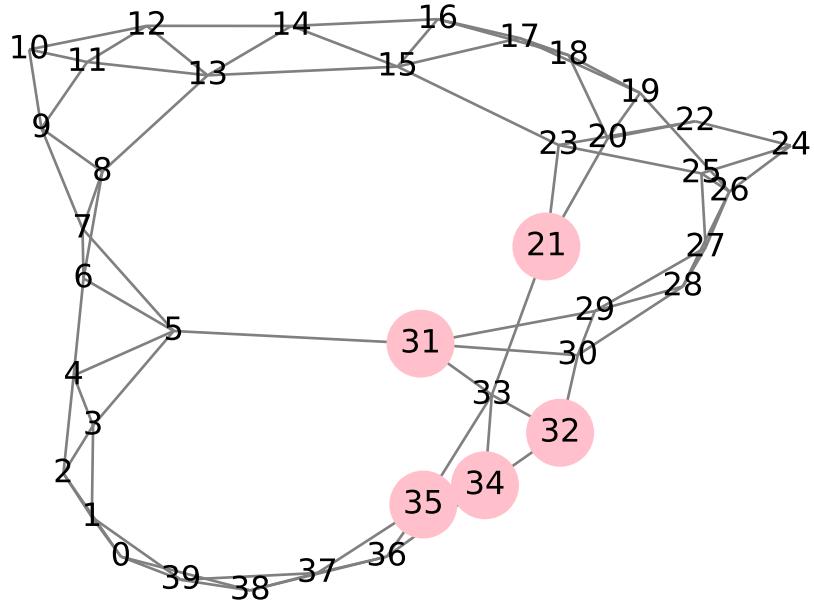
We set up a probability vector to start at node 0, and then use `W.dot` to compute the first hop. The result is to end up at 5 other nodes with equal probability:

```

init = 33
p = np.zeros(n)
p[init] = 1
p = W.dot(p)
sz = 3000*p
print("Total probability after 1 hop:", p.sum())
nx.draw(WS, node_size=sz, **style)

```

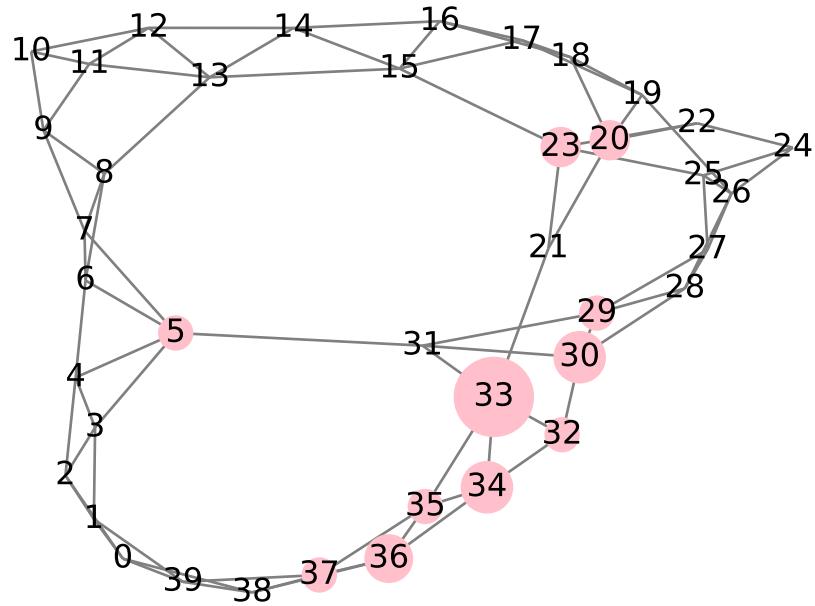
Total probability after 1 hop: 1.0



After the next hop, there will again be a substantial probability of being at node 33. But we could also be at some second-generation nodes as well.

```
p = W.dot(p)
print( "Total probability after 2 hops:", p.sum() )
nx.draw(WS, node_size=3000*p, **style)
```

Total probability after 2 hops: 1.0



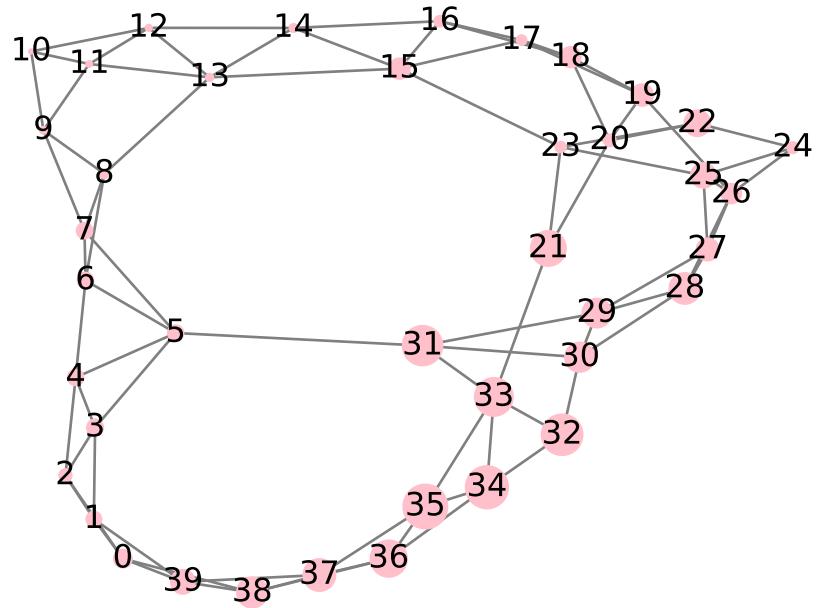
We'll take 3 more hops. That lets us penetrate a little into the distant nodes.

```

for k in range(3):
    p = W.dot(p)
print( "Total probability after 5 hops:", p.sum() )
nx.draw(WS, node_size=3000*p, **style)

```

Total probability after 5 hops: 1.0

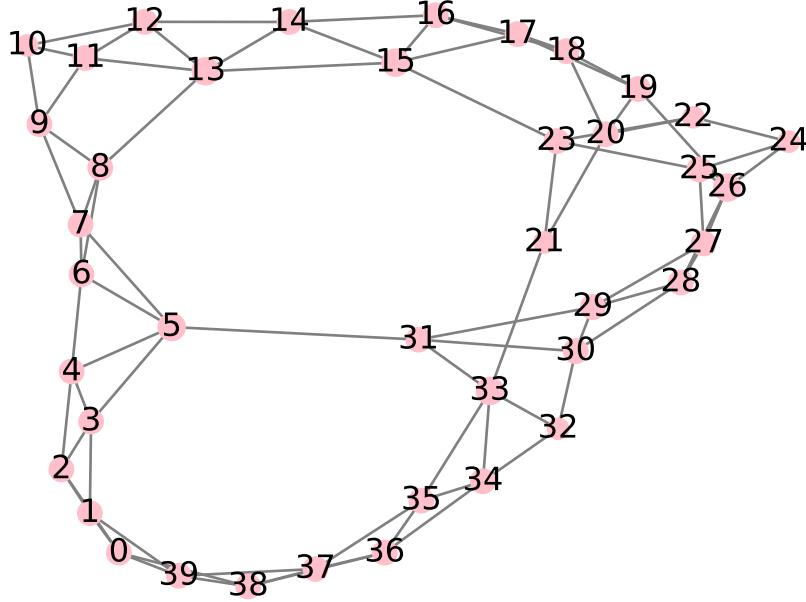


In the long run, the probabilities even out, as long as the graph is connected.

```

for k in range(200):
    p = W.dot(p)
nx.draw(WS, node_size=3000*p, **style)

```



### 7.8.2 Label propagation

The random walk brings us to a type of algorithm known as **label propagation**. We start off by “labelling” one or several nodes whose community we want to identify. This is equivalent to initializing the probability vector  $\mathbf{p}$ . Then, we take a running total over the entire history of the random walk:

$$\hat{\mathbf{x}} = \lambda \mathbf{p}_1 + \lambda^2 \mathbf{p}_2 + \lambda^3 \mathbf{p}_3 + \dots,$$

where  $0 < \lambda < 1$  is a damping parameter, and

$$\mathbf{p}_1 = \mathbf{W}\mathbf{p}, \mathbf{p}_2 = \mathbf{W}\mathbf{p}_1, \mathbf{p}_3 = \mathbf{W}\mathbf{p}_2, \dots.$$

In practice, we terminate the sum once  $\lambda^k$  is sufficiently small. The resulting  $\hat{\mathbf{x}}$  can be normalized to a probability distribution,

$$\mathbf{x} = \frac{\hat{\mathbf{x}}}{\|\hat{\mathbf{x}}\|_1}.$$

The value  $x_i$  can be interpreted as the probability of membership in the community.

Let's try looking for a community of node 0 in the WS graph above.

```
p = np.zeros(n)
p[init] = 1
lam = 0.8
```

We will compute  $\mathbf{x}$  by accumulating terms in a loop. Note that there is no need to keep track of the entire history of random-walk probabilities; we just use one generation at a time.

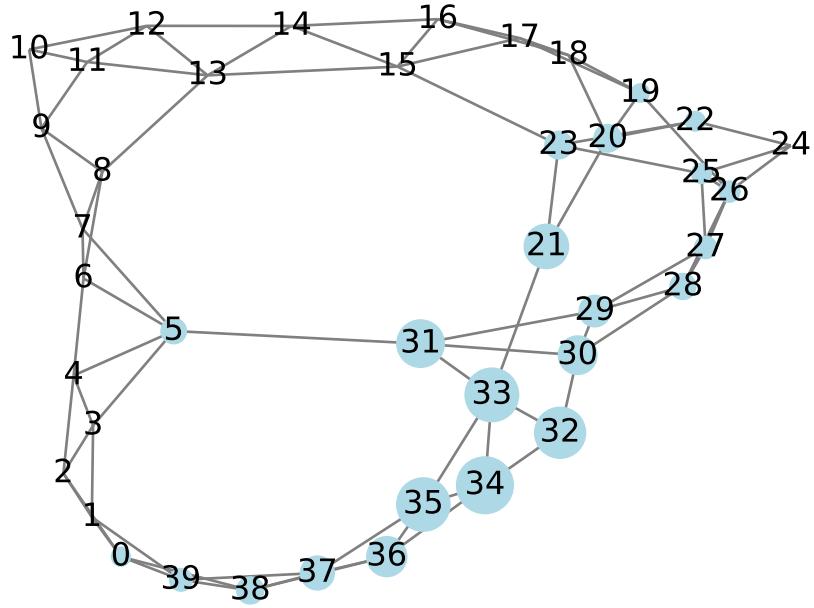
```
x = np.zeros(n)
mult = 1
for k in range(200):
    p = W.dot(p)
    mult *= lam
    x += mult*p

x /= np.sum(x) # normalize to probability distribution
```

The probabilities tend to be distributed logarithmically:

In the following rendering, any node  $i$  with a value of  $x_i < 10^{-2}$  gets a node size of 0. (You can ignore the warning below. It happens because we have negative node sizes.)

```
x[x<0.01] = 0
style["node_color"] = "lightblue"
nx.draw(WS, node_size=4000*x, **style)
```

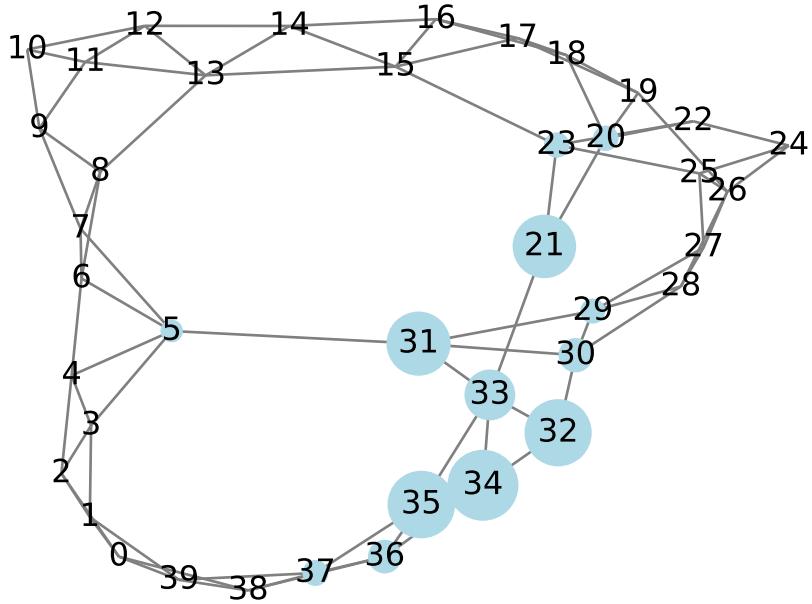


The parameter  $\lambda$  controls how quickly the random-walk process is faded out. A smaller value puts more weight on the early iterations, generally localizing the community more strictly.

```
p = np.zeros(n)
p[init] = 1
lam = 0.4
x = np.zeros(n)
mult = 1
for k in range(200):
    p = W.dot(p)
    mult *= lam
    x += mult*p

x /= np.sum(x)

x[x<0.01] = 0
nx.draw(WS, node_size=4000*x, **style)
```



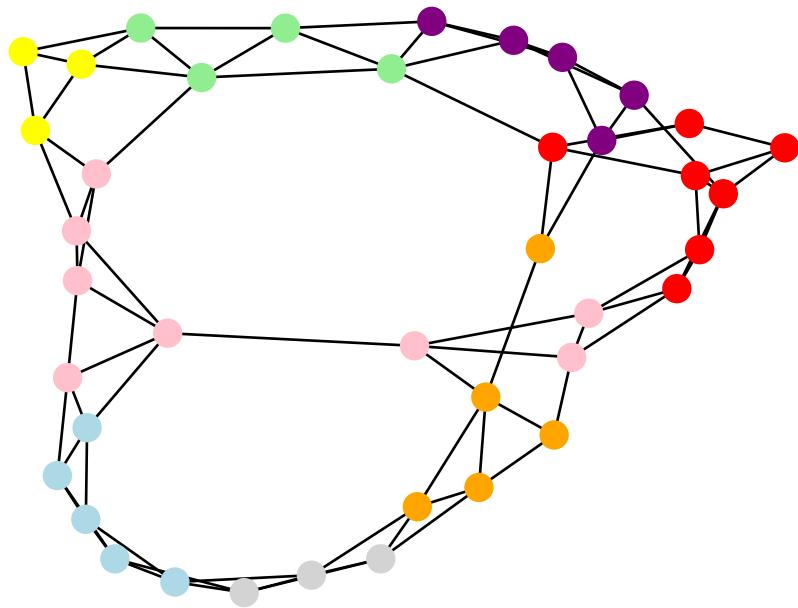
In practice, we could define a threshold cutoff on the probabilities, or set a community size and take the highest-ranking nodes. Then a new node could be selected and a community identified for it in the subgraph without the first community, etc.

A more sophisticated version of the label propagation algorithm (and many other community detection methods) is offered in a special module.

```
from networkx.algorithms.community import label_propagation_communities
comm = label_propagation_communities(WS)
[ print(c) for c in comm ];

{0, 1, 2, 3, 39}
{4, 5, 6, 7, 8, 29, 30, 31}
{9, 10, 11}
{12, 13, 14, 15}
{16, 17, 18, 19, 20}
{32, 33, 34, 35, 21}
{22, 23, 24, 25, 26, 27, 28}
{36, 37, 38}
```

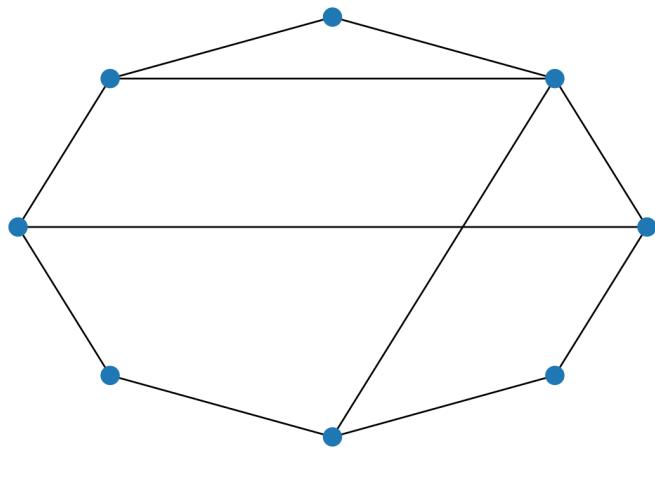
```
color = np.array( ["lightblue","pink","yellow","lightgreen","purple","orange","red","lightgrey"] )
color_index = [0]*n
for i,S in enumerate(comm):
    for k in S:
        color_index[k] = i
nx.draw( WS, node_size=100, pos=pos, node_color=color[color_index] )
```



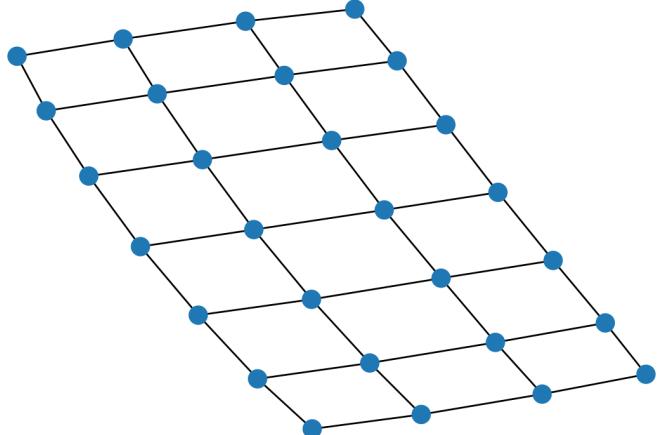
# Exercises

**Exercise 7.1.** For each graph, give the number of nodes, the number of edges, and the average degree.

(a) The complete graph  $K_6$ .



(b)



(c)

**Exercise 7.2.** Give the adjacency matrix for the graphs in Exercise 7.1 (parts (a) and (b) only).

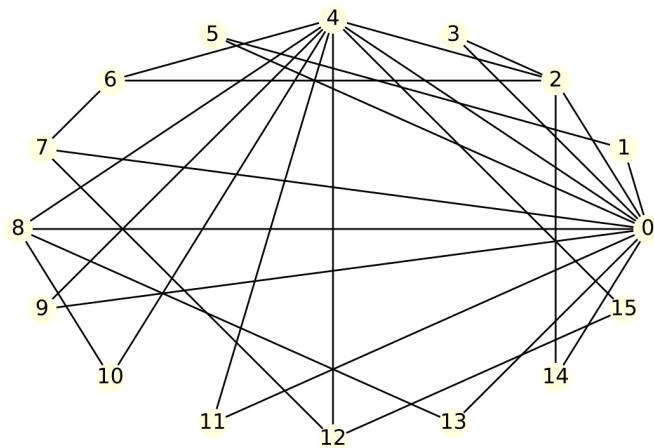
**Exercise 7.3.** For the graph below, draw the ego graph of (a) node 4 and (b) node 8.

**Exercise 7.4.** To construct an Erdős-Rényi graph on 25 nodes with expected average degree 8, what should the edge inclusion probability  $p$  be?

**Exercise 7.5.** Find the diameters of the graphs in Exercise 7.1.

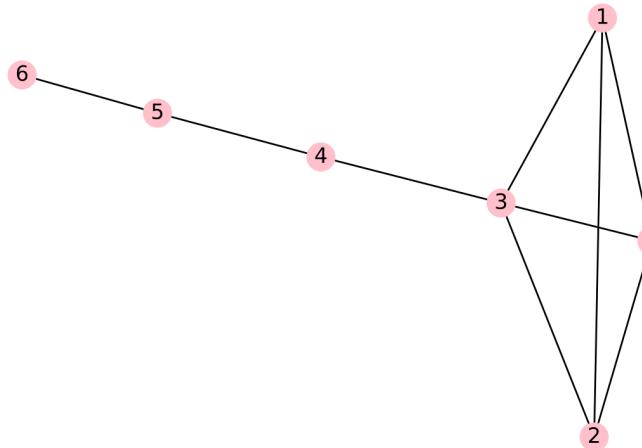
**Exercise 7.6.** Suppose that  $\mathbf{A}$  is the adjacency matrix of an undirected graph on  $n$  nodes. Let  $\mathbf{1}$  be the  $n$ -vector whose components all equal 1, and let

$$\mathbf{d} = \mathbf{A}\mathbf{1}.$$



Explain why  $\mathbf{d}$  is the vector whose components are the degrees of the nodes.

**Exercise 7.7.** Find (a) the clustering coefficient and (b) the betweenness centrality for each node in the following graph:



**Exercise 7.8.** A star graph with  $n$  nodes and  $n - 1$  edges has a central node that has an edge to each other node. In terms of  $n$ , find (a) the clustering coefficient and (b) the betweenness centrality of the central node of the star graph.

**Exercise 7.9.** The Watts–Strogatz construction starts with a *ring lattice* in which the nodes are arranged in a circle and each is connected to its  $k$  nearest neighbors (i.e.,  $k/2$  on each side). Show that the clustering coefficient of an arbitrary node in the ring lattice is

$$\frac{3(k-2)}{4(k-1)}.$$

(Hint: Count up all the edges between the neighbors on one side of the node of interest, then all the edges between neighbors on the other side, and finally, the edges going from a neighbor on one side to a neighbor on the other side. It might be easier to work with  $m = k/2$  and then eliminate  $m$  at the end.)

**Exercise 7.10.** Recall that the complete graph  $K_n$  contains every possible edge on  $n$  nodes. Prove that the vector  $\mathbf{x} = [1, 1, \dots, 1]$  is an eigenvector of the adjacency matrix of  $K_n$ . (Therefore, the eigenvector centrality is uniform over the nodes.)

**Exercise 7.11.** Prove that for the star graph on  $n$  nodes as described in Exercise 8, the vector

$$\mathbf{x} = [\sqrt{n-1}, 1, 1, \dots, 1]$$

is an eigenvector of the adjacency matrix, where the central node corresponds to the first element of the vector.

**Exercise 7.12.** Prove the friendship paradox, i.e., inequality Equation 7.3. (Hint: Start with Equation 6.1 using  $\mathbf{u} = \mathbf{d}$  and  $\mathbf{v}$  equal to a vector of all ones. Convert from equality to inequality to get rid of the angle  $\theta$ . Simplify the inner product, square both sides, and show that it can be rearranged into Equation 7.3.)