

Data Science 1

Tobin Driscoll

1/1/23

Table of contents

Preface	7
Prerequisites	7
Resources	8
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
Getting started	12
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
1 Representation of data	15
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	24
1.3 Qualitative data	27
1.3.1 Categorical	28
1.3.2 Text	29
1.3.3 Images	29

1.4	Series and frames	29
1.4.1	Loading from files	34
1.4.2	Selecting rows	35
1.5	Data preparation	36
1.5.1	Missing values	37
1.5.2	Loans example	38
1.5.3	Diamonds example	40
2	Descriptive statistics	43
2.1	Summary statistics	43
2.1.1	Mean and dispersion	44
2.1.2	z-scores	45
2.1.3	Populations and samples	46
2.1.4	Median and quantiles	48
2.2	Distributions	49
2.2.1	CDF	49
2.2.2	Empirical CDF	50
2.2.3	PDF	52
2.2.4	Mean and variance	56
2.2.5	Normal distribution	57
2.3	Grouping data	60
2.3.1	Splitting	60
2.3.2	Aggregation	67
2.3.3	Transformation	68
2.3.4	Filtering	69
2.4	Outliers	71
2.4.1	IQR	71
2.4.2	Mean and STD	73
2.4.3	Removing outliers	75
2.5	Correlation	77
2.5.1	Relational plots	77
2.5.2	Covariance	80
2.5.3	Pearson coefficient	81
2.5.4	Spearman coefficient	83
2.5.5	Categorical correlation	85
2.6	Cautionary tales	86
2.6.1	The datasaurus	86
2.6.2	Simpson's paradox	91
	Exercises	92
3	Classification	95
3.1	Classification basics	95
3.1.1	Encoding qualitative data	97
3.1.2	Walkthrough	97

	Introduction	1
	1 Data	17
1.1	Data types	17
1.2	Missing values	19
1.3	Categorical variables	21
1.4	Scaling and centering	23
1.5	Standardization	25
1.6	Normalizing	27
1.7	Logarithmic scale	29
1.8	Polynomial features	31
1.9	Combining features	33
1.10	Feature selection	35
1.11	Feature engineering	37
1.12	Summary	39
	Exercises	41
	Answers	43
	Further reading	45
	Summary	47
	Index	49
	References	51
	Contributors	53
	Code style guide	55
	Conventions	57
	Contributing	59
	Code repository	61
	Feedback	63
	License	65
	Attributions	67
	Copyright	69
	Contributor list	71
	Contributor contact	73
	Contributor contact	75
	Contributor contact	77
	Contributor contact	79
	Contributor contact	81
	Contributor contact	83
	Contributor contact	85
	Contributor contact	87
	Contributor contact	89
	Contributor contact	91
	Contributor contact	93
	Contributor contact	95
	Contributor contact	97
	Contributor contact	99
	Contributor contact	101
	Contributor contact	103
	Contributor contact	105
	Contributor contact	107
3.2	Classifier performance	99
3.2.1	Train–test paradigm	100
3.2.2	Binary classifiers	102
3.2.3	Multiclass classifiers	107
3.3	Decision trees	109
3.3.1	Gini impurity	109
3.3.2	Partitioning	110
3.3.3	Toy example	111
3.3.4	Penguin data	113
3.3.5	Limitations	116
3.4	Nearest neighbors	116
3.4.1	Norms	116
3.4.2	Algorithm	117
3.4.3	KNN in sklearn	118
3.4.4	Standardization	121
3.4.5	Pipelines	122
3.5	Quantifying votes	123
3.5.1	ROC curve	124
3.5.2	AUC	130
	Exercises	131
4	Model selection	134
4.1	Bias–variance tradeoff	135
4.1.1	Learner bias	135
4.1.2	Variance	136
4.1.3	Learning curves	137
4.2	Overfitting	140
4.2.1	Overfitting in kNN	140
4.2.2	Overfitting in decision trees	141
4.2.3	Overfitting and variance	143
4.3	Ensemble methods	147
4.4	Validation	152
4.4.1	Cross-validation	152
4.4.2	Hyperparameter tuning	154
5	Regression	160
5.1	Linear regression	161
5.1.1	Linear algebra	162
5.1.2	Performance metrics	163
5.1.3	Case study: Arctic ice	165
5.2	Multilinear and polynomial regression	173
5.2.1	Case study: Advertising and sales	174
5.2.2	Polynomial regression	178
5.2.3	Case study: Fuel efficiency	178

5.3	Regularization	184
5.3.1	Case study: Diabetes	185
5.4	Nonlinear regression	191
5.4.1	Nearest neighbors	192
5.4.2	Decision tree	193
5.5	Logistic regression	198
5.5.1	Loss function	199
5.5.2	Regularization	199
5.5.3	Case study: Personal spam filter	200
5.5.4	Multiclass case	203
	Exercises	205
6	Clustering	211
6.1	Similarity and distance	215
6.1.1	Distance metrics	216
6.1.2	Probability distributions	217
6.1.3	Distance matrix	218
6.1.4	Distance in high dimensions	220
6.2	Performance measures	221
6.2.1	Rand index and ARI	221
6.2.2	Silhouettes	222
6.3	k-means	230
6.3.1	Lloyd's algorithm	231
6.3.2	Practical issues	231
6.4	Hierarchical clustering	240
6.4.1	Case study: Penguins	247
	Exercises	249
7	Networks	251
7.1	Graphs	251
7.1.1	NetworkX	251
7.1.2	Common graph types	254
7.1.3	Adjacency matrix	258
7.1.4	Importing networks	259
7.1.5	Degree and average degree	261
7.1.6	Random graphs	262
7.2	Clustering	265
7.2.1	Watts–Strogatz graphs	271
7.3	Distance	274
7.3.1	ER graphs	276
7.3.2	Watts–Strogatz graphs	279
7.3.3	Twitch network	281
7.4	Degree distributions	285
7.4.1	Power-law distribution	289

7.4.2	Barabási–Albert graphs	291
7.5	Centrality	295
7.5.1	Betweenness centrality	297
7.5.2	Eigenvector centrality	299
7.5.3	Comparison	302
7.5.4	Power-law example	305
7.5.5	Friendship paradox	308
7.6	Communities	310
7.6.1	Simulating the random walk	313
7.6.2	Label propagation	318
	Exercises	322

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>.

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.

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 ∞/∞ 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('2023-01-13T21:51:59')
```

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

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

numpy.timedelta64(-28129183, 'm')
```

1.1.3 Random numbers

Generating truly random numbers on a computer is not simple. Mostly we rely on *pseudo-random* 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 “bell curve” (normal or Gaussian distribution).

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

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

In the long run, the average value of these numbers will be zero.

```
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')

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:

 Danger

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

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])
```

⚠ Warning

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]])
```

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., 1., 0., 0., 0.],
       [1., 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 could therefore use successive brackets to refer to an element:

```
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.

```
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 1.3. 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 1.4. 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 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 1.5. 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 1.6. 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"
)

print(wavelength)

violet    400
blue      470
green     520
yellow    580
orange    610
red       710
Name: wavelength, dtype: int64
```

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

```
print( wavelength["blue"] )
```

470

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

```

print( wavelength[ ["violet", "red"] ] )

violet      400
red         710
Name: wavelength, dtype: int64

```

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  
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 vector of rows, plus (optionally) the index and the names of the columns:

```

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

letters

```

	lowercase	uppercase
0	a	A
1	b	B
2	c	C

Pandas has facilities for dealing with categorical variables.

Example 1.7. 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 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 1.8. 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/advertis")
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

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

Here's a local file that contains estimates and projections for greenhouse gas emissions in Delaware:

```
emissions = pd.read_csv("Delaware_Greenhouse_Gas_Emissions.csv")
emissions.head()
```

	Year	Category	Economic sectors	Source	Type	Fuel type	Gas	MMTCC
0	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	CO2	5.0249
1	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Petroleum	CO2	1.8523
2	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Natural Gas	CO2	0.6112
3	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	N2O	0.0239
4	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Petroleum	N2O	0.0038

1.4.2 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 1.9. If we apply a relational operator to a column of a frame, we get a series of Boolean values:

```
emissions["Fuel type"] == "Coal"
```

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

```
coal = emissions[ emissions["Fuel type"]=="Coal" ]
coal.head()
```

	Year	Category	Economic sectors	Source	Type	Fuel type	Gas	MMTCO2
0	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	CO2	5.024903
3	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	N2O	0.02395
6	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	CH4	0.001348
20	1990	Estimate	Industrial	Fossil Fuel Combustion	Energy	Coal	CO2	0.077063
23	1990	Estimate	Industrial	Fossil Fuel Combustion	Energy	Coal	N2O	0.00038

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

```
ispower = (emissions["Economic sectors"]=="Electric power")
isco2 = (emissions["Gas"]=="CO2")
selected = emissions[ ispower & ~isco2 ]
selected.head(10)
```

	Year	Category	Economic sectors	Source	Type	Fuel type	Gas
3	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	N2O
4	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Petroleum	N2O
5	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Natural Gas	N2O
6	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	CH4
7	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Petroleum	CH4
8	1990	Estimate	Electric power	Fossil Fuel Combustion	Energy	Natural Gas	CH4
9	1990	Estimate	Electric power	Transmission and Distribution	Non-Energy	Nan	SF6
74	1991	Estimate	Electric power	Fossil Fuel Combustion	Energy	Coal	N2O
75	1991	Estimate	Electric power	Fossil Fuel Combustion	Energy	Petroleum	N2O
76	1991	Estimate	Electric power	Fossil Fuel Combustion	Energy	Natural Gas	N2O

💡 Tip

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

💡 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:

```
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("loan.csv")
loans.head()
```

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

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	install
0	1077501	1296599	5000	5000	4975.0	36 months	10.65	1
1	1077430	1314167	2500	2500	2500.0	60 months	15.27	
2	1077175	1313524	2400	2400	2400.0	36 months	15.96	
3	1076863	1277178	10000	10000	10000.0	36 months	13.49	
4	1075358	1311748	3000	3000	3000.0	60 months	12.69	

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.loc[:, 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()

```

	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)
```

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 dispersion

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 σ , 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.

Example 2.1. Suppose that $x_1 = 0$, $x_2 = t$, and $x_3 = -t$, where $|t| \leq 5$. What are the minimum and maximum possible values of the standard deviation?

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

$$\sigma^2 = 0^2 + t^2 + (-t)^2 = 2t^2.$$

The function $f(t) = 2t^2$ has a single critical value at $t = 0$. Clearly, $f(0) = 0$ is both a local and a global minimum. Since $-5 \leq t \leq 5$, we also find that $f(\pm 5) = 50$ is a global maximum.

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**:

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

The z-scores have mean zero and standard deviation equal to 1; in physical terms, they are dimensionless. This makes them attractive to work with and compare across data sets.

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

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

These are independent of t , which just scales the original values. So, for instance, the z-scores of a collection of distances are unaffected if the distances are expressed in centimeters or kilometers.

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

🔥 Danger

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.

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 μ . That is, we say that \bar{x} is an **unbiased estimator** for μ .

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 σ^2 ; hence, s_n^2 is called a **biased estimator** of the population variance.

An unbiased estimator for σ^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.

🔥 Danger

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 σ . 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 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.

In other words, percentiles are the inverse function of the CDF.

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.

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 2.4. 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 $(4 + 5)/2 = 4.5$.

A set of percentiles dividing probability into q equal pieces is called the q -**quantiles**.

Example 2.5. The 4-quantiles are called **quartiles**. The first quartile is the 25th percentile, or the value that exceeds 1/4 of the population. The second quartile is the median. The third quartile is the 75th percentile.

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



Danger

If this all isn't confusing enough yet, sometimes the word *quantile* is used to mean *percentile*. This is the case for the `quantile` method in pandas.

Definition 2.3. The **interquartile range** (IQR) is the difference between the 75th percentile and the 25th percentile.

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 `describe` method includes mean, standard deviation, and the quartiles.

```
from numpy.random import default_rng
import pandas as pd
rng = default_rng(19716)
df = pd.DataFrame( {
    "normal" : rng.normal( size=(4000,) ),
    "uniform" : rng.uniform( size=(4000,) )
} )
df.describe()
```

	normal	uniform
count	4000.000000	4000.000000
mean	-0.026801	0.493338
std	0.995135	0.293177
min	-3.361732	0.000257
25%	-0.685307	0.232339
50%	-0.017852	0.486322
75%	0.633775	0.750565
max	3.115486	0.999204

2.2 Distributions

Mean and dispersion (variance or STD) 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

$$F(t) = \text{fraction of the population that is less than or equal to } t,$$

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

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

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.

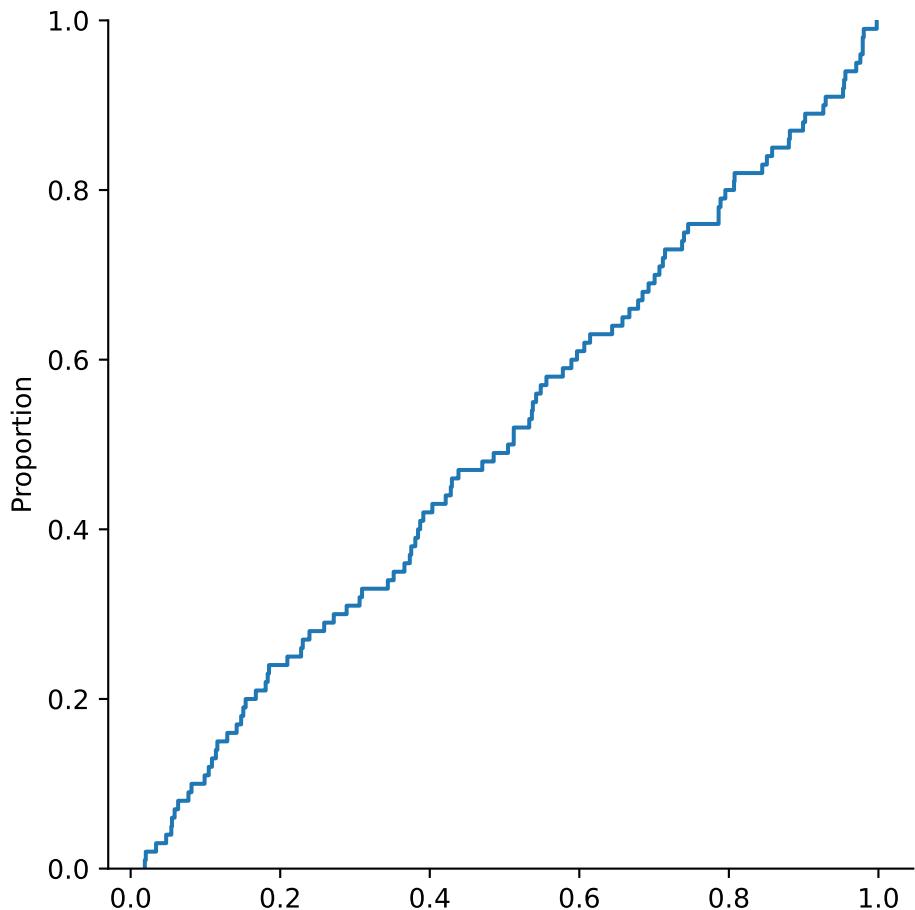
2.2.2 Empirical CDF

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 .

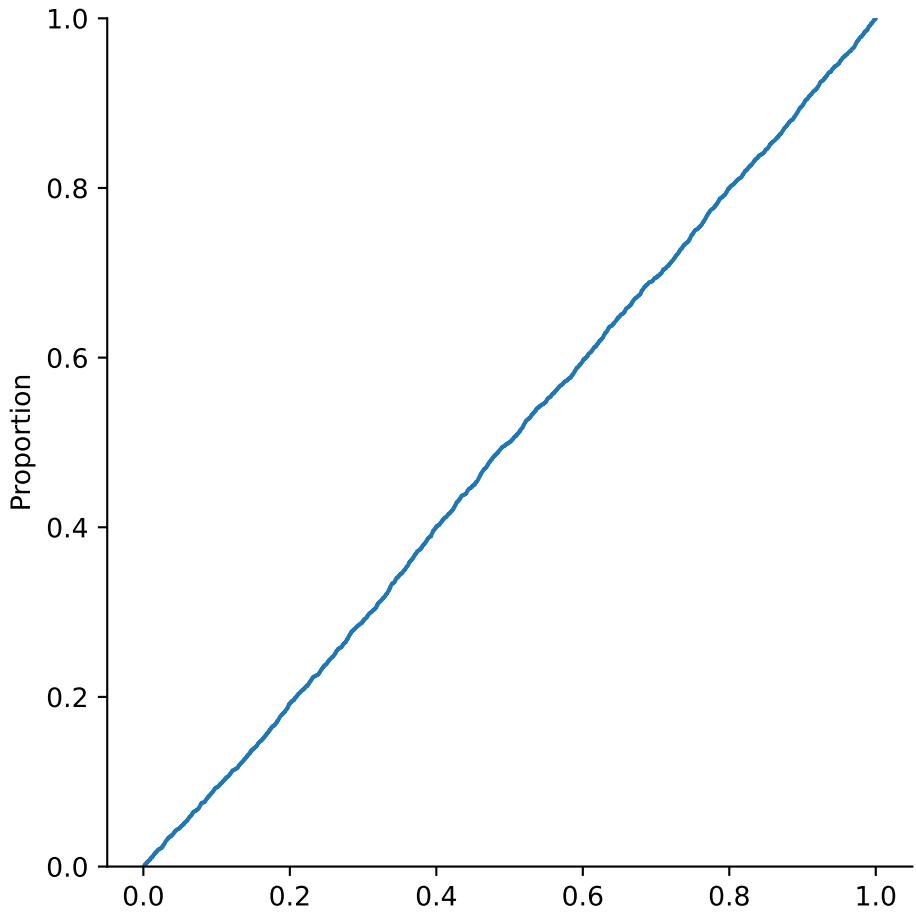
Example 2.8. Here is an experiment that producing the ECDF for a sample from the random number generator:

```
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.

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

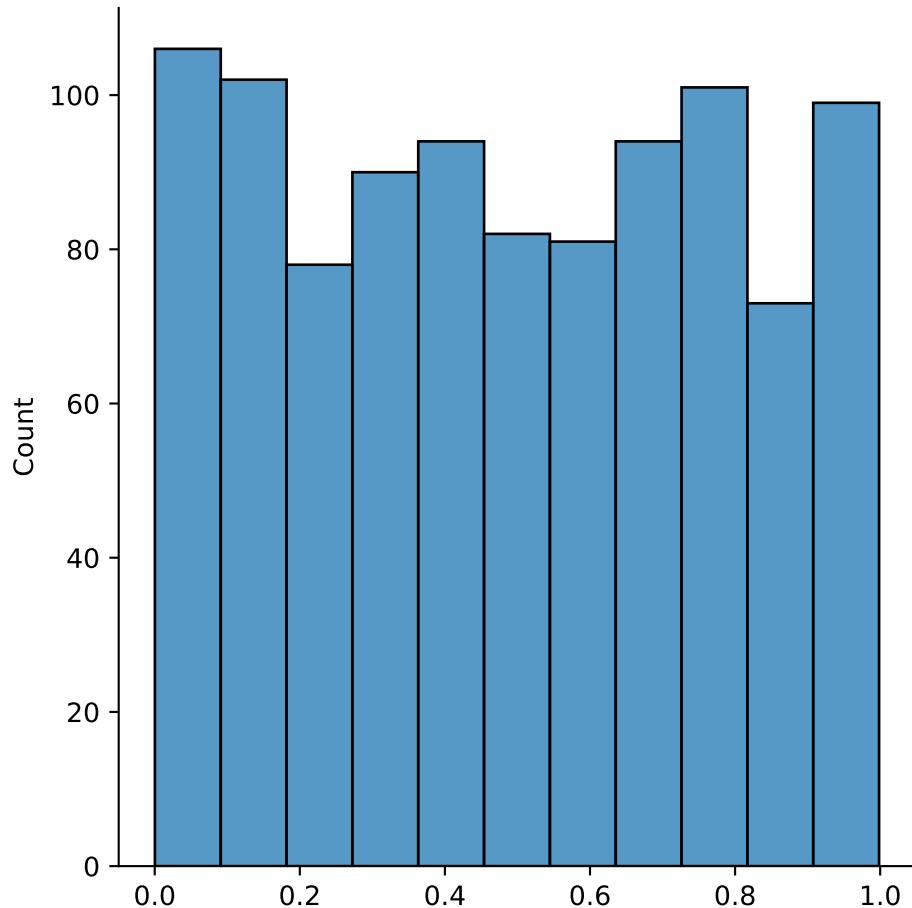
$$B_k = (t_k, t_{k+1}], \quad k = 0, \dots, m,$$

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.

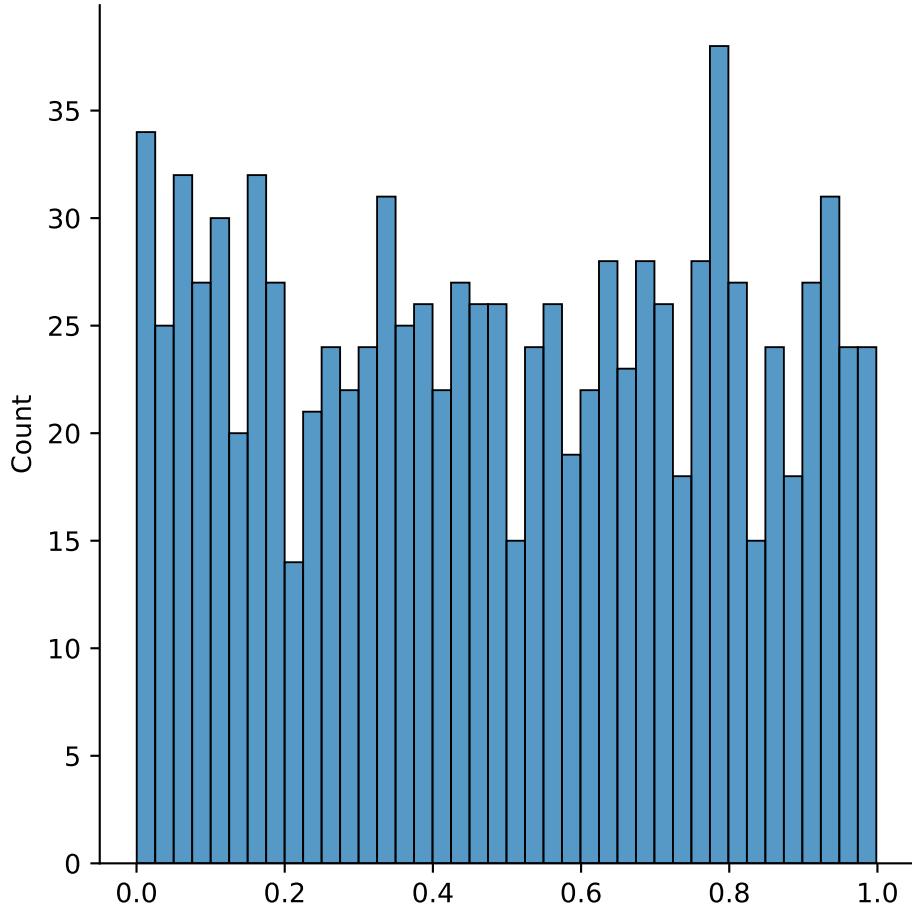
Example 2.9. Continuing with the uniform distribution over $[0, 1]$:

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



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

```
import numpy as np
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

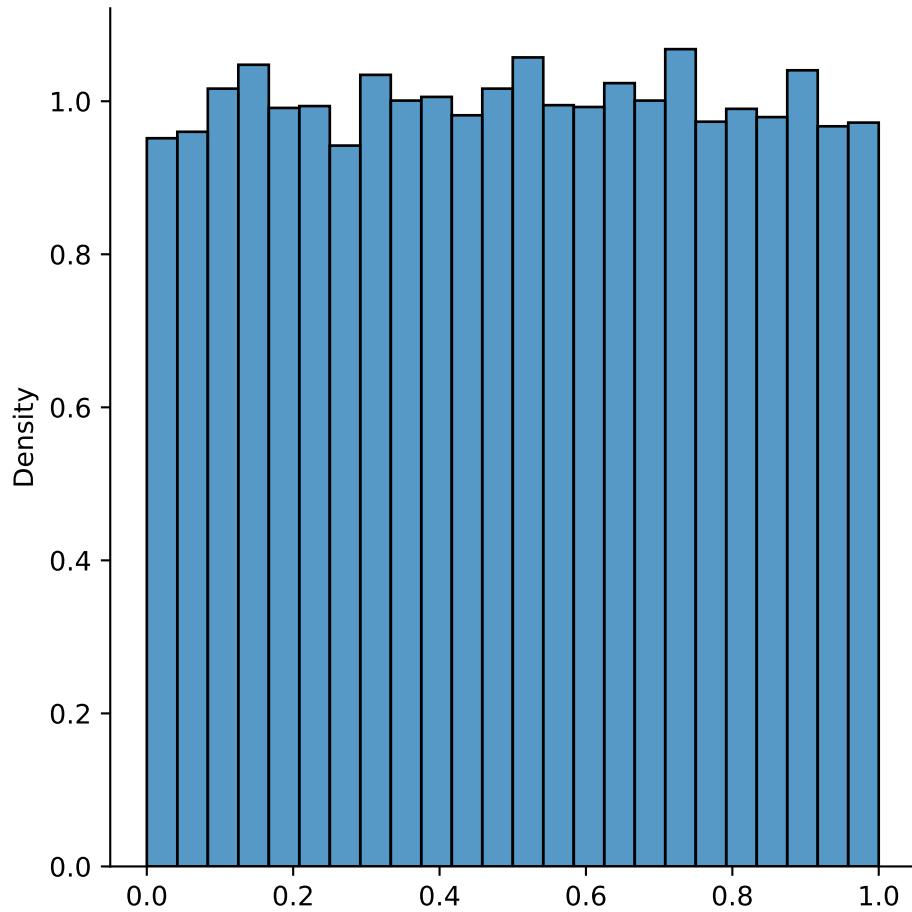
$$\frac{c_k}{t_{k+1} - t_k} = \frac{\hat{F}(t_{k+1}) - \hat{F}(t_k)}{t_{k+1} - t_k}. \quad (2.7)$$

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.

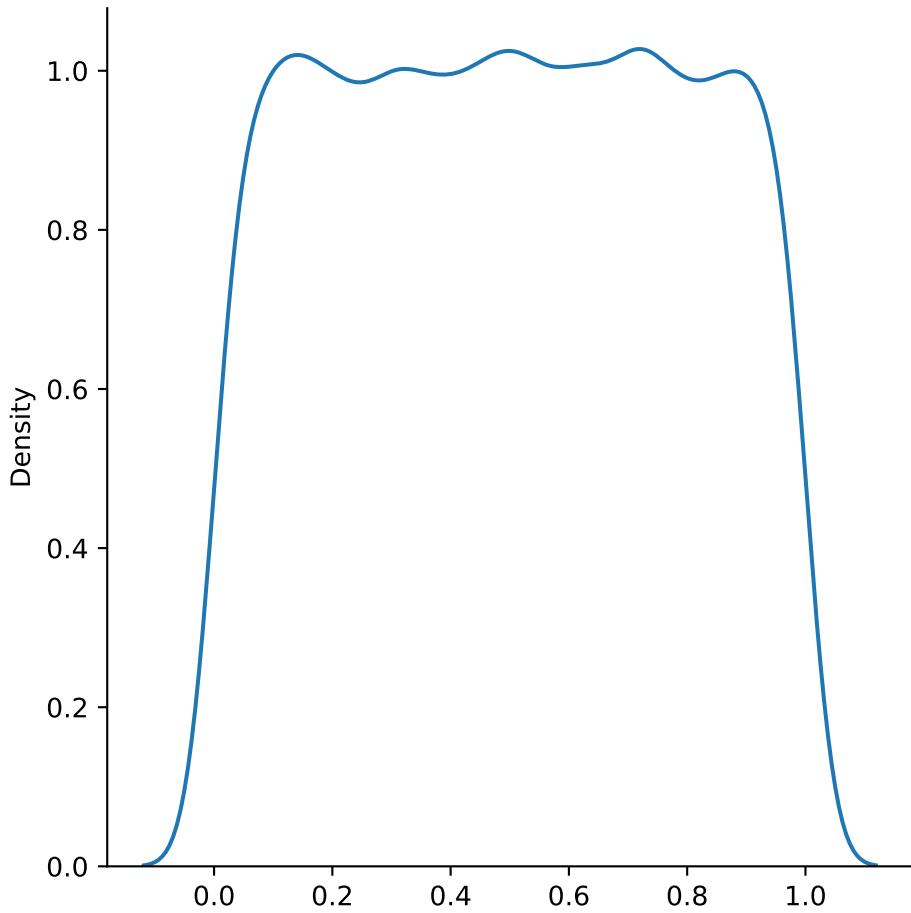
Example 2.10. If we have many samples, then we can use a normalized histogram to give an approximation of the PDF:

```
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:

$$\begin{aligned}\mu &= \int x f(x) dx \\ \sigma^2 &= \int (x - \mu)^2 f(x) dx,\end{aligned}$$

where the integrals are taken over the domain of f .

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}")

\u03bc = 0.50518, 12 ** 2 = 1.00019
```

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 μ and variance σ^2 is defined by the PDF

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

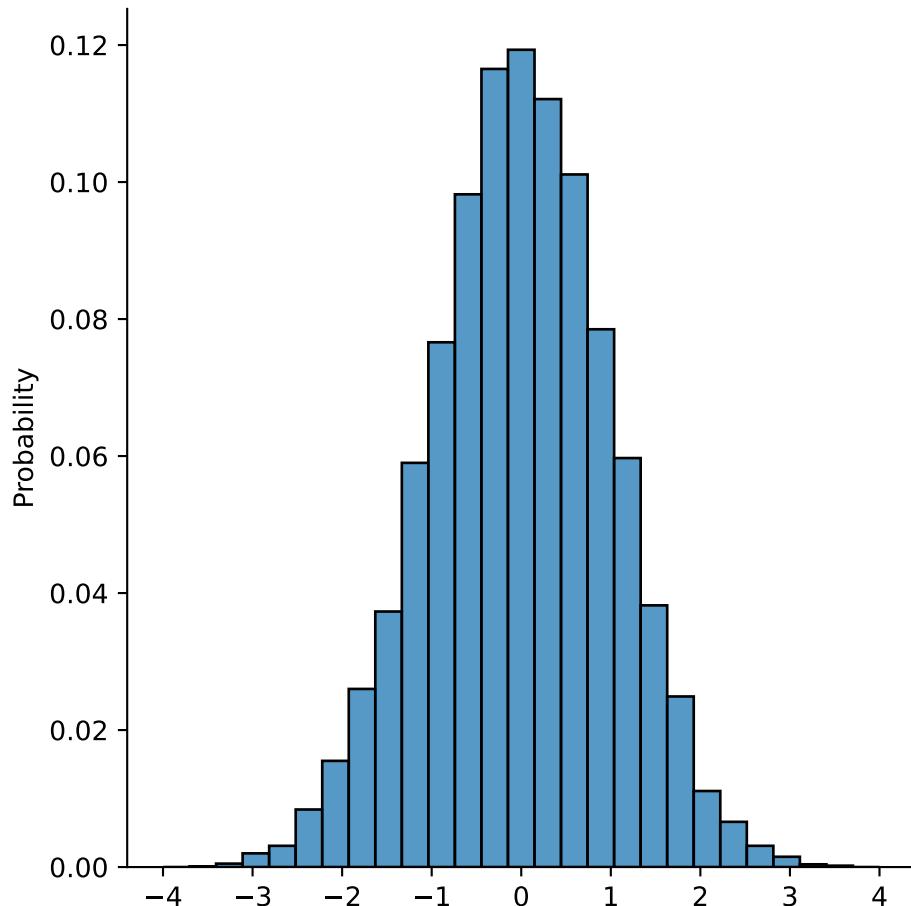
The **standard normal** distribution uses $\mu = 0$ and $\sigma = 1$.

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.

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");
```



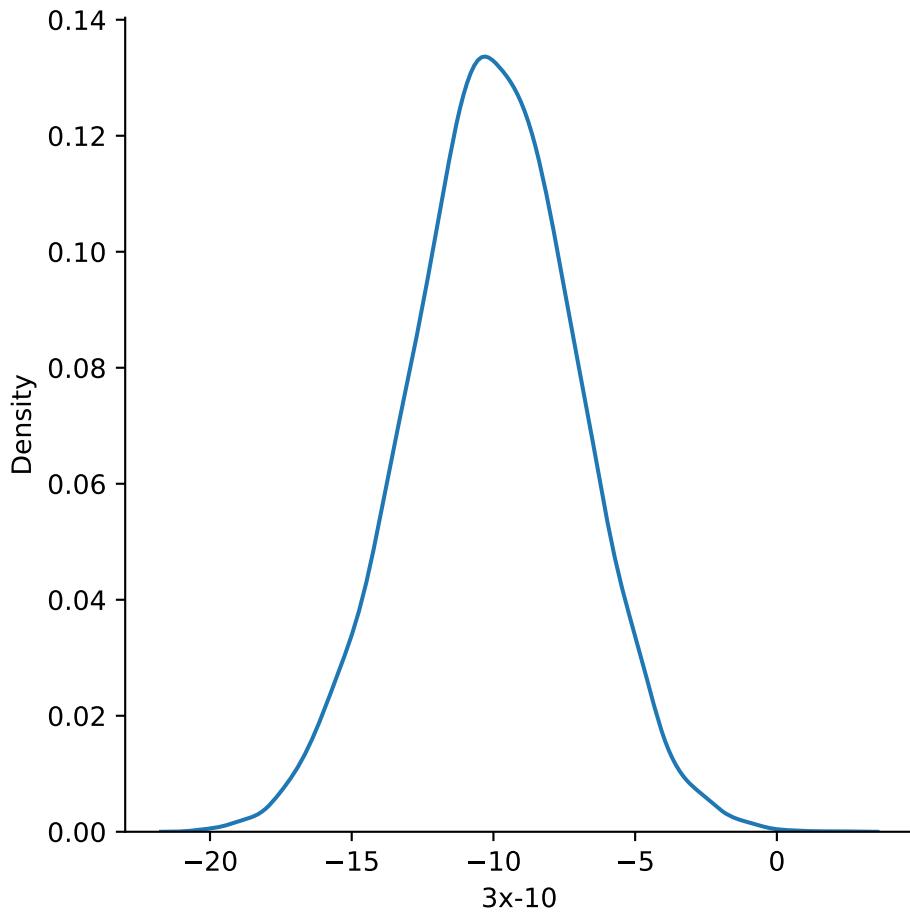
We can change the variance by multiplication by σ and change the mean by adding μ :

```
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");
```

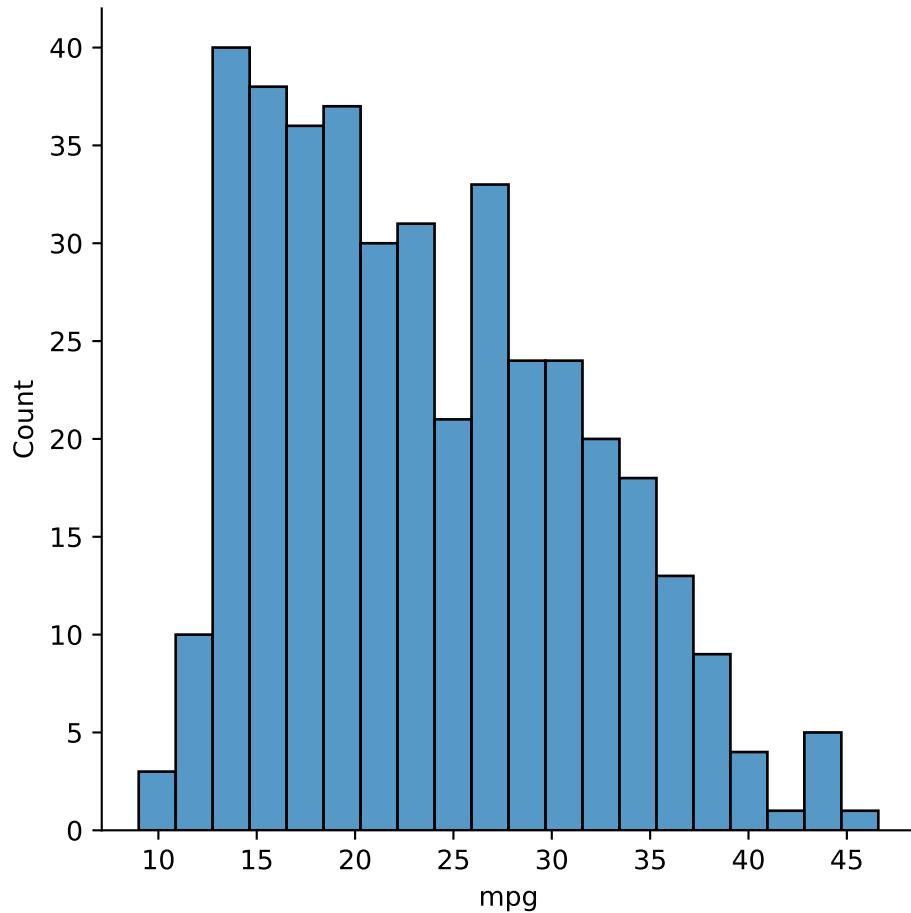


2.3 Grouping data

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

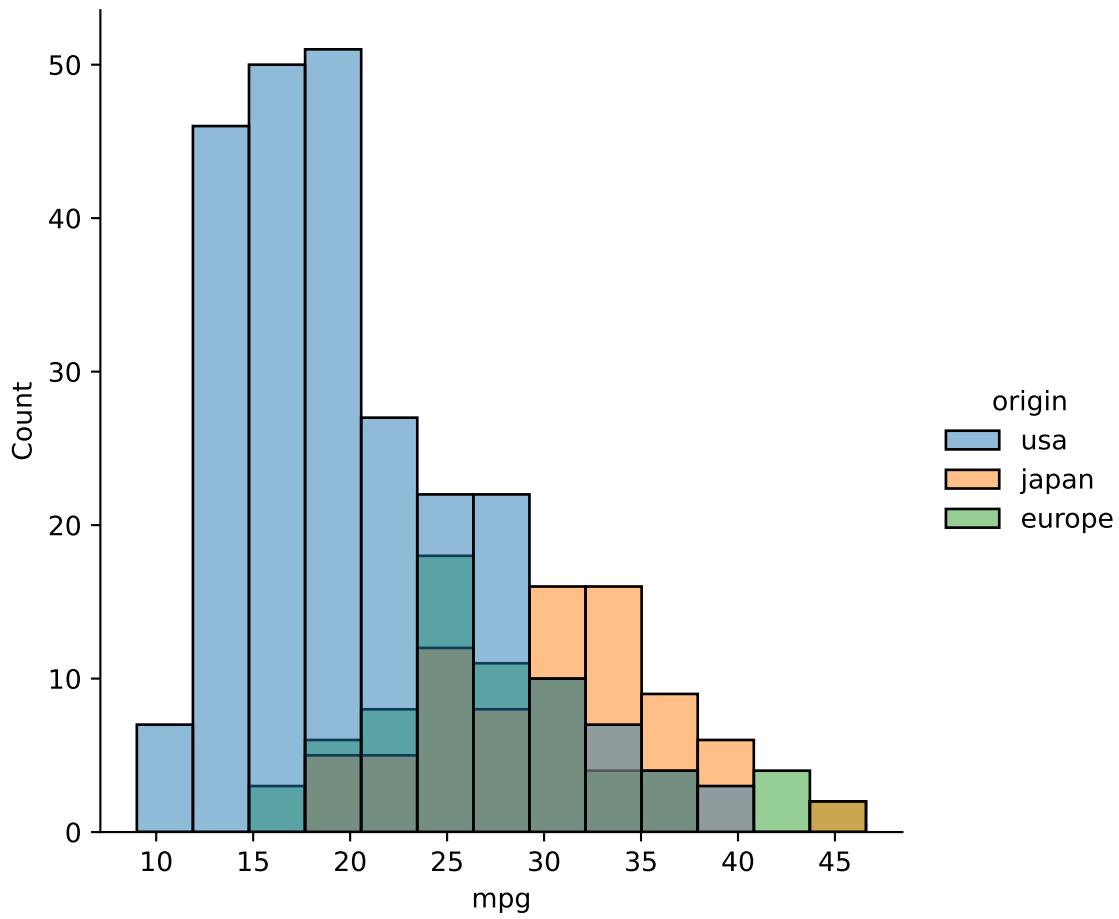
The `groupby` method for a data frame splits the frame into groups based on categorical values in a designated column:

```
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

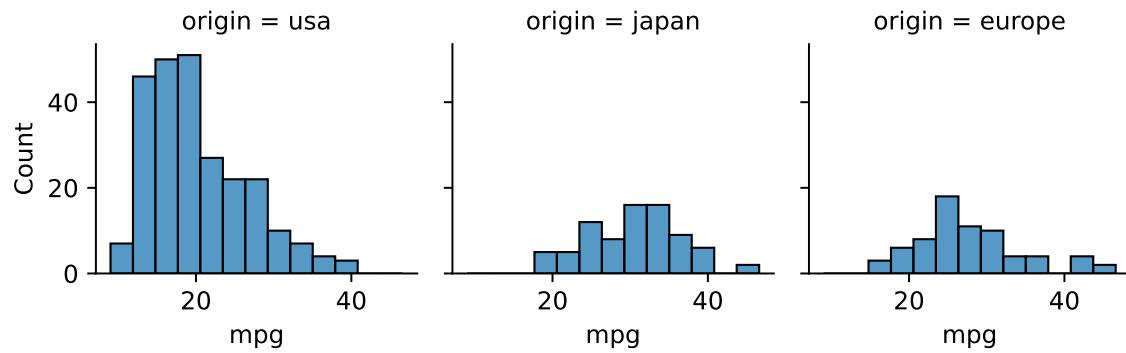
Both the median and the mean values are quite a bit lower for *usa* cars than for the other regions. This is also apparent when we plot the distributions individually using different colors:

```
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*:

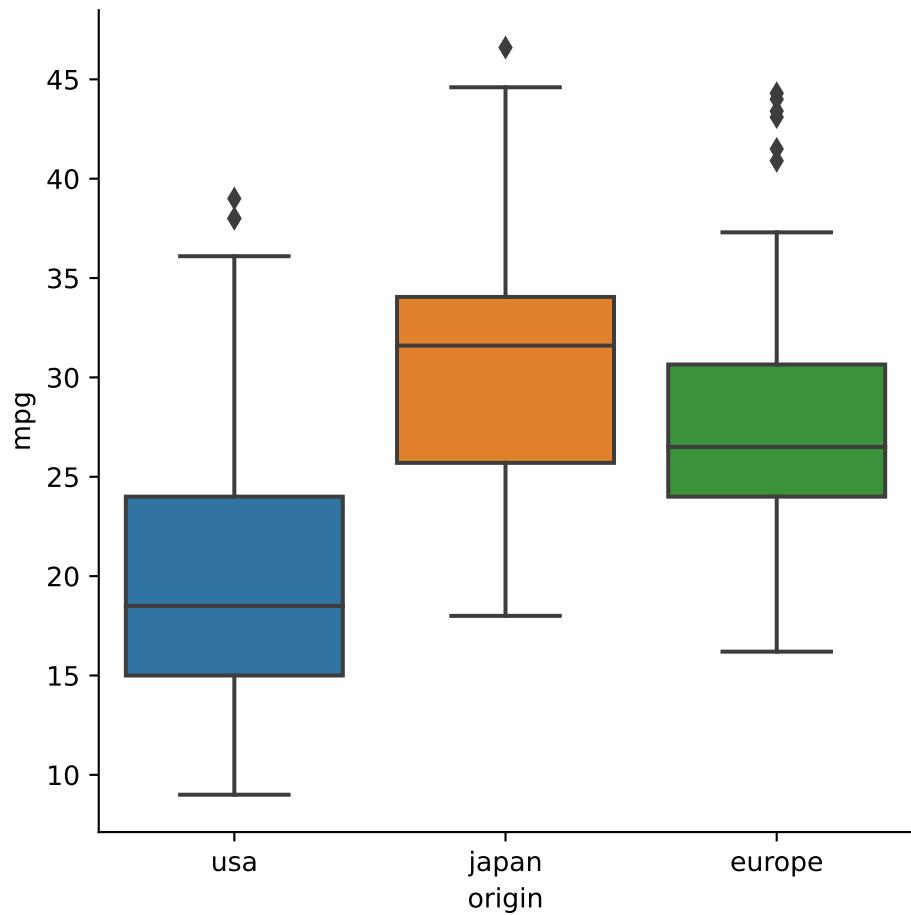
```
sns.displot(data=cars,
    x="mpg",
    col="origin", height=2.2
);
```



It's now clear that the U.S.A. cars are more clustered on the left (smaller MPG) than are the Japanese and European cars.

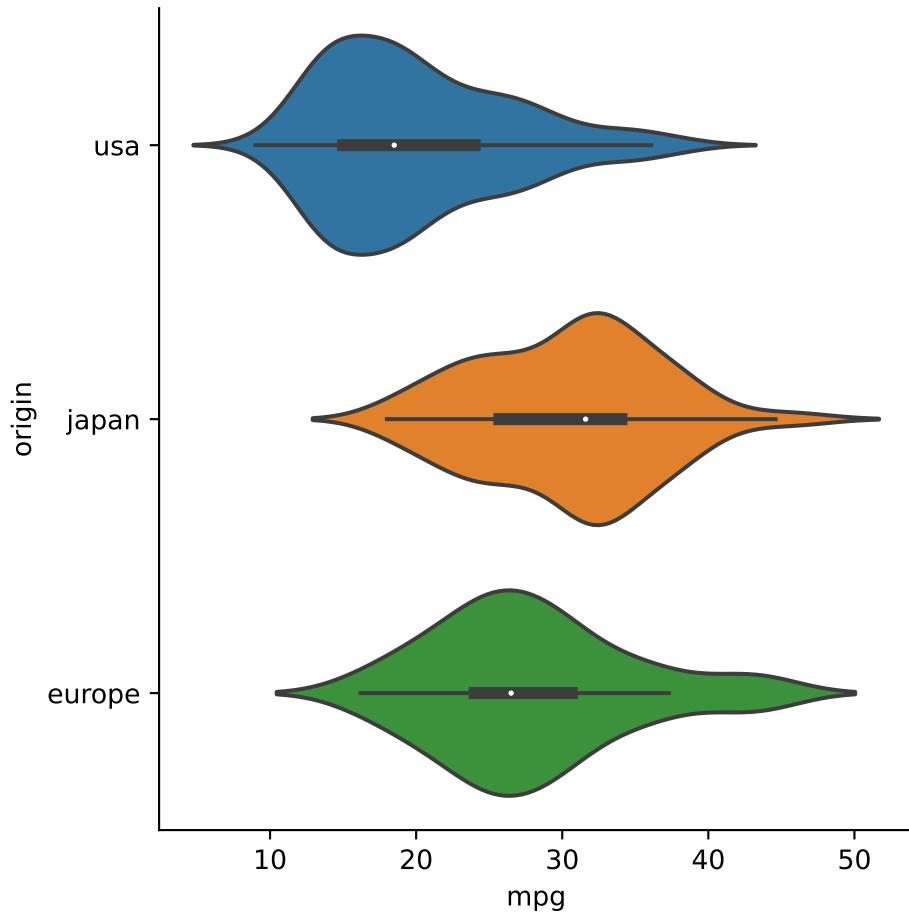
Another way to visualize grouped data is with a **box plot**:

```
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.

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

```

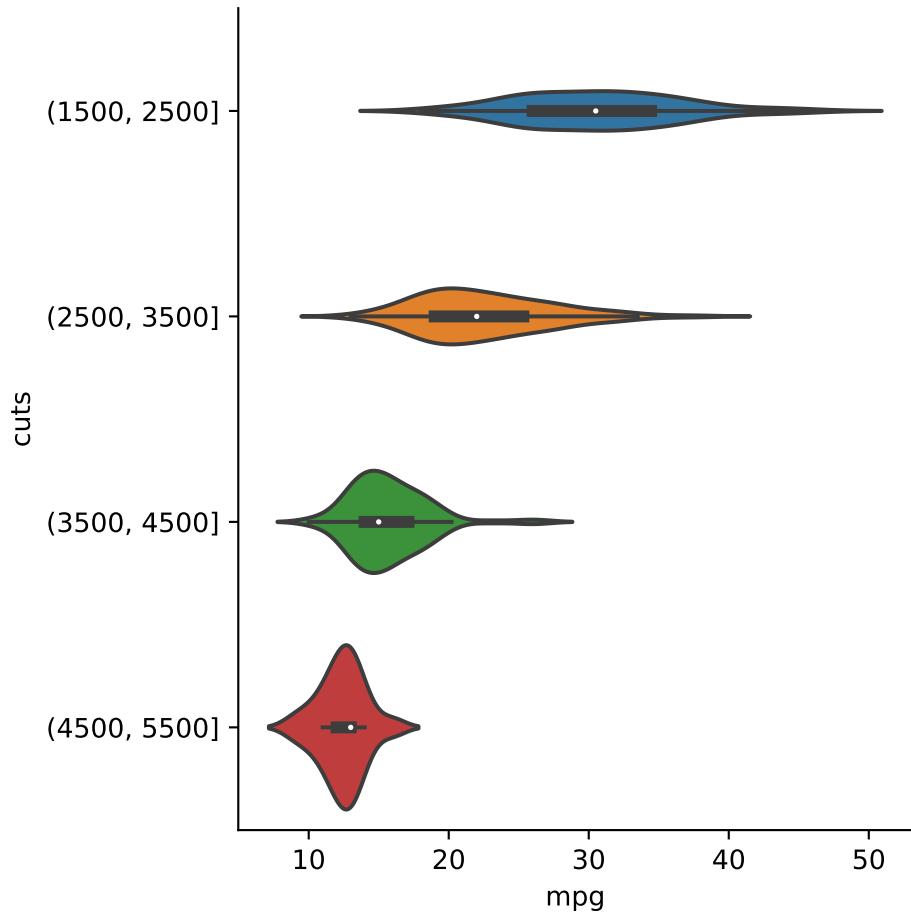
cuts = pd.cut(
    cars["weight"],
    # series to cut by
    range(1500, 5800, 1000)    # bin edges
)

cars["cuts"] = cuts
cars.head(10)

```

	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
6	14.0	8	454.0	220.0	4354	9.0	70	usa	chevrolet imp
7	14.0	8	440.0	215.0	4312	8.5	70	usa	plymouth f
8	14.0	8	455.0	225.0	4425	10.0	70	usa	pontiac cat
9	15.0	8	390.0	190.0	3850	8.5	70	usa	amc ambass

```
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

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

weight	count	mean	std	min	25%	50%	75%	max
(1500, 2500]	147.0	30.631293	5.864413	18.0	26.0	30.5	34.450	46.6
(2500, 3500]	142.0	22.547183	4.754061	13.0	19.0	22.0	25.325	38.0
(3500, 4500]	92.0	15.688043	2.760428	10.0	14.0	15.0	17.125	26.6
(4500, 5500]	17.0	12.411765	1.622453	9.0	12.0	13.0	13.000	16.0

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:

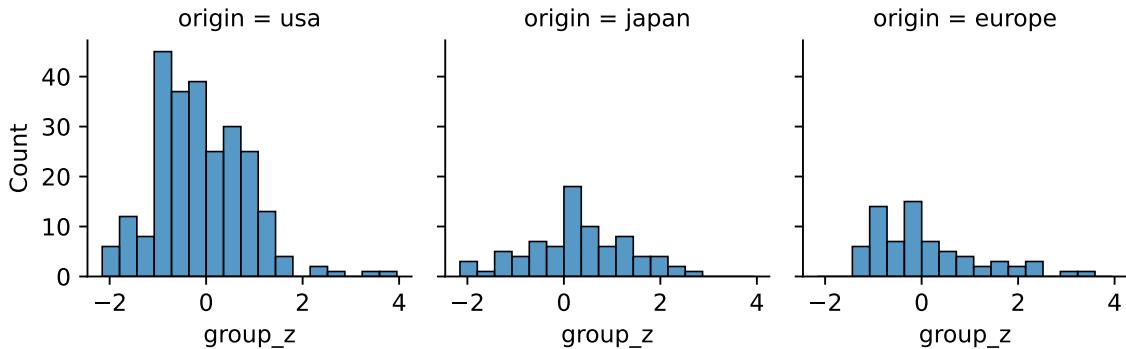
```
def standardize(x):
    return (x - x.mean()) / x.std()

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

```

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

```



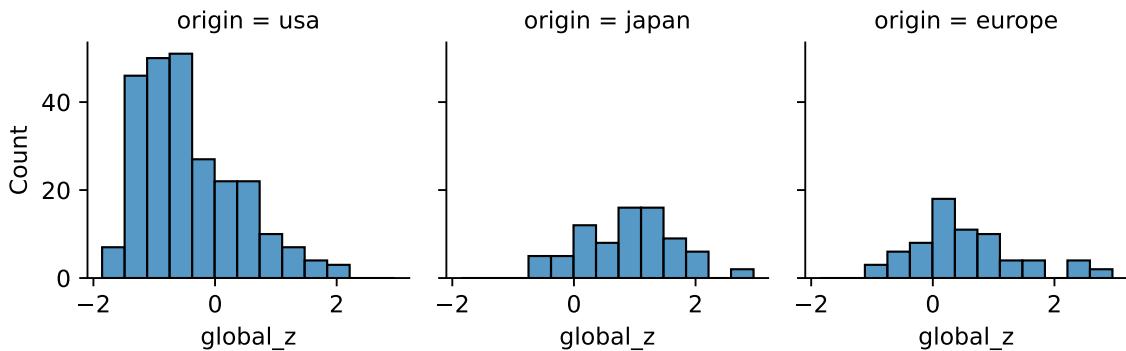
Note how this differs from computing z-scores based on global statistics:

```

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 group-

wise.

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.count()
```

horsepower	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
(40, 60]	20	20	20	20	20	20	20	20	20
(60, 80]	99	99	99	99	99	99	99	99	99
(80, 100]	123	123	123	123	123	123	123	123	123
(100, 120]	48	48	48	48	48	48	48	48	48
(120, 140]	25	25	25	25	25	25	25	25	25
(140, 160]	40	40	40	40	40	40	40	40	40
(160, 180]	20	20	20	20	20	20	20	20	20
(180, 200]	7	7	7	7	7	7	7	7	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

Notice that the result has been merged back into a single frame. If we want to work with the groups again, we have to apply the grouping anew.

```
cuts = pd.cut(hp_30["horsepower"], range(40,220,20))
hp_30.groupby(cuts)[["mpg"]].median()
```

	mpg
horsepower	
(40, 60]	NaN
(60, 80]	31.00
(80, 100]	23.80
(100, 120]	20.40
(120, 140]	NaN
(140, 160]	14.25
(160, 180]	NaN
(180, 200]	NaN

💡 Tip

There is a balance to strike between a plot that is information-poor versus one that is too busy to read clearly. But you can probably fit more information comfortably than you have been accustomed to. Great data visualizations reward time spent by the reader to examine them. [Edward Tufte](#) has written several great books on this subject.

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.

There are various ways of deciding what “typical” means, and there is no one-size recommendation for all applications.

2.4.1 IQR

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*.

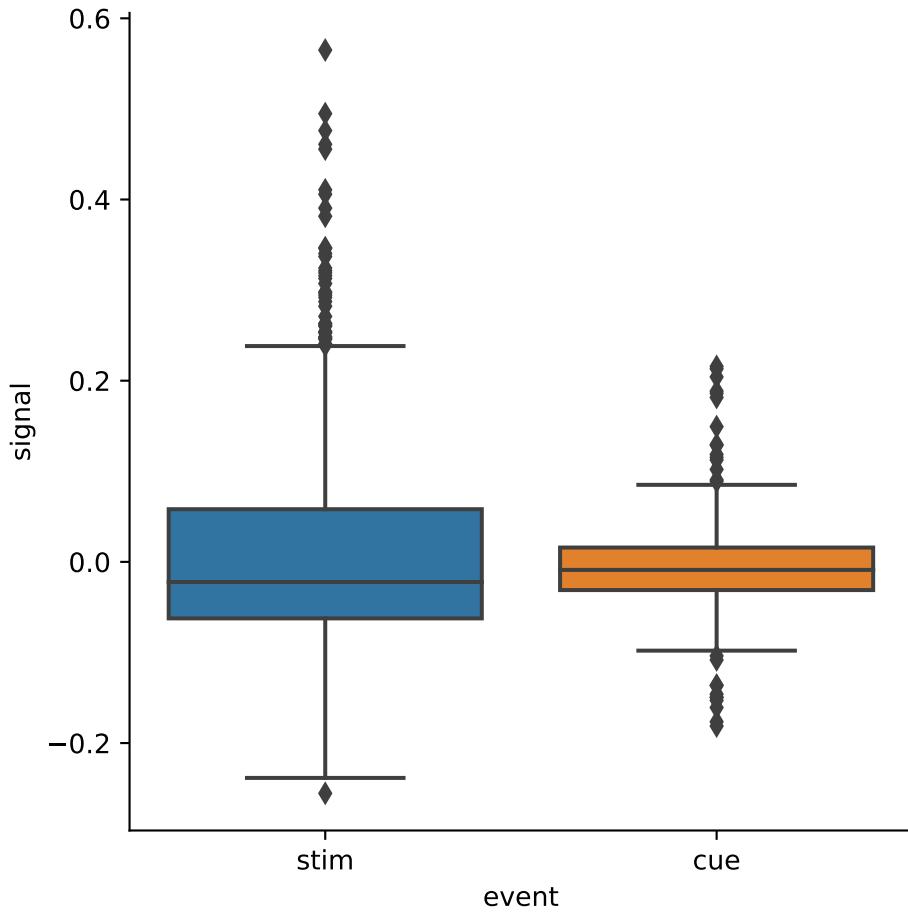
```
fmri.groupby("event")["signal"].describe()
```

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 may be considered outliers. They are determined by the quartiles. Let Q_1 and Q_3 be the first and third quartiles (i.e., 25% and 75% percentiles), and let $I = Q_3 - Q_1$ be the interquartile range (IQR). Then x is an outlier value if

$$x < Q_1 - 1.5I \text{ or } x > Q_3 + 1.5I.$$

2.4.2 Mean and STD

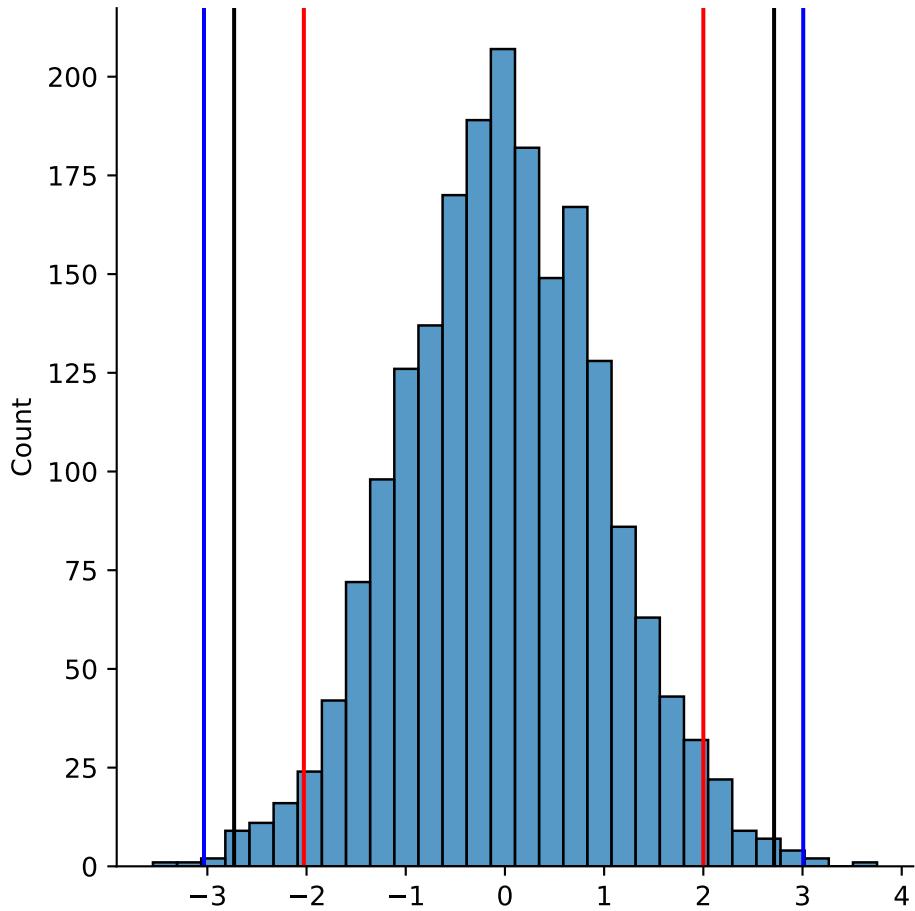
For normal distributions, values more than twice the standard deviation σ from the mean might be declared to be outliers; this would exclude 5% of the values, on average. A less aggressive criterion is to allow a distance of 3σ , which excludes only about 0.3% of the values. The IQR criterion above corresponds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

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.

i Note

A criticism of classical statistics is that much of it is conditioned on the assumption of normally distributed variables. This assumption is often violated by real datasets, and quantities that depend on normality should be used judiciously.

2.4.3 Removing outliers

It is well known 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!

Let's use IQR to remove outliers from the fmri data set. We do this by creating a Boolean-valued series indicating which rows of the frame represent outliers within their group.

```
def is_outlier(x):
    Q1,Q3 = x.quantile([.25,.75])
    I = Q3-Q1
    return (x < Q1-1.5*I) | (x > Q3+1.5*I)

outs = fmri.groupby("event")["signal"].transform(is_outlier)
fmri[outs]["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[~outs]
```

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

```
print( "medians with outliers:" )
print( fmri.groupby("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 show much greater change.

```
print( "means with outliers:" )
print( fmri.groupby("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.)

2.5 Correlation

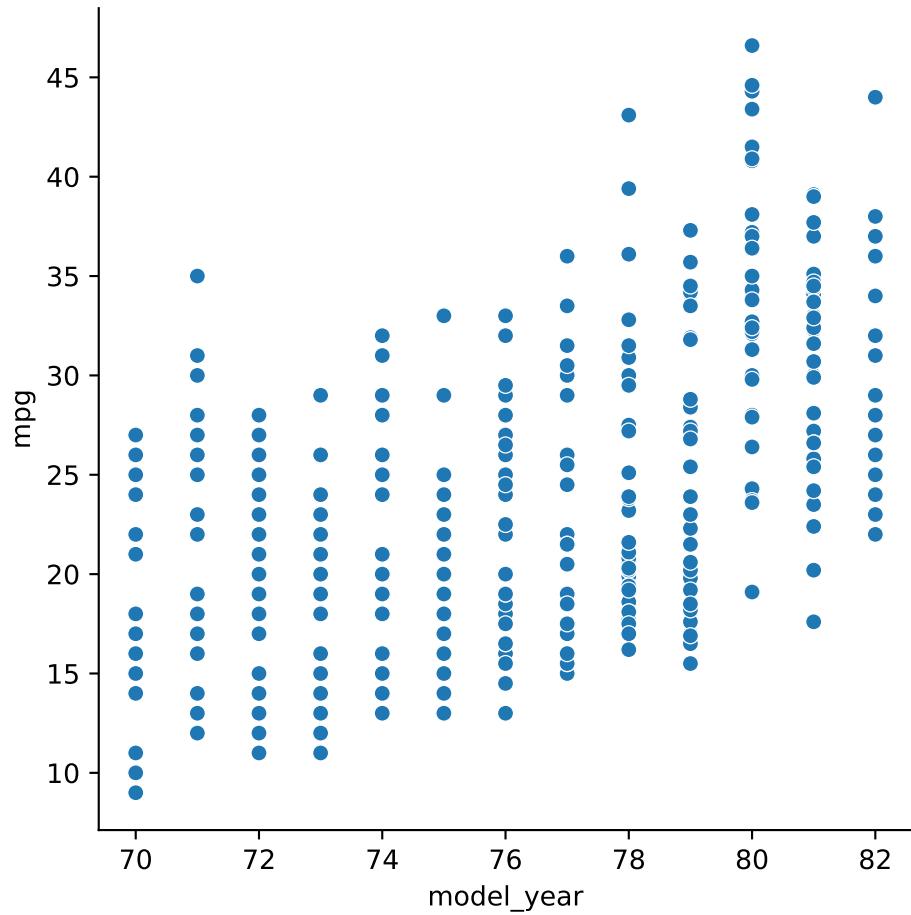
There are often observations that we believe to be linked, either because one influences the other, or both are influenced by some other factor. That is, we say the quantities are *correlated*.

There are several ways to measure correlation, but it's good practice to look at the data 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:

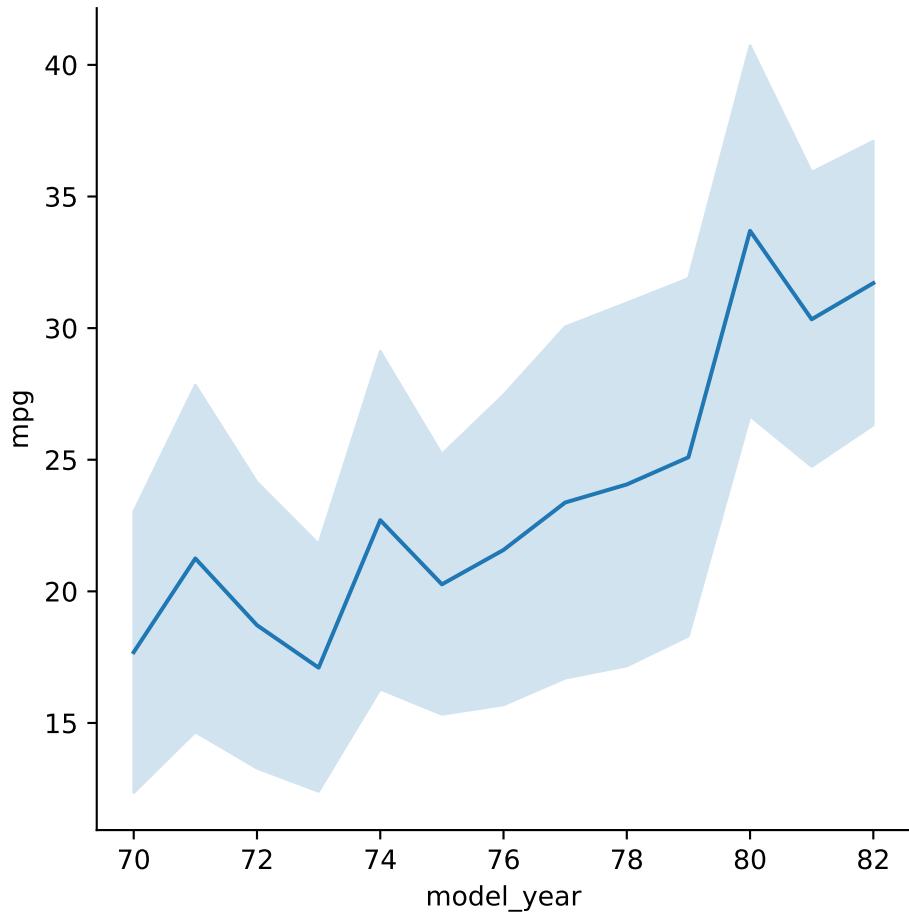
```
sns.relplot(data=cars,
             x="model_year", y="mpg"
            );
```



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:

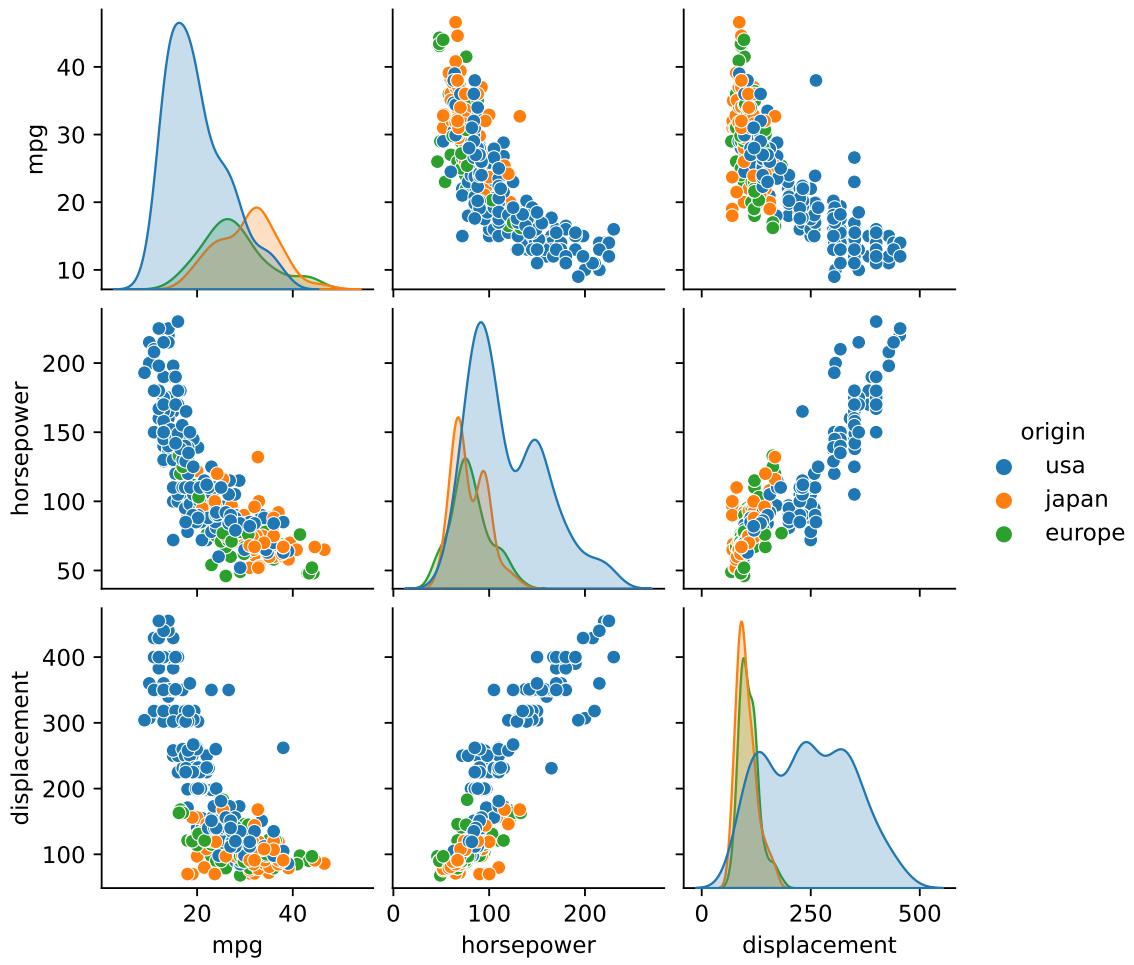
```
sns.relplot(data=cars,
             x="model_year", y="mpg",
             kind="line", errorbar="sd"
            );
```



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 μ_X and μ_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

$$\begin{aligned}\rho(X, Y) &= \frac{1}{n} \sum_{i=1}^n \left(\frac{x_i - \mu_X}{\sigma_X} \right) \left(\frac{y_i - \mu_Y}{\sigma_Y} \right) \\ &= \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y},\end{aligned}\tag{2.9}$$

where σ_X^2 and σ_Y^2 are the population variances of X and Y .

For samples from the two populations, we use

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}},\tag{2.10}$$

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

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

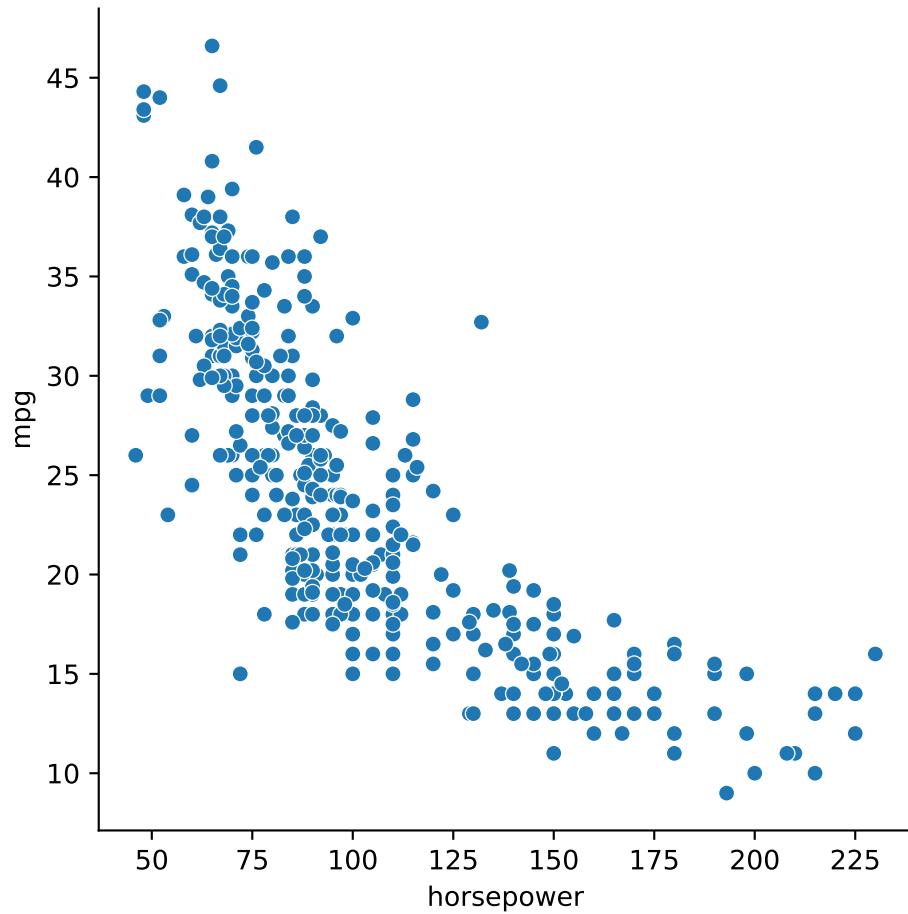
An equivalent formula for r_{xy} is

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

where the quantities in parentheses are z-scores.

Example 2.14. 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 2.15. 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 :

```
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 2.16. Returning to Example 2.15, 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

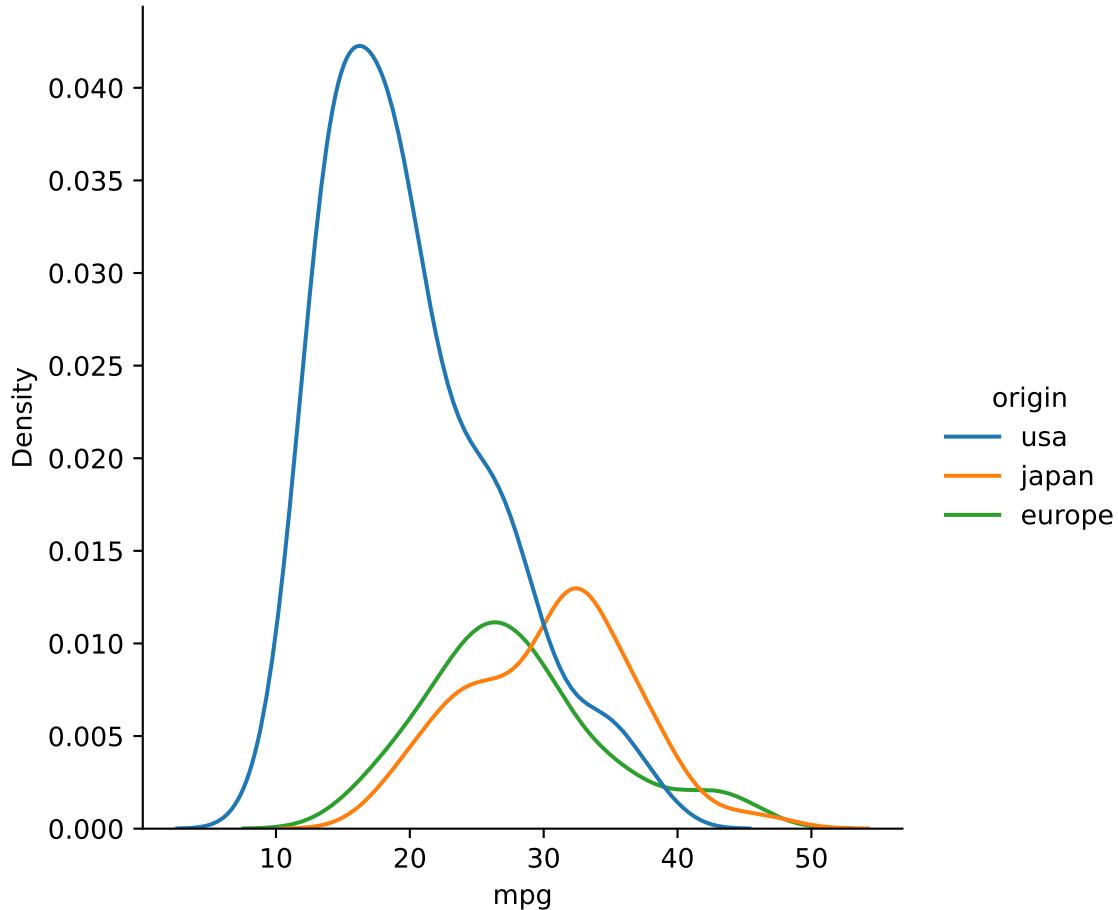
Since real data almost always features outlying or anomalous values, it's important to think about the robustness of the statistics you choose.

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 2.17. 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

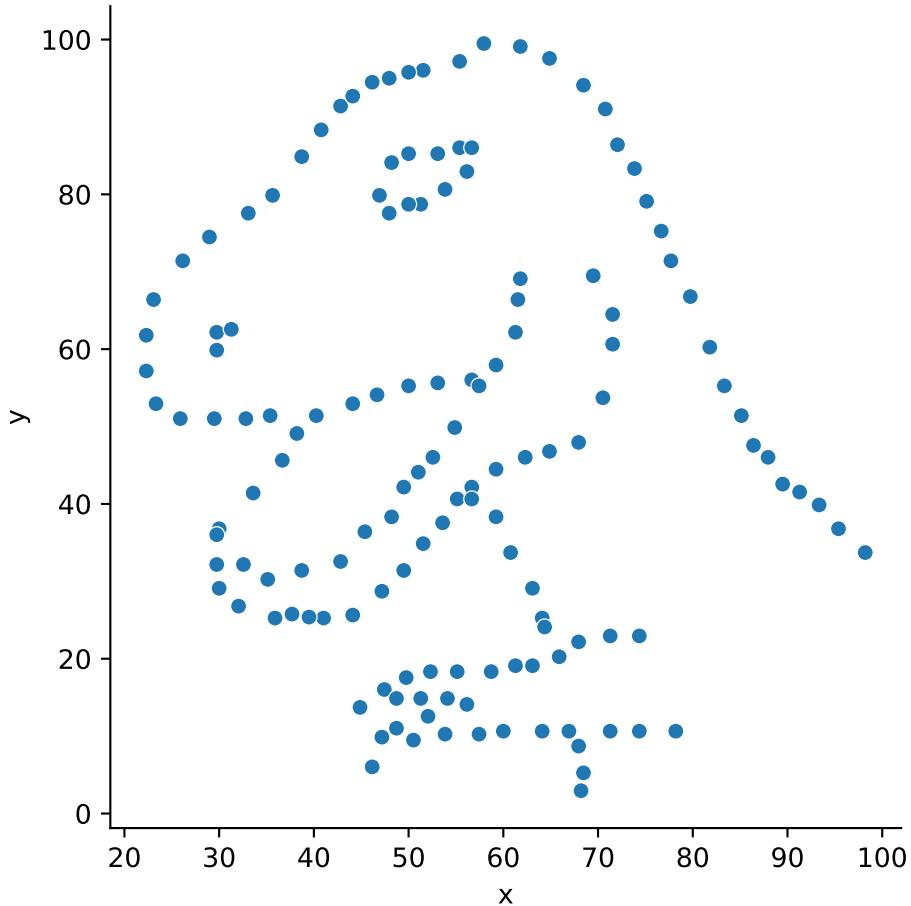
Attaching theorem-supported numbers to real data feels precise and infallible. The theorems do what they say, of course—they’re theorems—but our intuition can be a little too ready to attach significance to the numbers, causing misconceptions or mistakes. Proper visualizations can help us see through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

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()`

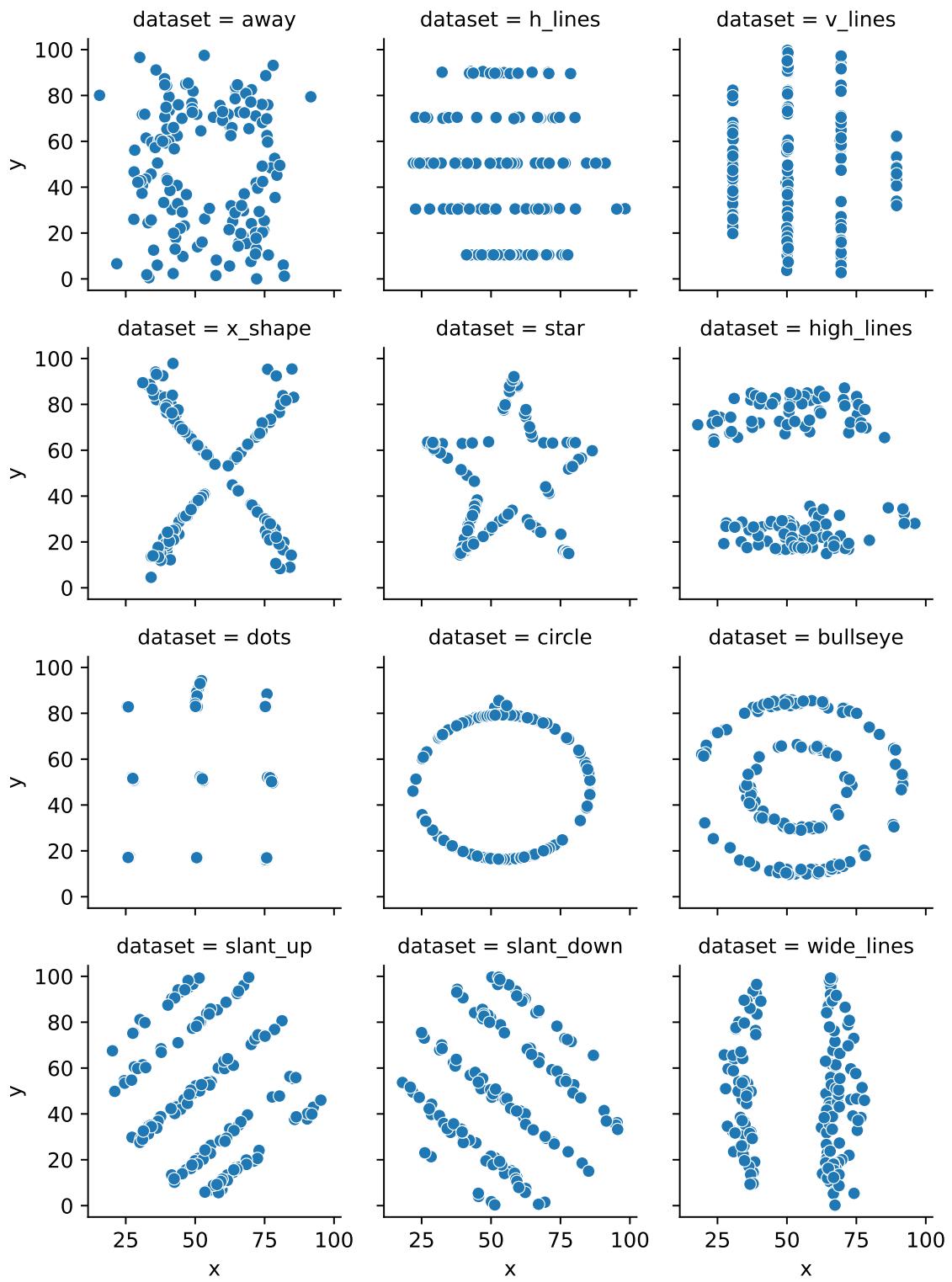
dataset	x	y
away	16.769825	26.939743
bullseye	16.769239	26.935727
circle	16.760013	26.930036
dino	16.765142	26.935403
dots	16.767735	26.930192
h_lines	16.765898	26.939876
high_lines	16.766704	26.939998
slant_down	16.766759	26.936105
slant_up	16.768853	26.938608
star	16.768959	26.930275
v_lines	16.769959	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 Simpson's paradox

The penguins dataset contains a common paradox (well, a counterintuitive phenomenon, anyway). 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.

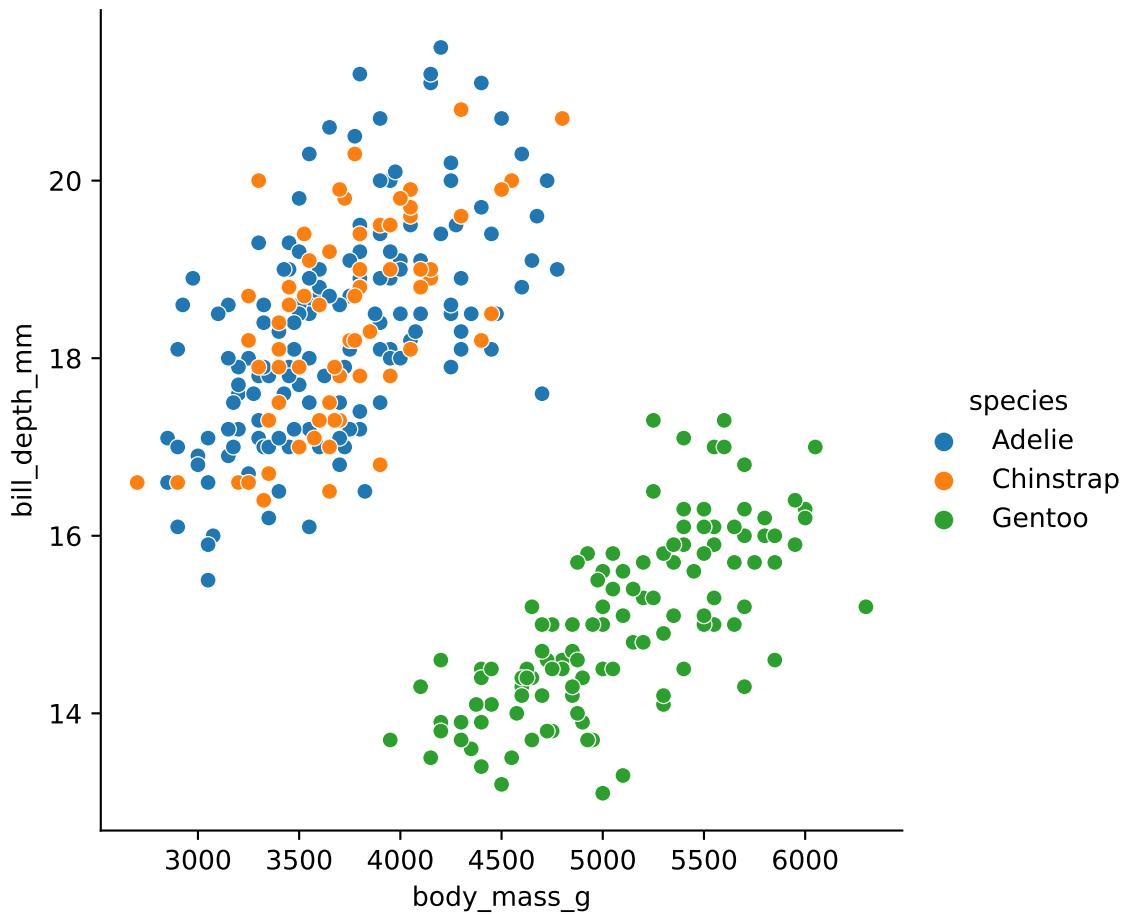
```
penguins.groupby("species")[columns].corr()
```

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

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. For $n > 2$, let $x_i = 0$ for $i = 1, \dots, n - 1$, and $x_n = M > 0$. Find the **(a)** sample mean, **(b)** sample median, **(c)** corrected sample variance s_{n-1}^2 , and **(d)** sample z-scores of the x_i in terms of M and n .

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

(a) Show that $\sum_{i=1}^n z_i = 0$.

(b) Show that $\sum_{i=1}^n z_i^2 = n - 1$.

Exercise 2.3. For the sample set in Exercise 2.1, find a value N such that if $n > N$, there is at least one outlier according to the 2 criterion.

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) Which of the following are outliers according to the 1.5 IQR criterion?

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

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 that r_2 has a global minimum 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 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 piece of the real axis between the x_i , and explain why 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 appealing to the derivative 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.

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 200×9 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. It includes a few classic example datasets. We will load one derived from automatic recognition of handwritten digits.

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

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

It happens that the 64 features are the pixel grayscale values from an 8×8 bitmap of a handwritten digit. The labels are the integer values 0 through 9, indicating the true value of the digit.

Let's consider the binary problem, “is this digit a 6?” That implies the following label vector:

```
y = (dig == 6)
print("Number of sixes in training set:", sum(y))
```

```
Number of sixes in training set: 181
```

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("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
```

💡 Tip

Scikit-learn works as seamlessly with pandas data frames as it does with numerical arrays. In an application, it's often worthwhile to refer to quantities by memorable names, rather than relying on numerical indexes into a matrix.

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_tr, X_te, y_tr, y_te = train_test_split(X, y,
    test_size=0.2,
    shuffle=True,
    random_state=3
)
```

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

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_tr).describe() )

print("\nlabels in the testing set:")
print( pd.Series(y_te).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_tr, y_tr)    # fit only to train set
```

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

...and test on the rest.

```
acc = knn.score(X_te, y_te)      # 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 import metrics
generous = [True]*len(y_te)
acc = metrics.accuracy_score(y_te, 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”

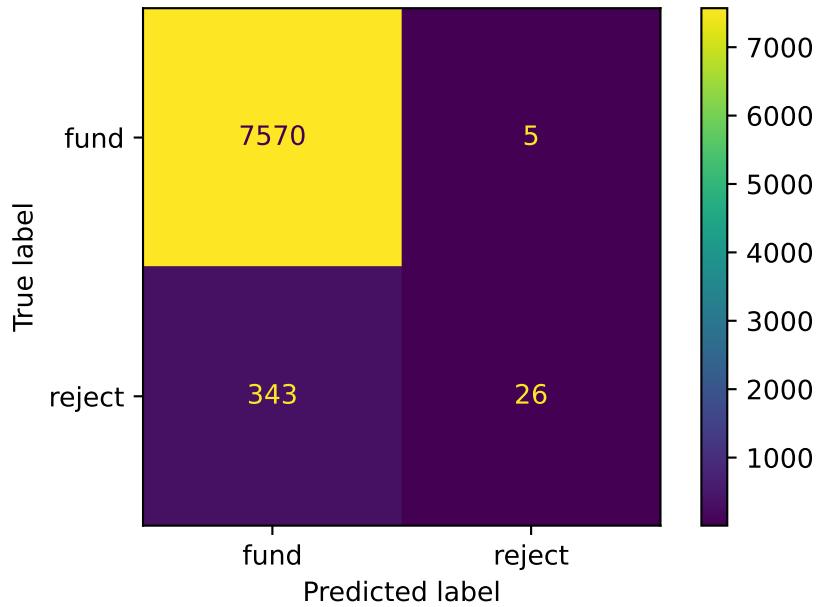
		prediction		
		yes	no	
ground truth	yes	TP	FN	recall
	no	FP	TN	specificity
		precision	NPV	

Figure 3.1: Confusion matrix

The four cases correspond to a 2×2 table according to the states of the prediction and *ground truth*, which is the accepted correct value (i.e., the given label). The table can be filled with counts or percentages of tested instances, to create a **confusion matrix**, as illustrated in Figure 3.1.

`sklearn` makes it straightforward to compute a confusion matrix:

```
yhat = knn.predict(X_te)
C = metrics.confusion_matrix(y_te, yhat, labels=[True, False])
lbl = ["fund", "reject"]
metrics.ConfusionMatrixDisplay(C, display_labels=lbl).plot();
```



🔥 Danger

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

Hence, there are 7570 true positives. Therefore, the **accuracy** is

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} = \frac{7570}{7944} \approx 0.95292$$

i.e., 95.3%. However, there are four other useful quantities defined by putting a "number correct" value in the numerator and the sum of a confusion matrix row or column in the denominator:

$$\text{recall (aka sensitivity)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{negative predictive value (NPV)} = \frac{\text{TN}}{\text{TN} + \text{FN}}$$

In words, these 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?

For our loan classifier, here are the 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%
```

The high recall rate means that few who ought to get a loan will go away disappointed. However, the low specificity would be concerning to those providing the funds, because almost all of those who should be rejected will be approved by the classifier.

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 of the labels corresponds to a “positive” result. Swapping the “positive” role effectively swaps recall with specificity, and precision with NPV.

```
for pos in [True,False]:
    print("With", pos, "as positive:")
    s = metrics.recall_score(y_te, yhat, pos_label=pos)
```

```

print(f"    recall is {s:.3f}")
s = metrics.precision_score(y_te, 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 several ways to combine the measures above into a single value. None is universally best, because different applications emphasize different aspects of performance. One of the most popular is the **F score**, which is the harmonic mean of the precision and the recall:

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

This score varies between 0 (poor) and 1 (ideal). If one of the quantities is much smaller than the other, their harmonic mean will be close to the small value. Thus, F score punishes a classifier if either recall or precision is poor.

Another composite score is **balanced accuracy**, which is the arithmetic mean of recall and specificity. It also ranges from 0 to 1, with 1 meaning perfect accuracy.

```

print("F1:", metrics.f1_score(y_te, yhat))
print("Balanced:", metrics.balanced_accuracy_score(y_te, yhat))

```

```

F1: 0.9775309917355371
Balanced: 0.5349003193002227

```

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

Example 3.2. If k of the n testing samples were funded loans, then the fund-them-all loan classifier has

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

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 $K \times K$.

It's easiest to see how this works by an example. We will 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()
```

	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

```
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

```

X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True,
    random_state=1
)
knn = KNeighborsClassifier(n_neighbors=8)
knn.fit(X_tr, y_tr)
yhat = knn.predict(X_te)
print(f"accuracy is {metrics.accuracy_score(y_te, yhat):.1%}")

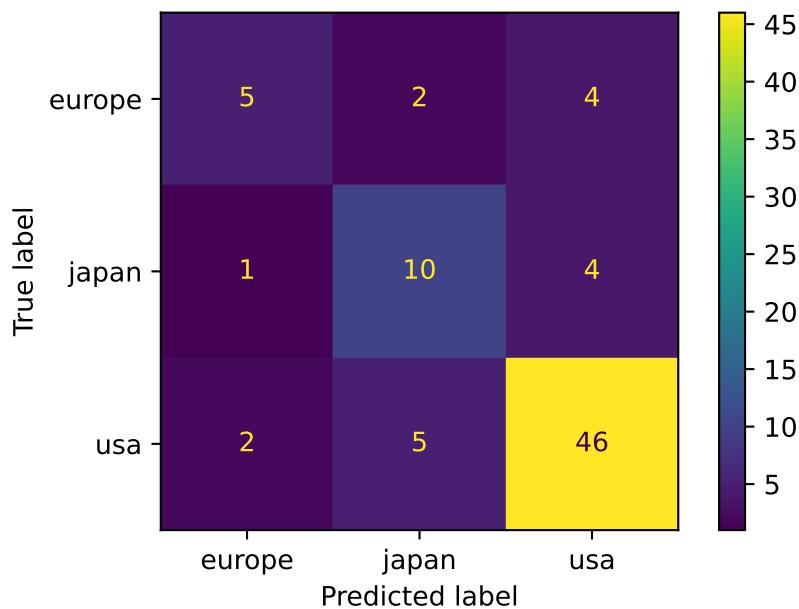
```

accuracy is 77.2%

```

labels = y.categories
C = metrics.confusion_matrix(y_te, yhat, labels=labels)
metrics.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:

```

prec = metrics.precision_score(y_te, yhat, average=None)
for (i,p) in enumerate(prec):
    print(f"{labels[i]}\t{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 = metrics.precision_score(y_te, yhat, average="macro")
print(mac)

```

0.6883623819898329

There are other ways to perform the averaging, depending on whether poorly represented classes should be weighted more weakly than others.

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) 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 subsets whose labels are as uniform as possible. The process is then repeated recursively on the subsets. Defining a measurement of label uniformity is a key step.

3.3.1 Gini impurity

Let S be a subset of the 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

$$p_k = \frac{1}{|S|} \sum_{i \in S} \mathbb{1}_k(y_i),$$

where $|S|$ is the number of elements in S and $\mathbb{1}_k$ is the **indicator function**

$$\mathbb{1}_k(t) = \begin{cases} 1, & \text{if } t = k, \\ 0, & \text{otherwise.} \end{cases}$$

In words, p_k is the proportion of samples in S that have label k . Then the **Gini impurity** is defined as

$$H(S) = \sum_{k=1}^K p_k(1 - p_k).$$

If one of the p_k is 1, then the others are all zero and $H(S) = 0$. This is considered optimal. At the other extreme, if $p_k = 1/K$ for all k , then

$$H(S) = \sum_{k=1}^K \frac{1}{K} \left(1 - \frac{1}{K}\right) = K \cdot \frac{1}{K} \cdot \frac{K-1}{K} = \frac{K-1}{K} < 1.$$

Example 3.3. Suppose a set S has n members with label 1, 1 member with label 2, and 1 member with label 3. What is the Gini impurity of S ?

Solution. We have $p_1 = n/(n+2)$, $p_2 = p_3 = 1/(n+2)$. 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

Now we can describe the partition process. If j is a dimension (feature) number and θ is a numerical threshold, then the sample set can be partitioned into complementary sets S_L , in which $x_j \leq \theta$, and S_R , in which $x_j > \theta$. Define the **total impurity** of the partition to be

$$Q(j, \theta) = |S| H(S) + |T| H(T).$$

Choose the (j, θ) that minimize Q , and then recursively partition S and T .

Example 3.4. Suppose the 1D real samples $x_i = i$ for $i = 0, 1, 2, 3$ have labels A,B,A,B. What is the optimal partition?

Solution. There are three ways to partition them.

- $S = \{0\}, T = \{1, 2, 3\}$. We have $H(S) = 0$ and $H(T) = (2/3)(1/3) + (1/3)(2/3) = 4/9$. Hence the total impurity for this partition is $(1)(0) + (3)(4/9) = 4/3$.
- $S = \{0, 1\}, T = \{2, 3\}$. Then $H(S) = H(T) = 2(1/2)(1/2) = 1/2$, and the total impurity is $(2)(1/2) + (2)(1/2) = 2$.
- $S = \{0, 1, 2\}, T = \{3\}$. This arrangement is the same as the first case with $A \leftrightarrow B$.

The best partition threshold is $x \leq 0$ (or $x \leq 2$, which is equivalent).

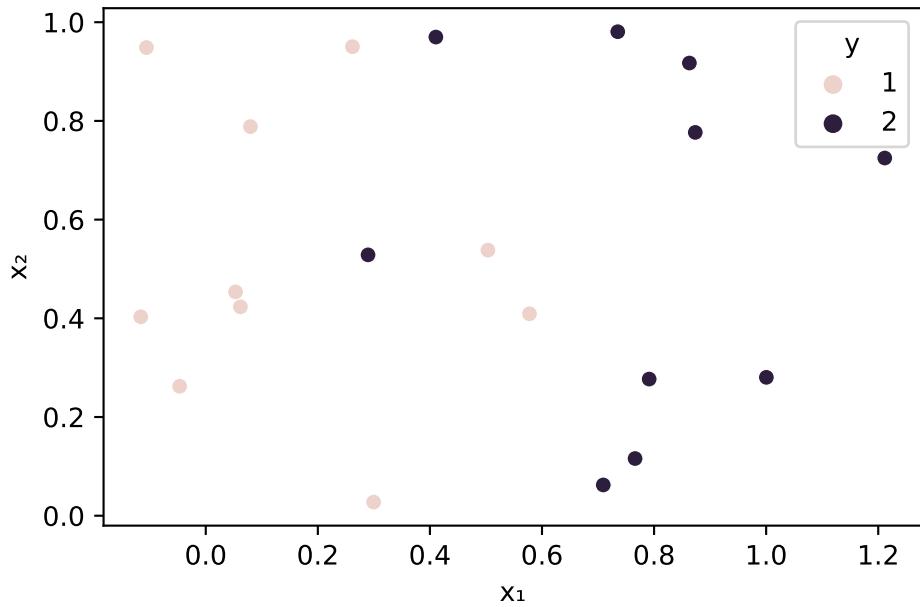
3.3.3 Toy example

We first create a toy dataset with 20 random points, with two subsets of 10 that are shifted left/right a bit.

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

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.hstack(([1]*10,[2]*10))

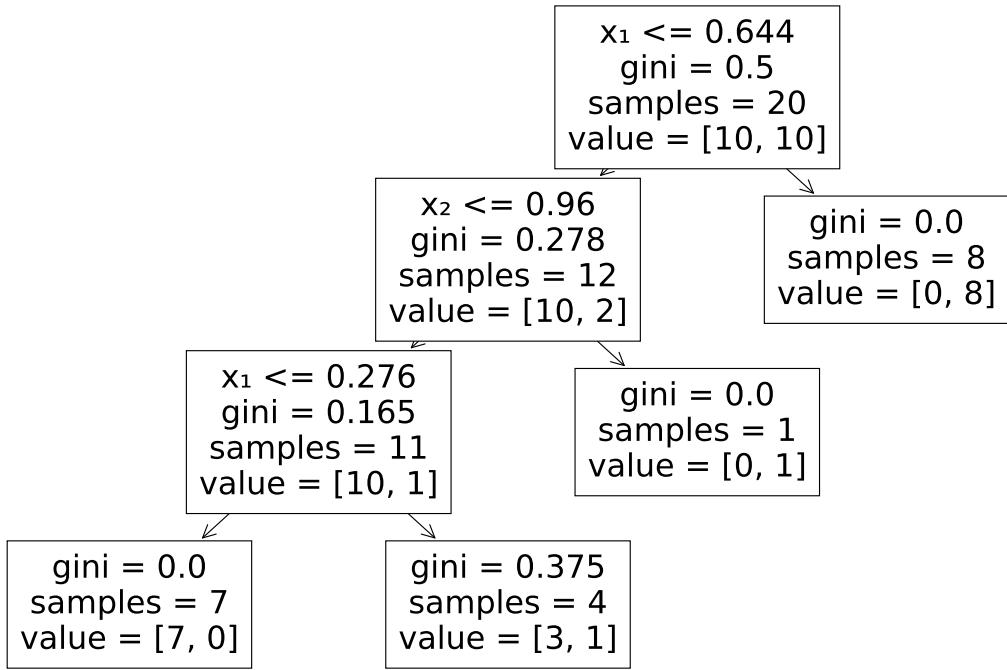
import seaborn as sns
df = pd.DataFrame( {"x":X[:,0], "x":X[:,1], "y":y} )
sns.scatterplot(data=df, x="x", y="x", hue="y");
```



Now we create a decision tree for these samples.

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
t = DecisionTreeClassifier(max_depth=3)
t.fit(X,y)

from matplotlib.pyplot import figure
figure(figsize=(18,11), dpi=160)
plot_tree(t, feature_names=["x ", "x "]);
```



The root of the tree (at the top) shows that the best 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. Thus, any future prediction by this tree will immediately return label 2 if the first feature of the input exceeds 0.644. Otherwise, it moves to the left child node and tests whether the second feature is greater than 0.96. This splits along a horizontal line, above which there is a single sample with label 2. And so on.

Notice that the bottom right node has a nonzero 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 the 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 numbers of features and observations increase.

3.3.4 Penguin data

We return to the penguins. There is no need to standardize the columns for a decision tree, because each feature is considered on its own.

```

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"]

```

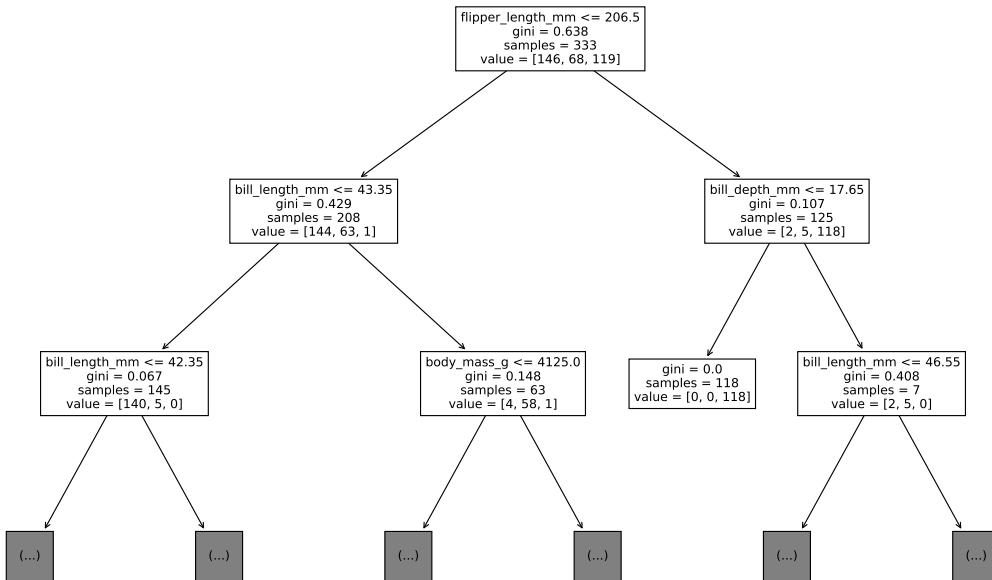
We get some interesting information from looking at the top levels of a decision tree trained on the full dataset.

```

dt = DecisionTreeClassifier(max_depth=4)
dt.fit(X, y)

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 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. This is known as **Gini importance**.

```
pd.Series(dt.feature_importances_, index=features)
```

	0
bill_length_mm	0.381063
bill_depth_mm	0.075938
flipper_length_mm	0.529362
body_mass_g	0.013638

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_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True,
    random_state=0
)
dt.fit(X_tr, y_tr)

yhat = dt.predict(X_te)
print( confusion_matrix(y_te, yhat) )
print( classification_report(y_te, yhat) )

[[39  0  0]
 [ 2  8  0]
 [ 0  0 18]]

          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_tr.value_counts()
```

species	
Adelie	107
Gentoo	101
Chinstrap	58

3.3.5 Limitations

Decision trees depend sensitively on the sample locations. A tree trained on one data subset may not do well on a new set. A small change can completely rewrite large parts of the tree, which gives a caveat about interpretation. Also, the partition algorithm, which is *greedy* by doing the best thing at the moment, does not necessarily find a globally optimal tree, or even a nearby one.

3.4 Nearest neighbors

Our first learning algorithm is conceptually simple: Given a new point to classify, survey the nearest known examples and choose the most frequently occurring class. This is called the **k nearest neighbors** (KNN) algorithm, where k is the number of neighboring examples to survey.

3.4.1 Norms

The existence of “closest” examples means that we need to define a notion of distance in spaces of any dimension. Let \mathbb{R}^d be the space of vectors with d real components, and let 0 be the vector of all zeros.

Definition 3.2. A **norm** is a function $\|\mathbf{x}\|$ on \mathbb{R}^d that satisfies the following properties:

$$\begin{aligned}\|0\| &= 0, \\ \|\mathbf{x}\| &> 0 \text{ if } \mathbf{x} \text{ is a nonzero vector,} \\ \|c\mathbf{x}\| &= |c| \|\mathbf{x}\| \text{ for any real number } c, \\ \|\mathbf{x} + \mathbf{y}\| &\leq \|\mathbf{x}\| + \|\mathbf{y}\|\end{aligned}$$

The last inequality above is called the **triangle inequality**. It turns out that these four characteristics are all we expect from a function that behaves like a distance.

On the number line (i.e., \mathbb{R}^1), the distance between two values is just the absolute value of their difference, $|x - y|$. In \mathbb{R}^d , the distance between two vectors is the norm of their difference, $\|\mathbf{x} - \mathbf{y}\|$.

There are three commonly used norms:

- The **2-norm** or Euclidean norm:

$$\|\mathbf{x}\|_2 = (x_1^2 + x_2^2 + \cdots + x_d^2)^{1/2}.$$

- The **1-norm** or Manhattan norm:

$$\|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_d|.$$

- The **∞ -norm** or max norm:

$$\|\mathbf{x}\|_\infty = \max_i |x_i|.$$

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 sometimes it means a generic, unspecified norm.

3.4.2 Algorithm

As data, we are given labeled examples $\mathbf{x}_1, \dots, \mathbf{x}_n$ in \mathbb{R}^d . Given a new query vector \mathbf{x} , find the k labeled vectors closest to \mathbf{x} and choose the most frequently occurring label among them. Ties can be broken randomly.

KNN divides up the feature space into domains that are dominated by nearby instances. The boundaries between those domains, called **decision boundaries**, can be fairly complicated, though, as shown in the animation below.

[_media/knn_demo.mp4](#)

Implementation of KNN is straightforward for small data sets, but requires care to get reasonable execution efficiency for large sets.

3.4.3 KNN in sklearn

Let's revisit 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)
```

	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_tr, X_te, y_tr, y_te = train_test_split(X,y,test_size=0.2)
knn.fit(X_tr,y_tr)

yhat = knn.predict(X_te)
print(confusion_matrix(y_te,yhat))
print(classification_report(y_te,yhat))

[[22  1  1]
 [12  3  0]
 [ 2  0 26]]

```

	precision	recall	f1-score	support
Adelie	0.61	0.92	0.73	24
Chinstrap	0.75	0.20	0.32	15
Gentoo	0.96	0.93	0.95	28
accuracy			0.76	67
macro avg	0.77	0.68	0.66	67
weighted avg	0.79	0.76	0.73	67

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.

3.4.4 Standardization

The values in the columns of the penguin frame 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 can transform the data into z-scores:

```
Z = X.transform( lambda x: (x - x.mean()) / x.std() )
```

In this instance, standardization makes performance dramatically better:

```
Z_tr, Z_te, y_tr, y_te = train_test_split(Z,y,test_size=0.2)
knn.fit(Z_tr,y_tr)

yhat = knn.predict(Z_te)
print(confusion_matrix(y_te,yhat))
print(classification_report(y_te,yhat))
```

```
[[31  0  0]
 [ 1 15  0]
 [ 0  0 20]]

      precision    recall  f1-score   support

     Adelie      0.97     1.00     0.98      31
 Chinstrap      1.00     0.94     0.97      16
    Gentoo      1.00     1.00     1.00      20

  accuracy                           0.99      67
  macro avg       0.99     0.98     0.98      67
weighted avg       0.99     0.99     0.98      67
```

3.4.5 Pipelines

One nuisance of the standardization step above is that it must be performed again for any new query vector that comes along. This means we need to keep track of the mean and std of the original training set.

Scikit-learn allows us to create a **pipeline** that automates the transformation. Pipelines make it easy to chain together data transformations followed by a learner. The composite object acts much like a regular learner.

As you might guess, standardization of data is so common that it is predefined:

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler    # converts to z-scores

X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True,
    random_state=0
)

knn = KNeighborsClassifier(n_neighbors=5)
pipe = make_pipeline(StandardScaler(), knn)

pipe.fit(X_tr, y_tr)
pipe.score(X_te, y_te)
```

0.9701492537313433

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_ )

[ 44.49774436  17.13496241 201.80827068 4253.28947368]
[2.86849573e+01 4.06520620e+00 1.88493315e+02 6.27702995e+05]
```

3.5 Quantifying votes

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 3.3. 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 fraction received by each class.

Example 3.5. 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 5×3 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]
] )
```

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 the information we get is upstream in the process.

Example 3.6. 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_tr, X_te, y_tr, y_te = 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_tr, y_tr)
p_hat = knn.predict_proba(X_te)
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_te)
yhat[:6]

array(['Adelie', 'Adelie', 'Gentoo', 'Gentoo', 'Adelie', 'Adelie'],
      dtype=object)

```

3.5.1 ROC curve

Having values between the classes means that we can fine-tune how we decide to assign them.

Definition 3.4. Let θ be a number in the interval $[0, 1]$. We say that a class T **hits** at level θ at a query point if the fraction of votes that T receives at that point is at least θ .

Example 3.7. Continuing with the data in Example 3.5, 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 θ 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.

The **ROC curve**, or *receiver operator characteristic* curve, is a way to visualize the hits as a function of θ over a fixed testing set. The name is a little misleading, since the multiclass case requires multiple curves. The idea is to tally, at each value of θ , all the hits within each class that represent true positives and false positives.

i Note

The name of the ROC curve is a throwback to the early days of radar, when the idea was first developed.

Example 3.8. We continue with the data from Example 3.5, 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 θ 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

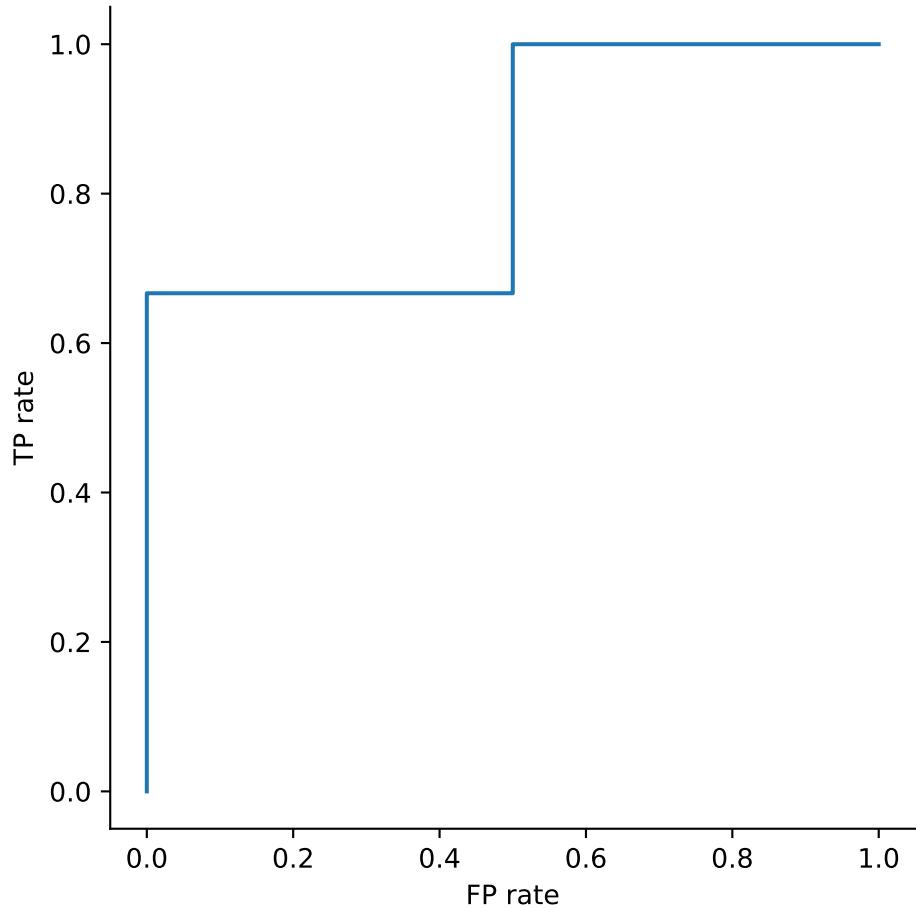
$$\left(\frac{2}{2}, \frac{3}{3}\right), \left(\frac{1}{2}, \frac{3}{3}\right), \left(\frac{1}{2}, \frac{2}{3}\right), \left(\frac{0}{2}, \frac{2}{3}\right), \left(\frac{0}{2}, \frac{1}{3}\right) \left(\frac{0}{2}, \frac{0}{3}\right).$$

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
            );

```



Unsurprisingly, `sklearn` can compute the points defining the ROC curve automatically, which greatly simplifies drawing them.

Example 3.9. Continuing Example 3.6, 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_te)
results = []
for i, label in enumerate(knn.classes_):
    actual = (y_te==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

```

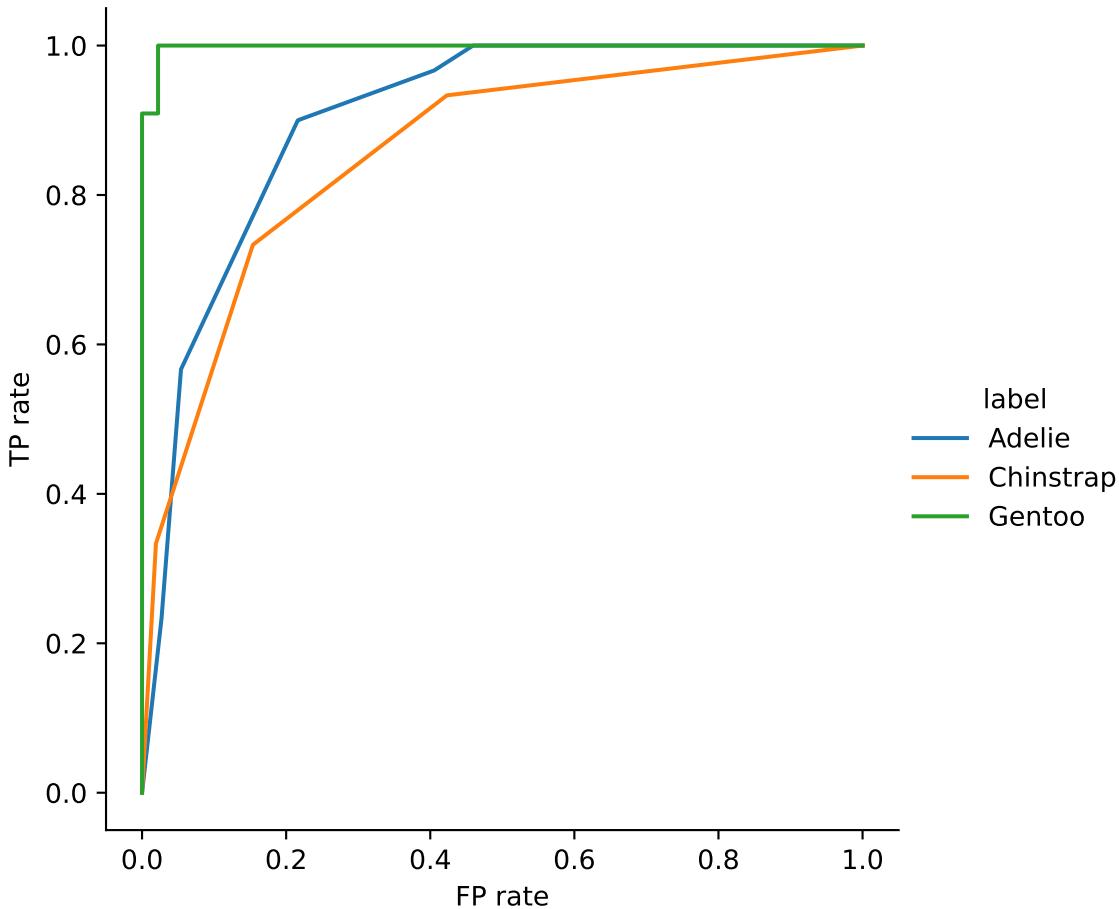
	label	FP rate	TP rate
0	Adelie	0.000000	0.000000
1	Adelie	0.027027	0.233333
2	Adelie	0.054054	0.566667
3	Adelie	0.216216	0.900000
4	Adelie	0.405405	0.966667
5	Adelie	0.459459	1.000000
6	Adelie	1.000000	1.000000
7	Chinstrap	0.000000	0.000000
8	Chinstrap	0.019231	0.333333
9	Chinstrap	0.153846	0.733333
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_tr, y_tr)
p_hat = knnw.predict_proba(X_te)

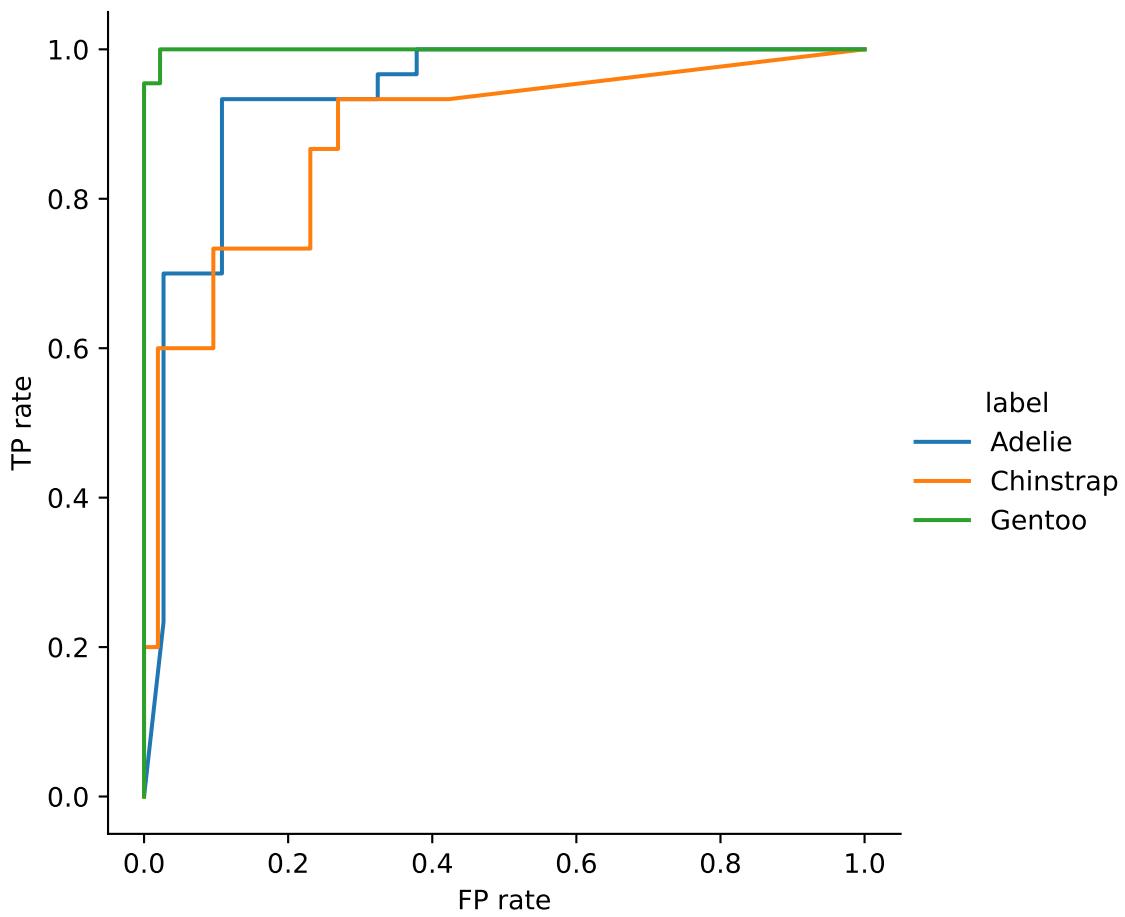
results = []
for i, label in enumerate(knn.classes_):
    actual = (y_te==label)
    fp, tp, theta = roc_curve(actual,p_hat[:,i])

```

```

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 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 3.10. The AUC metric allows us to compare the standard and weighted kNN classifiers from Example 3.9. 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_te, knn.predict_proba(X_te),
    multi_class="ovr", average=None
)

sw = roc_auc_score(
    y_te, knnw.predict_proba(X_te),
    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	

prediction

Calculate the (a) recall, (b) precision, (c) specificity, (d) accuracy, and (e) F_1 score of the classifier, where *dank* is the positive outcome.

Exercise 3.2. Here is a confusion matrix for a classifier of ice cream flavors.

	vanilla	75	10	4
<i>true flavor</i>	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. 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 3.5. 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 3.6. 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 of k -nearest neighbors with $k = 1$, using the 2-norm.

Exercise 3.7. Define 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 $\theta_k = 2k\pi/8$ for $k = 0, 1, \dots, 7$. Show that if the x_k and y_k are standardized into z-scores, then the resulting points all lie on a circle centered at the origin. (Standardizing points into z-scores is sometimes called *sphereing* them.)

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
```

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*.

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. On the other hand, the 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 the best-case scenario of a training set that is perfectly representative, it's more prudent to consider what's best 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 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:

$$\mathbb{E} [\hat{f}(x)] = \frac{1}{N} \sum_{i=1}^N \hat{f}_i(x).$$

i 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 by averaging over all parallel universes that received 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. Each architecture can reproduce a function of limited complexity.

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 ring toss game at a carnival. You have an array of sticks on a grid laid out horizontally in front of you, and the goal is to toss rings so that they land surrounding any one of the sticks.

One strategy is to aim for the middle of the grid, because missing in any direction still gives you a chance to score. This is like aiming directly for the mean position. However, it's likely that you have more consistency with shorter throws than longer ones: the farther away your aiming point is, the more variation there will be in the landing spots. Hence, it might be superior to aim at a closer stick. Even though throws that are too short may have no chance at scoring, you will do better due to the decreased spread of your throws.

In essence, you have only a finite—probably small—number of throws to make, so the performance of the average case isn't the only consideration. Reducing variance can improve your odds of getting close, even if the average case is mediocre.

To see this tradeoff mathematically, we can compute the variance of the predicted labels at

any x :

$$\begin{aligned}
\mathbb{E}[(y - \hat{y})^2] &= \frac{1}{N} \sum_{i=1}^N (y - \hat{f}_i(x))^2 \\
&= \frac{1}{N} \sum_{i=1}^N (y - \hat{y} + \hat{y} - \hat{f}_i(x))^2 \\
&= \frac{1}{N} \sum_{i=1}^N (y - \hat{y})^2 + \frac{1}{N} \sum_{i=1}^N (\hat{y} - \hat{f}_i(x))^2 \\
&\quad - 2(y - \hat{y}) \cdot \frac{1}{N} \sum_{i=1}^N (\hat{y} - \hat{f}_i(x)).
\end{aligned}$$

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,

$$\begin{aligned}
\mathbb{E}[(y - \hat{y})^2] &= \frac{1}{N} \sum_{i=1}^N (y - \hat{y})^2 + \frac{1}{N} \sum_{i=1}^N (\hat{y} - \hat{f}_i(x))^2 \\
&= (y - \hat{y})^2 + \mathbb{E}[(\hat{y} - \hat{f}(x))^2]
\end{aligned}$$

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?

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

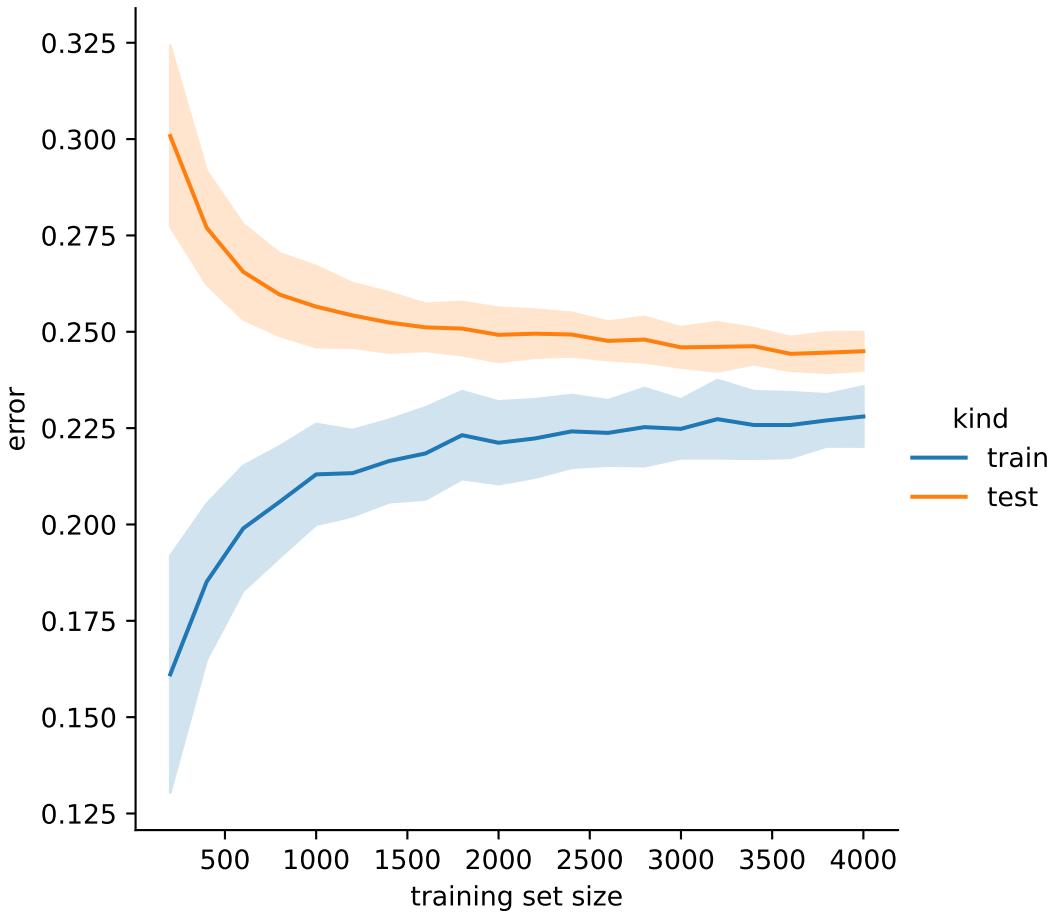
We can illustrate the tradeoff between bias and variance by running an artificial experiment with different sizes for the training datasets.

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_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.05,
    shuffle=True, random_state=0
)

alln = range(200, 4001, 200)      # sizes of the training subsets
results = []                      # for tracking results
tree = DecisionTreeClassifier(max_depth=4)
for n in alln:                    # iterate over training set sizes
    for i in range(50):           # iterate over training sets
        X_tr, y_tr = shuffle(X_tr, y_tr, random_state=i)
        XX, yy = X_tr[:n,:], y_tr[: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_te, y_te)) )

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 a small training set, the tree has more than enough resolving power, and the result is severe overfitting, as seen by the large gap between testing and training. As the size of the training set grows, however, variance decreases and the two error measurements come together.

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 a cosine curve, you can't get it exactly correct no matter how much information you have about it.

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.

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

Consider 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. Let's see how this plays out in about as simple a classification problem as we can come up with: a single feature, with the class being the sign of the feature's value. (We arbitrarily assign zero to have class +1.)

Using $k = 1$ produces fine results, as shown here for 4 different training sets of size 40:

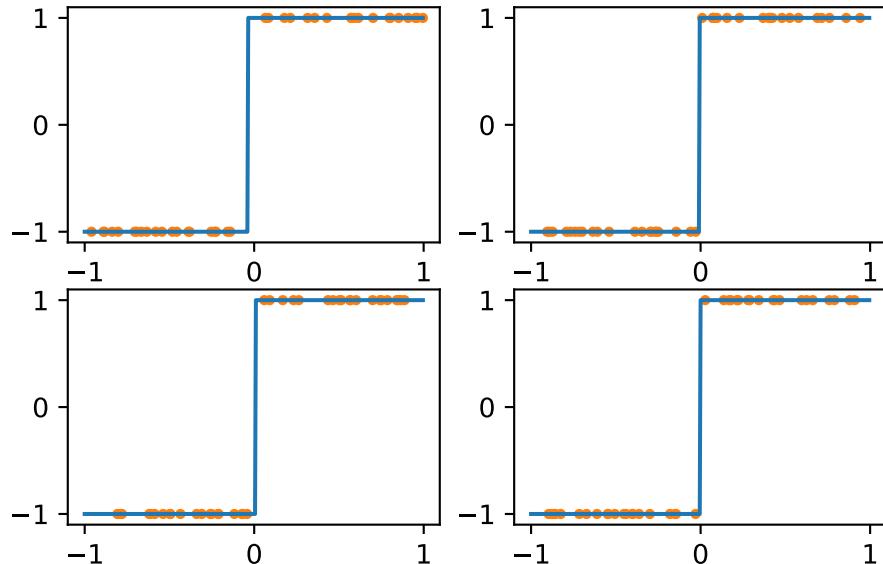


Figure 4.1: kNN with $k=1$ and perfect data

Now suppose we use training sets that have just 3 mislabeled examples each. Here are some resulting classification functions:

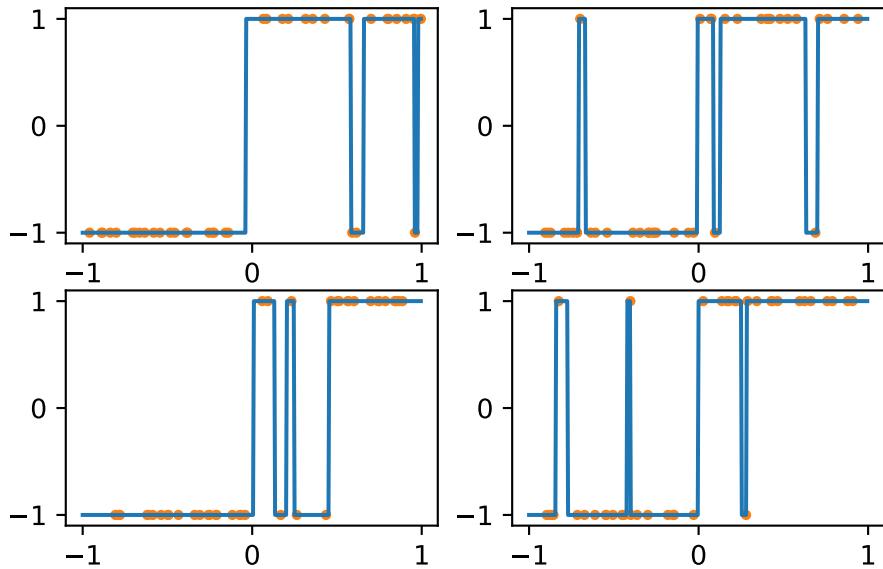


Figure 4.2: kNN with $k=1$ and noisy data

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. This is an extreme example of overfitting.

Now let's bump up to $k = 3$. The results are more like we want, even with noisy data:

The voting mechanism of kNN allows the classifier to ignore isolated outliers. If we continue to $k = 7$, then the 3 outliers will never be able to outvote the correct values:

Note above that the decision boundary is still affected by the noisy values, so some of the more-borderline predictions are wrong, but clearer cases are always handled correctly.

Danger

The lesson here is not simply that “bigger k is better.” In the extreme case of $k = 21$ above, the classifier will predict the same value everywhere! If the true classification boundary were more complicated (i.e., if the classes switched back and forth at high frequency), using even $k = 7$ would be unable to capture many of the details.

4.2.2 Overfitting in decision trees

As mentioned in Example 4.1, the depth of a decision tree correlates with its ability to parse the samples more finely. For $n = 40$ values, a tree of depth 6 is guaranteed to reproduce every sample value perfectly. With noisy data, we see clear signs of overfitting:

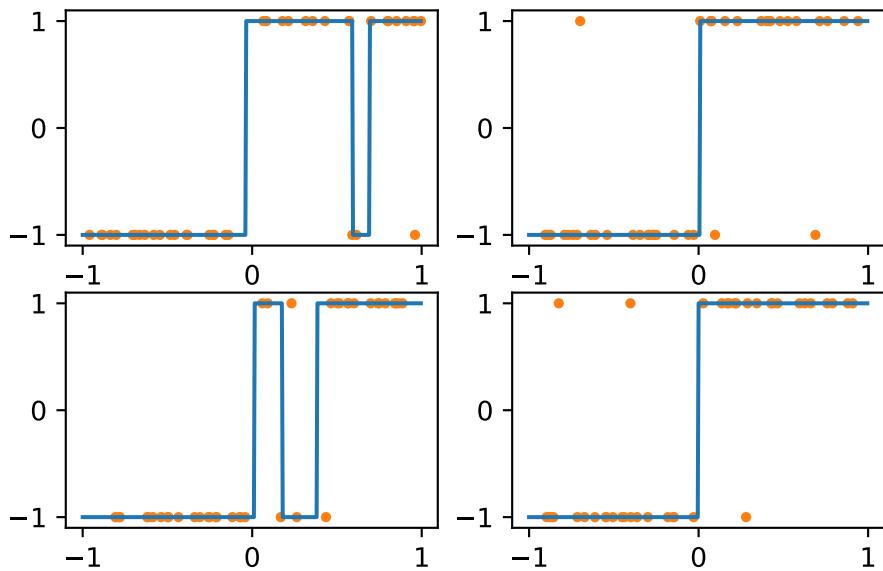


Figure 4.3: kNN with $k=3$ and noisy data

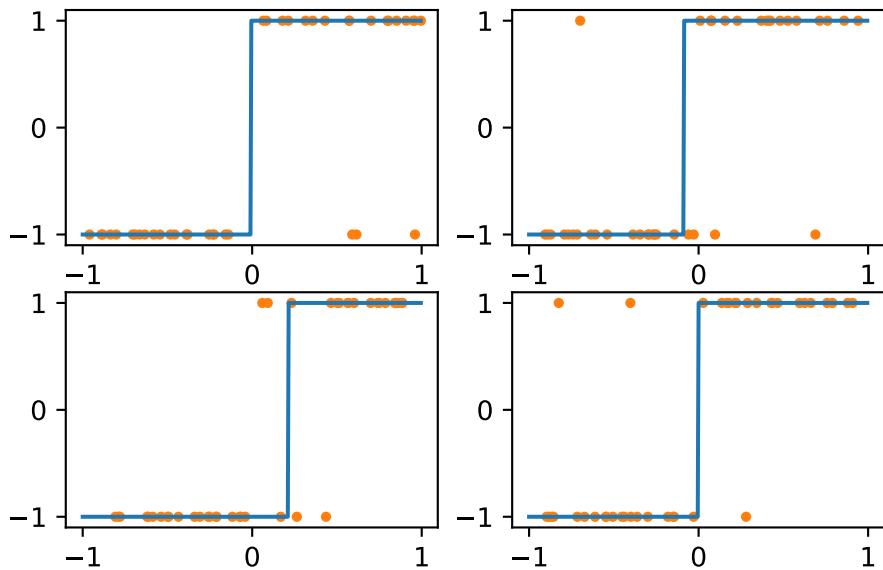


Figure 4.4: kNN with $k=7$ and noisy data

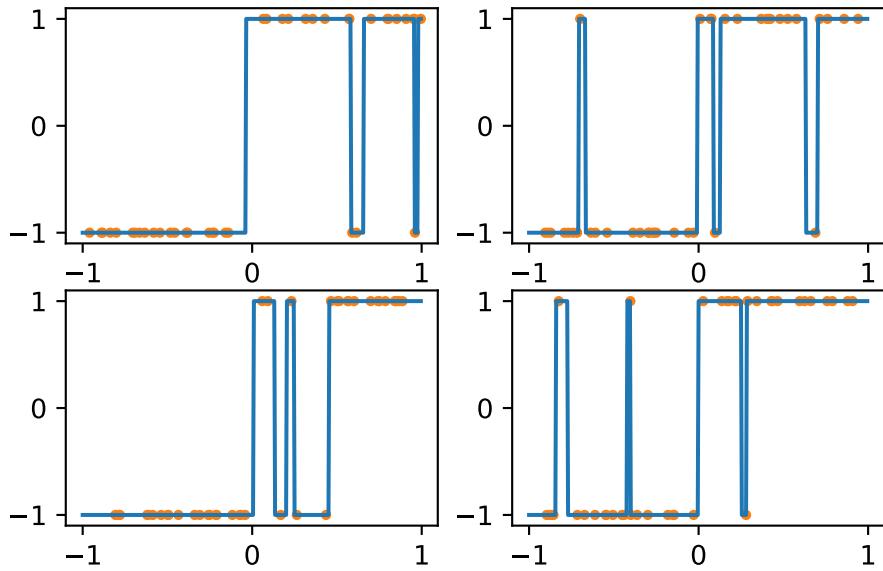


Figure 4.5: Decision tree with depth=6 and noisy data

Using a shallower tree reduces the extent of overfitting:

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 4.2, and Figure 4.5, 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_tr, X_te, y_tr, y_te = train_test_split(
        X, y,
```

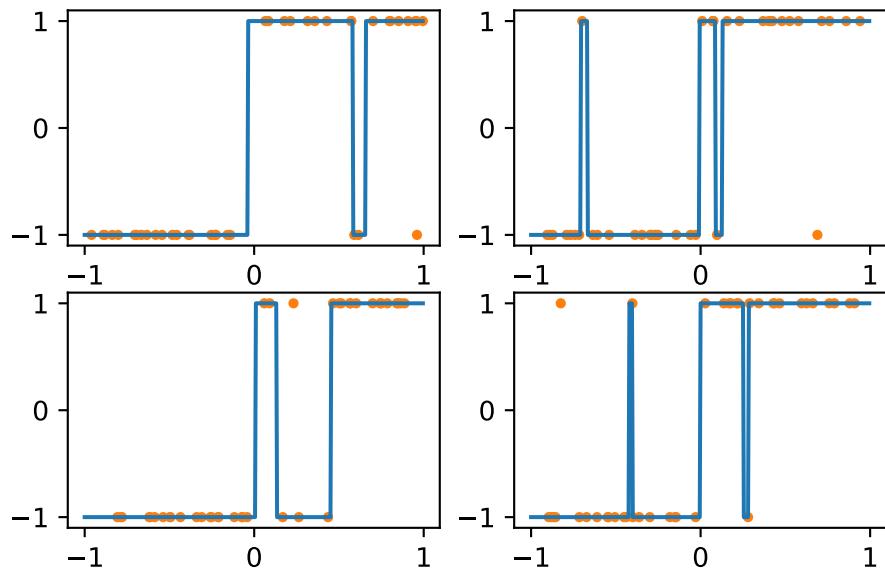


Figure 4.6: Decision tree with depth=3 and noisy data

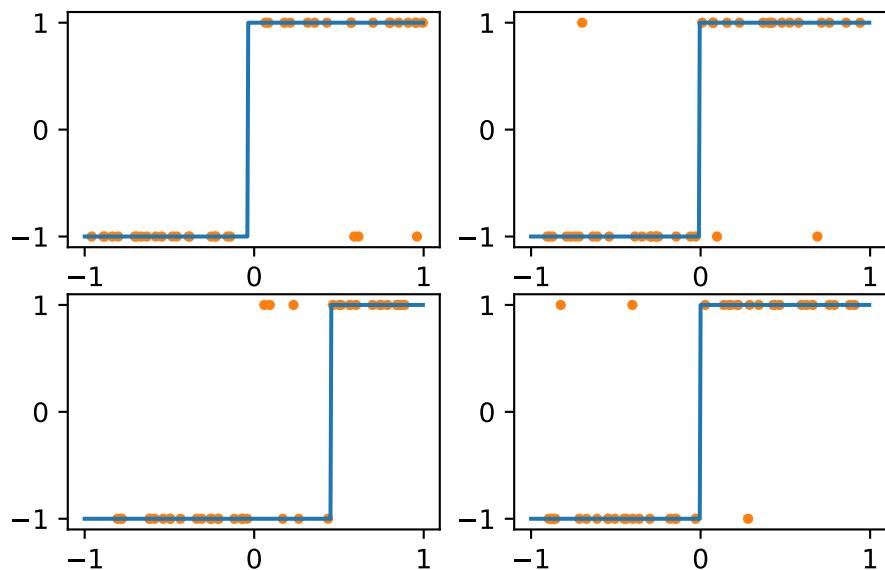


Figure 4.7: Decision tree with depth=2 and noisy data

```

    test_size=0.2,
    shuffle=True,
    random_state=1
)
results = []
for i in range(100):
    X_tr, y_tr = shuffle(X_tr, y_tr, random_state=i)
    XX, yy = X_tr[:n,:], y_tr[: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_te, learner.predict(X_te))
    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)

```

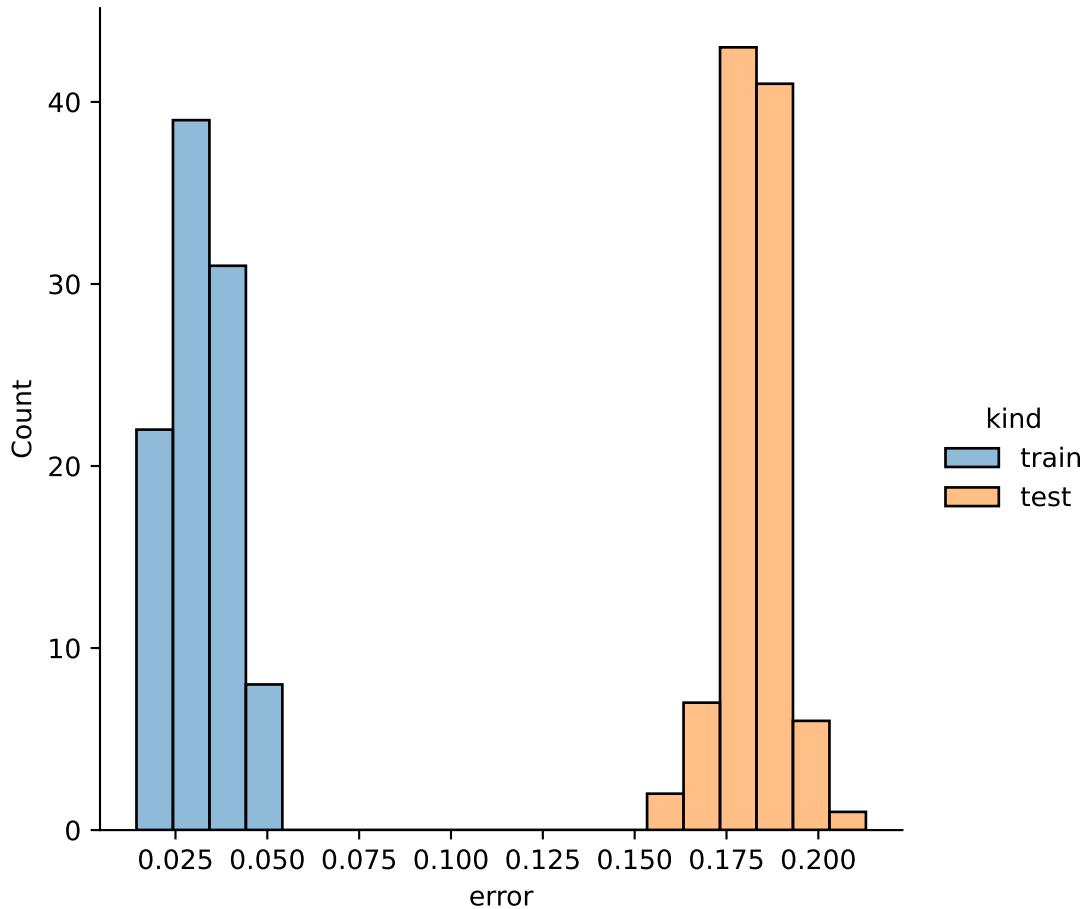
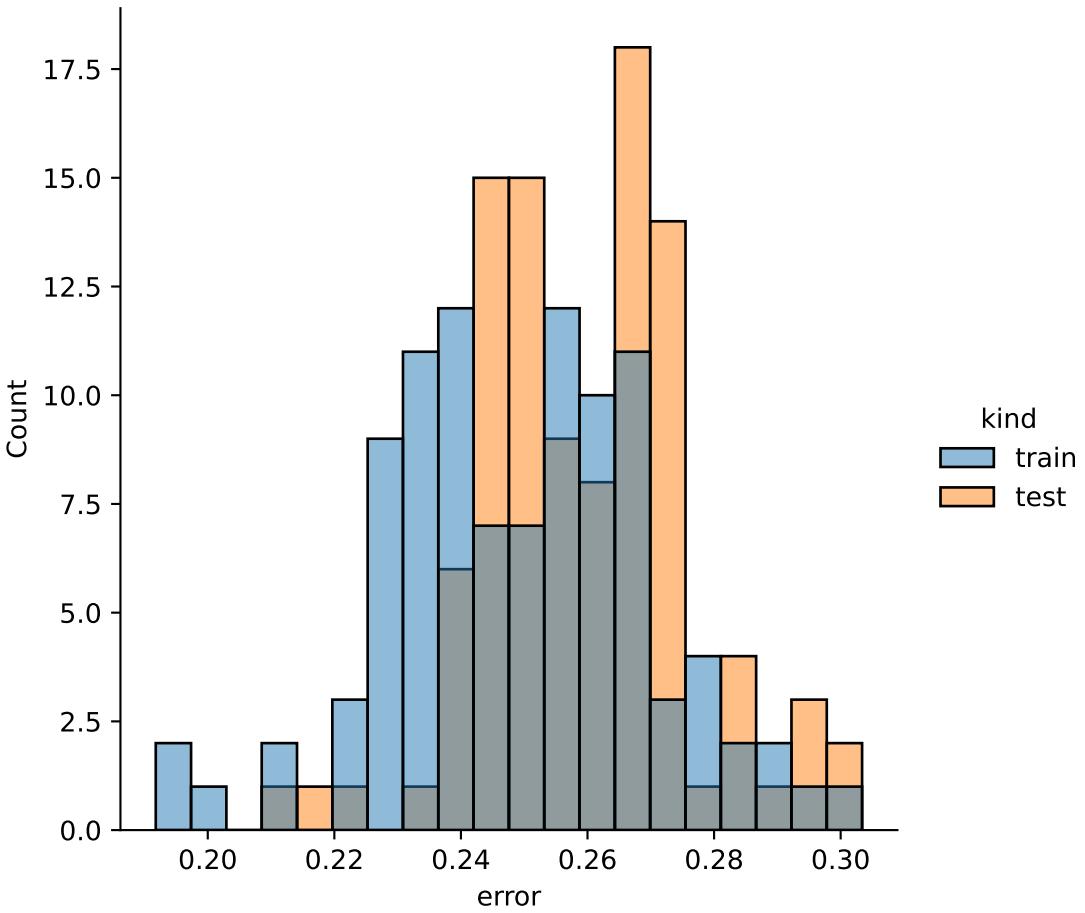


Figure 4.8: ?(caption)

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 actually *underfit* to the data: the behavior of the data is probably too complex to be replicated well by such a shallow tree. We have encountered the bias–variance tradeoff again.

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 much like trying to simulate the computation we used in the theory of expected values. The most common way to construct the training sets

is called *bootstrap aggregation*, or **bagging** for short, in which 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.)

Sklearn 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("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"] == "DERMASON"
```

Here is the confusion matrix we get from training a single kNN classifier on this dataset:

```
X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True,
    random_state=1
)

pipe = make_pipeline(StandardScaler(), KNeighborsClassifier(n_neighbors=3))

pipe.fit(X_tr, y_tr)
yhat = pipe.predict(X_te)

print( confusion_matrix(y_te, yhat, labels=[True,False]) )
```

```
[[ 650   55]
 [ 46 1972]]
```

Here, we create an ensemble with 100 such classifiers, each trained on a different subset of size 40% 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=0
)

ensemble.fit(X_tr, y_tr)
yhat = ensemble.predict(X.iloc[:1,:])
print("prediction on first sample is",
      bool(yhat[0]))
)
```

```
prediction on first sample is False
```

The `estimators_` field of the `ensemble` object is a list of the individual trained classifiers. With a little work, we can manually tally up the number that vote `True` on a query:

```
query = X.to_numpy()[:1,:] # must use an array
yy = [ model.predict(query)[0] for model in ensemble.estimators_ ]
print(f"sum(yy) / len(yy):{0:.0%} vote for True")
```

```
26% vote for True
```

Since only 26% vote `True`, the prediction of the ensemble is `False`, as printed out above. Over the testing set, we find some improvement in the confusion matrix:

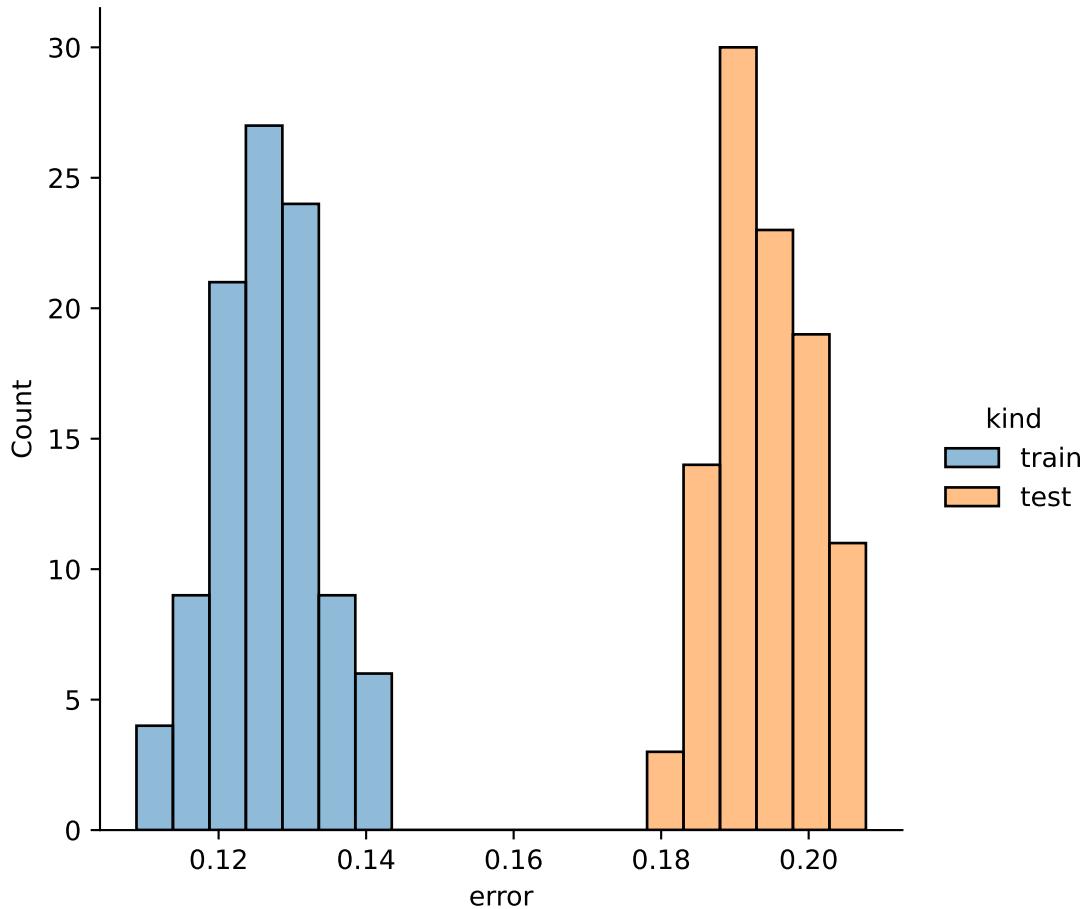
```
yhat = ensemble.predict(X_te)
print( confusion_matrix(y_te, yhat, labels=[True,False]) )
```

```
[[ 654   51]
 [ 41 1977]]
```

Example 4.5. Let's return to the experiment of Example 4.3, where we found that a decision tree of depth $r = 12$ was badly overfit. Now we use an ensemble of 50 such trees:

```
forest = datasets.fetch_covtype()
X = forest["data"][:50000,:8]
y = (forest["target"][:50000] == 1)

tree = DecisionTreeClassifier(max_depth=12)
ensemble = BaggingClassifier(
    tree,
    max_samples=0.25,    # fraction of data to use for each
    n_estimators=50,
    random_state=0,
    n_jobs=-1           # use processes in parallel
)
experiment(ensemble, X, y, 5000)
```



Compared to the earlier experiment (see Figure 4.8), the separation between training and testing was greatly reduced, although seemingly at a small cost to the bias.

Note

An ensemble of decision trees is known as a **random forest**. We could have used a `RandomForestClassifier` to accomplish the bagged decision tree ensemble in Example 4.5.

In addition to training on random subsets of the data, a bagging classifier can use random subsets of features (i.e., dimensions). The purpose again is to increase the diversity of the individual estimators in order to make the ensemble more robust.

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 potential for interpreting the decision process the way we can for an individual tree.

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 test scores over a range of hyperparameter choices and simply choose the best that scores best. That amounts to inspecting the graphs and values in the examples above and choosing the best outcomes. However, if we base hyperparameter optimization on a fixed test 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, which we often want to avoid.

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. The most common version is ***k*-fold cross-validation**:

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 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 4.6. 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 4.7. Let's apply cross-validation to the beans dataset.

```

beans = pd.read_excel("Dry_Bean_Dataset.xlsx")
X = beans.drop("Class", axis=1)
y = beans["Class"]

X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.15,
    shuffle=True, random_state=0
)

```

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=0)
scores = cross_validate(
    learner,
    X_tr, y_tr,
    cv=kf,
    scoring="balanced_accuracy"
)

print("Validation scores:")
print( scores["test_score"] )

```

```
Validation scores:  
[0.93393194 0.93762617 0.923854 0.93617109 0.93037398 0.9326016 ]
```

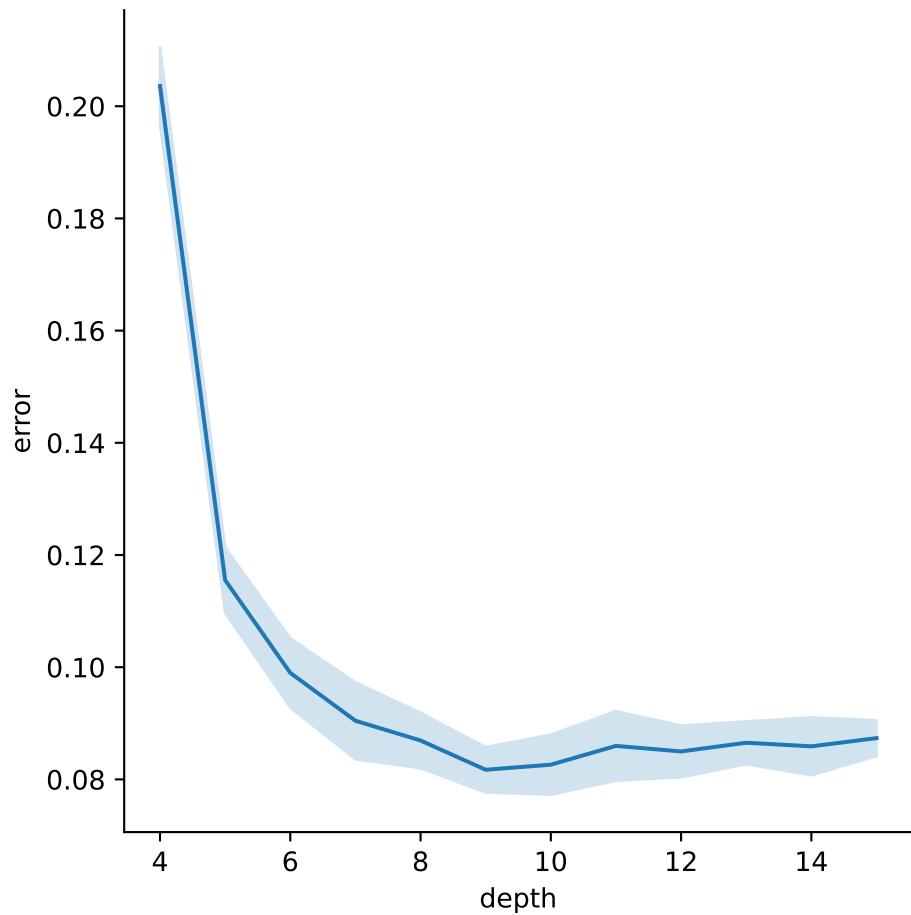
The low variance across the folds that we see above is reassurance that they are representative. Conversely, if the scores were spread more widely, we would be concerned that there was too much dependence on the training set, which might indicate overfitting.

4.4.2 Hyperparameter tuning

If we perform cross-validations as we vary a hyperparameter, we get a **validation curve**.

Example 4.8. 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_tr, y_tr,  
        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	0.203600
5	0.115527
6	0.098991
7	0.090455
8	0.086932
9	0.081718
10	0.082619
11	0.085970
12	0.084984
13	0.086531
14	0.085893
15	0.087356

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_tr, y_tr)
yhat = tree.predict(X_te)
print("score is", balanced_accuracy_score(y_te, yhat))

score is 0.9191394340897692
```

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 4.9. 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_tr, X_te, y_tr, y_te = train_test_split(
```

```

X, y,
test_size=0.15,
shuffle=True, random_state=2
)
X_te.head()

```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness
528	13.940	13.17	90.31	594.2	0.12480	0.09755
291	14.960	19.10	97.03	687.3	0.08992	0.09823
467	9.668	18.10	61.06	286.3	0.08311	0.05428
108	22.270	19.67	152.80	1509.0	0.13260	0.27680
340	14.420	16.54	94.15	641.2	0.09751	0.11390

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 = KFold(n_splits=4, shuffle=True, random_state=0)

grid_dt = GridSearchCV(learner, grid,
                      scoring="f1",
                      cv=kf,
                      n_jobs=-1
                     )
grid_dt.fit(X_tr, y_tr)

print("Best parameters:")
print(grid_dt.best_params_)
print()
print("Best score:")
print(grid_dt.best_score_)

```

```

Best parameters:
{'criterion': 'gini', 'max_depth': 5, 'min_impurity_decrease': 0.0}

```

```

Best score:
0.9582328658156416

```

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_tr, y_tr)  
grid_knn.best_params_, grid_knn.best_score_  
  
({'kneighborsclassifier__metric': 'euclidean',  
 'kneighborsclassifier__n_neighbors': 8,  
 'kneighborsclassifier__weights': 'uniform'},  
 0.9749108434555516)
```

Each fitted grid search object is itself a classifier that was trained on the full training set at the optimal hyperparameters:

```
score = lambda cl, X, y: f1_score( y, cl.predict(X) )  
  
print("best tree f1 score:", score(grid_dt, X_te, y_te))  
print("best knn f1 score:", score(grid_knn, X_te, y_te))  
  
best tree f1 score: 0.9306930693069307  
best knn f1 score: 0.9814814814814815
```

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 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.

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.

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 :

$$\frac{\partial L}{\partial a} = \sum_{i=1}^n 2x_i(ax_i + b - y_i) = 2a \left(\sum_{i=1}^n x_i^2 \right) + 2b \left(\sum_{i=1}^n x_i \right) - 2 \sum_{i=1}^n x_i y_i.$$

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 :

$$\begin{aligned} a \left(\sum_{i=1}^n x_i^2 \right) + b \left(\sum_{i=1}^n x_i \right) &= \sum_{i=1}^n x_i y_i, \\ a \left(\sum_{i=1}^n x_i \right) + bn &= \sum_{i=1}^n y_i. \end{aligned} \tag{5.1}$$

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 adopt 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.
:::{#def-regression-inner-product} Given any d -dimensional real-values vectors \mathbf{u} and \mathbf{v} , their **inner product** is

$$\mathbf{u}^T \mathbf{v} = \sum_{i=1}^d u_i v_i = u_1 v_1 + u_2 v_2 + \dots + u_d v_d. \quad (5.2)$$

:::

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 $\mathbf{u} \cdot \mathbf{v}$.

There is an important link between the dot product and the 2-norm:

$$\mathbf{u}^T \mathbf{u} = \sum_{i=1}^d u_i^2 = \|\mathbf{u}\|_2^2. \quad (5.3)$$

The equations in Equation 5.1 may now be written as

$$\begin{aligned} a(\mathbf{x}^T \mathbf{x}) + b(\mathbf{x}^T \mathbf{e}) &= \mathbf{x}^T \mathbf{y}, \\ a(\mathbf{e}^T \mathbf{x}) + b(\mathbf{e}^T \mathbf{e}) &= \mathbf{e}^T \mathbf{y}. \end{aligned} \quad (5.4)$$

5.1.2 Performance metrics

We need to establish 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.

In this section, we will use x_i for $i = 1, \dots, n$ to mean the training set features, y_i to mean the corresponding training values, and \hat{y}_i to mean the values predicted on the training set by the regressor.

Definition 5.2. The **residuals** of the regression are

$$y_i - \hat{y}_i, \quad i = 1, \dots, n. \quad (5.5)$$

We can express them as the vector $\mathbf{y} - \hat{\mathbf{y}}$.

A quirk of linear regression is that it's an older idea than most of machine learning, and it's often presented as though the training and testing sets are identical. We give the following definitions in terms of \mathbf{y} and $\hat{\mathbf{y}}$ from the training data. They can also be calculated for a set of values and predictions obtained from a separate testing set, though a few of the properties don't translate in that case.



Danger

The terms *error* and *residual* are frequently used interchangeably and inconsistently. We try to follow the most common practices here, even though the names can be confusing if you think about them too hard.

Definition 5.3. The **mean squared error** (MSE) is

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2 = \frac{1}{m} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2.$$

The **mean absolute error** (MAE) is

$$\text{MAE} = \frac{1}{m} \sum_{i=1}^m |y_i - \hat{y}_i| = \frac{1}{m} \|\mathbf{y} - \hat{\mathbf{y}}\|_1.$$

MAE is less sensitive than MSE to large outliers. Both quantities are dimensional and therefore depend on how the variables are scaled, but at least the units of MAE are the same as of the data.

Definition 5.4. 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 mean of y_1, \dots, y_n .

Important things to know about the coefficient of determination:

1. The CoD is dimensionless and therefore independent of scaling.
2. 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 $\hat{\mathbf{y}}$.
3. If $\hat{y}_i = y_i$ for all i (i.e., perfect predictions), then $R^2 = 1$.
4. If $\hat{y}_i = \bar{y}$ for all i , then $R^2 = 0$.

The notation is *highly* unfortunate, though, because for other regression methods, R^2 can actually be negative! Such a result indicates that the predictor is doing worse than just predicting the mean value every time.

Example 5.2. Let's find the coefficient of determination for the fit in Example 5.1, where we found the regressor $\hat{f}(x) = \frac{3}{2}x + \frac{5}{3}$. Now $\bar{y} = \frac{1}{3}(0 + 2 + 3) = \frac{5}{3}$, and

$$\sum_{i=1}^m \sum_{i=1}^n (y_i - \hat{y}_i)^2 = (0 - \frac{1}{6})^2 + (2 - \frac{5}{3})^2 + (3 - \frac{19}{6})^2 = \frac{1}{6},$$

$$\sum_{i=1}^m (y_i - \bar{y})^2 = (0 - \frac{5}{3})^2 + (2 - \frac{5}{3})^2 + (3 - \frac{5}{3})^2 = \frac{14}{3}.$$

This yields $R^2 = 1 - (1/6)(3/14) = 27/28$.

If we instead use the arbitrary regressor $\hat{f}(x) = x$, then

$$\sum_{i=1}^m \sum_{i=1}^n (y_i - \hat{y}_i)^2 = (0 + 1)^2 + (2 - 0)^2 + (3 - 1)^2 = 9,$$

$$\sum_{i=1}^m (y_i - \bar{y})^2 = \frac{14}{3}.$$

This yields $R^2 = 1 - (9)(3/14) = -13/14$. Since the result is negative, we would be better off always predicting $5/3$.

5.1.3 Case study: Arctic ice

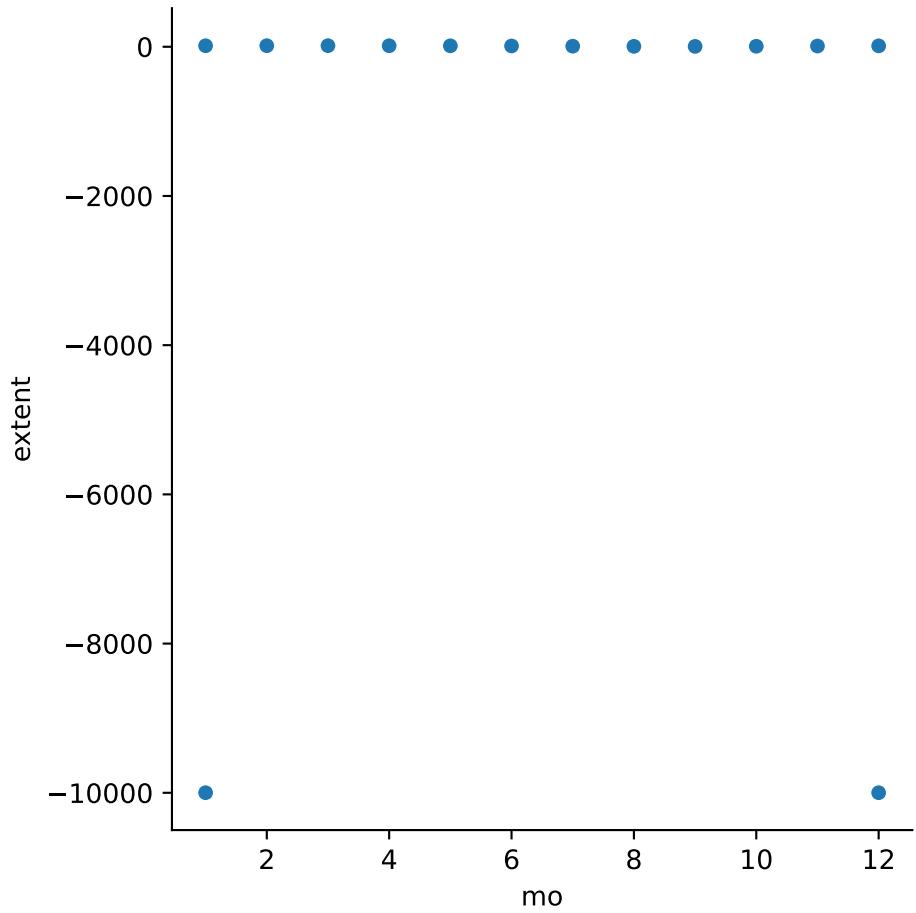
Let's import data about the extent of sea ice in the Arctic circle, collected monthly since 1979.

```
ice = pd.read_csv("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

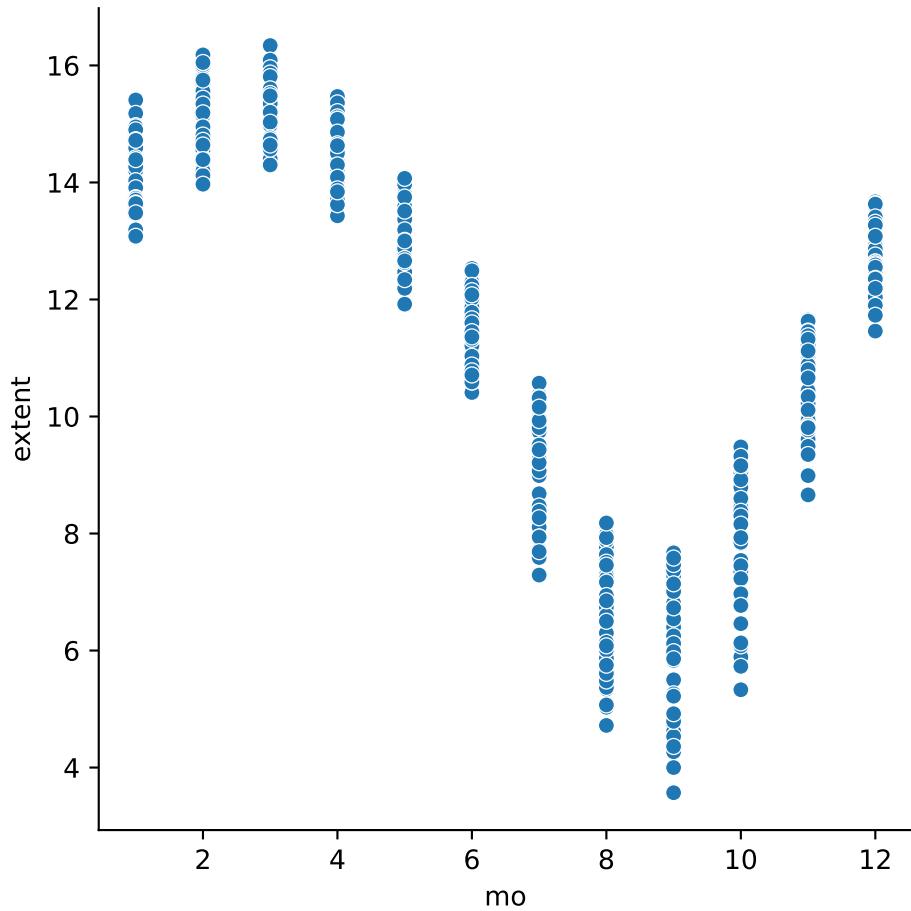
A quick plot reveals something odd-looking.

```
sns.relplot(data=ice, x="mo", y="extent");
```



Everything in the plot is dominated by two large negative values. These probably represent missing or unreliable data, so we remove those rows.

```
ice = ice[ ice["extent"]>0 ].copy()
sns.relplot(data=ice, x="mo", y="extent");
```



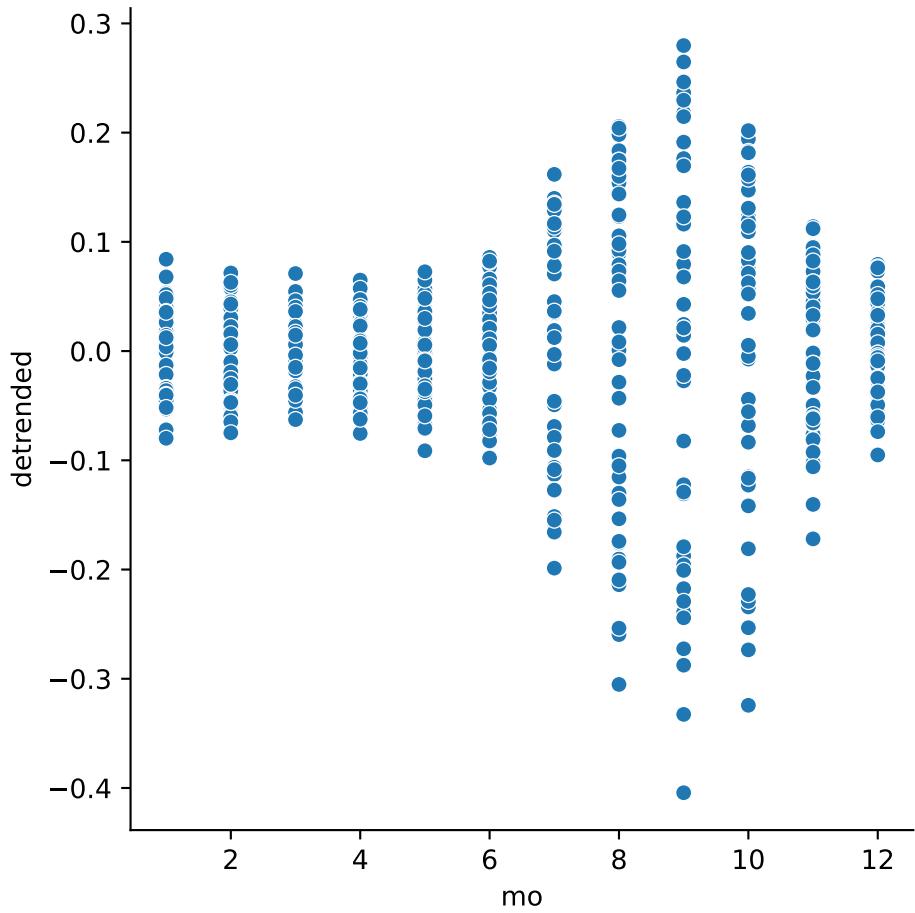
Each dot represents one measurement. As you would expect, the extent of ice rises in the winter and falls in summer.

```
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

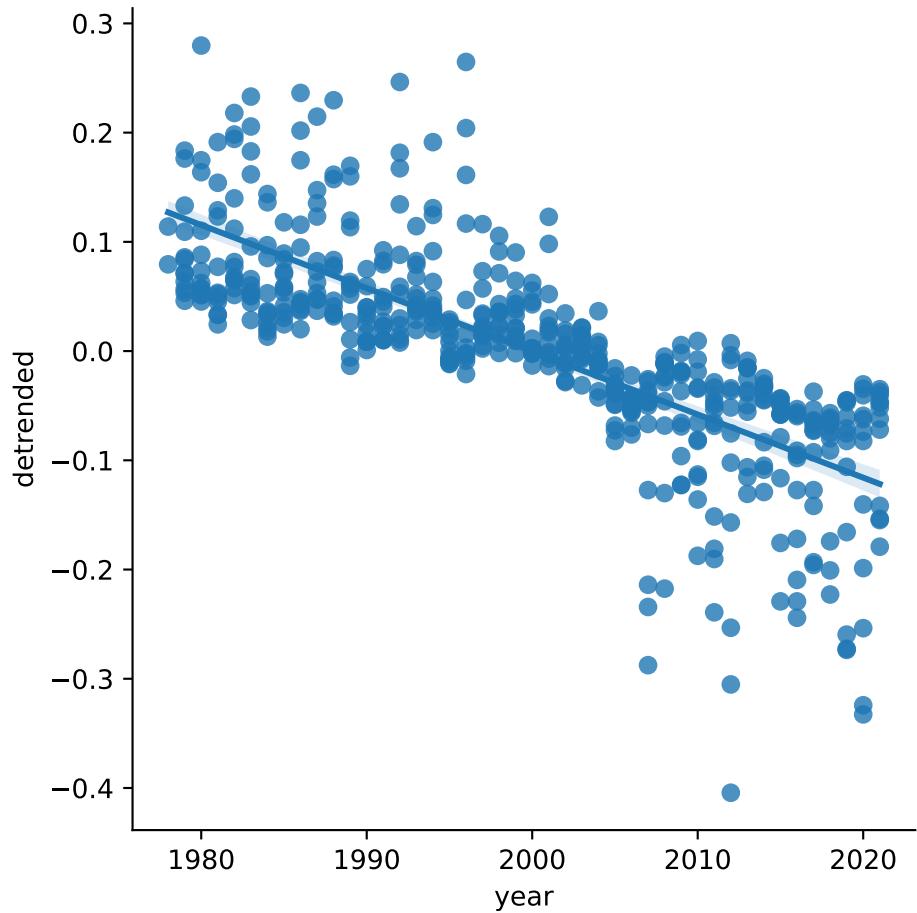
While the effect of the seasonal variation somewhat cancels out over time when fitting a line, it's preferable to remove this obvious trend before the fit takes place. We will add a column that measures the relative change from the mean in each month, i.e., $(x - \bar{x})/\bar{x}$ within each group.

```
recenter = lambda x: x/x.mean() - 1
ice["detrended"] = bymonth["extent"].transform(recenter)
sns.relplot(data=ice, x="mo", y="detrended");
```



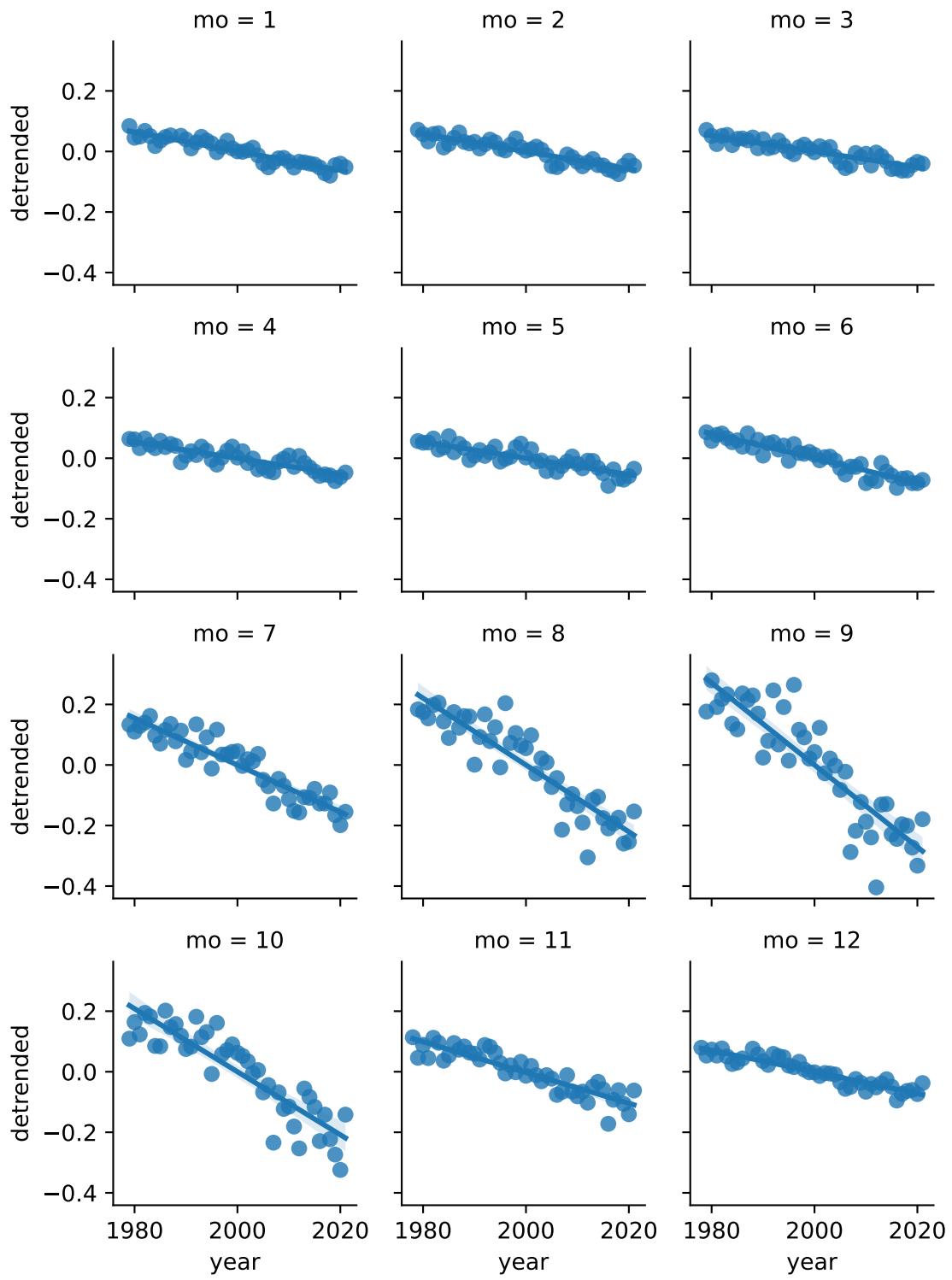
An `lmplot` in seaborn will show the best-fit line.

```
sns.lmplot(data=ice, x="year", y="detrended");
```



However, keep Simpson's paradox in mind. The previous plot showed considerably more variance in the warm months. How do the fits look for the data *within* each month?

```
sns.lmplot(data=ice,
            x="year", y="detrended",
            col="mo", col_wrap=3, height=2
            );
```



While the correlation is negative in each month, the effect size is clearly larger in the summer.

We can get numerical information about a regression line by using a `LinearRegression()` in sklearn. We will focus on the data for August.

```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()

aug = ice["mo"]==8
# We need a frame, not a series, so use a vector for columns for X:
X = ice.loc[aug, ["year"] ]
y = ice.loc[aug, "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.)

```
(lm.coef_, lm.intercept_)
```

```
(array([-0.01103658]), 22.073164930307787)
```

The slope indicates average decrease over time. Here, 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("R-squared:", R2)
```

R-squared: 0.8237357450183893

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.

5.2 Multilinear and polynomial regression

We can extend linear regression to d predictor variables x_1, \dots, x_d :

$$y \approx \hat{f}(\mathbf{x}) = b + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d.$$

We can drop the intercept term b from the discussion, because we could always define an additional constant predictor variable $x_{d+1} = 1$ and get the same effect.

Definition 5.5. 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*.

i Note

Multilinear regression is also simply called linear regression in many contexts.

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 introduce a shorthand notation that is actually the backbone of linear algebra.

Definition 5.6. 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

$$\mathbf{X}\mathbf{w} = \begin{bmatrix} \mathbf{x}_1^T \mathbf{w} \\ \mathbf{x}_2^T \mathbf{w} \\ \vdots \\ \mathbf{x}_n^T \mathbf{w} \end{bmatrix}.$$

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 \mathbf{w} . We do not present them here.

5.2.1 Case study: Advertising and sales

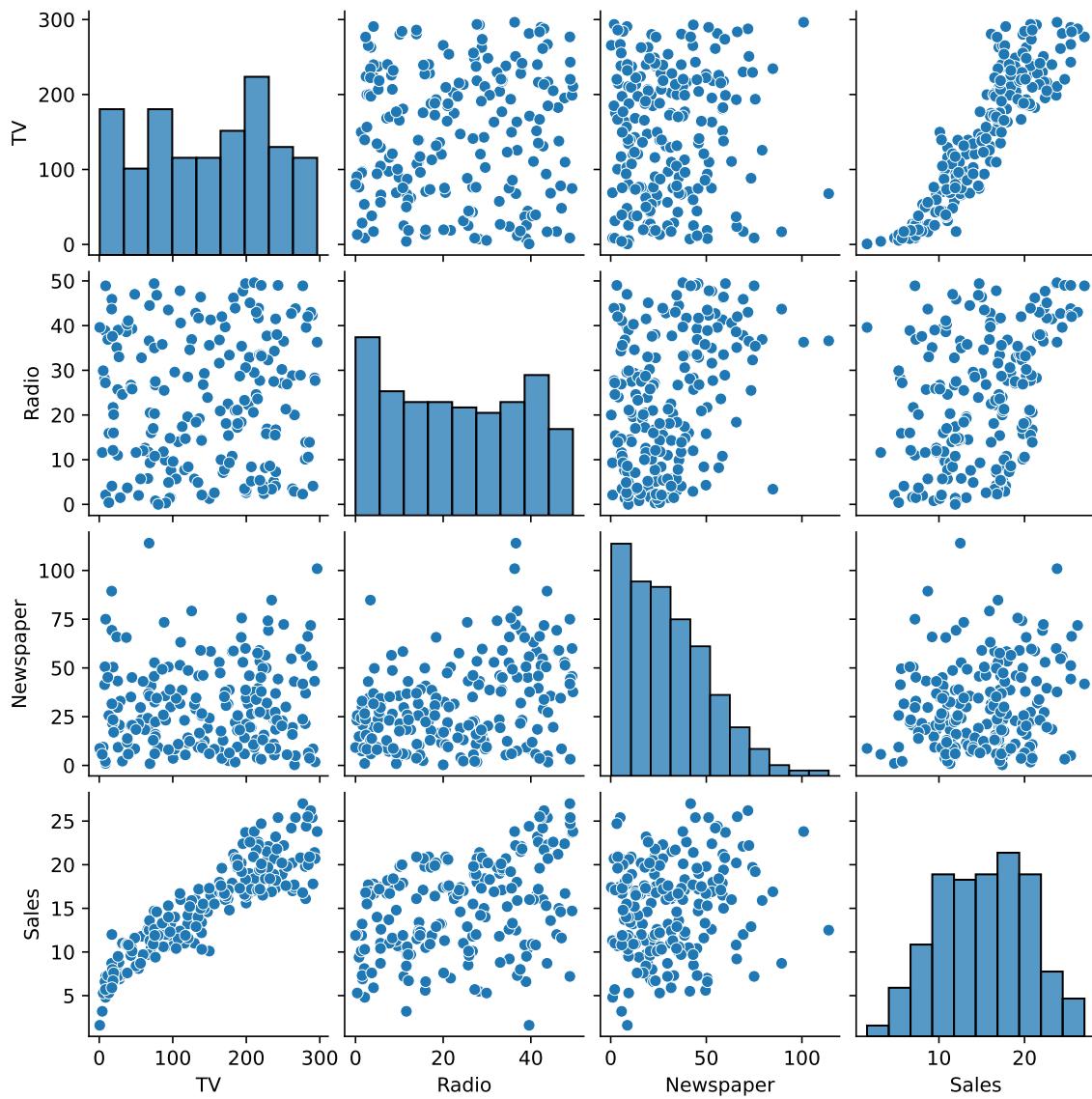
Here we load data about advertising spending on different media in many markets:

```
ads = pd.read_csv("advertising.csv")
ads.head(8)
```

	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
6	57.5	32.8	23.5	11.8
7	120.2	19.6	11.6	13.2

Pairwise scatter plots yield some hints about what to expect from this dataset:

```
sns.pairplot(data=ads, height=2);
```



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"]

from sklearn.linear_model import LinearRegression
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
```

💡 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 change.

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:

```
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
```

💡 Tip

In order to modify a frame, it has to be an independent copy, not just a subset of another frame.

We did see some increase in the R^2 score, and therefore the combination of both types of spending does have a positive effect on *Sales*.

🔥 Danger

We have to be careful interpreting the magnitudes of regression coefficients. These are sensitive to the 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.

Comparisons of coefficients are more meaningful if the features are first standardized.

Interpreting linear regression is a major topic in statistics. There are tests that can lend more precision and rigor to the brief discussion above.

5.2.2 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

$$y \approx w_1 + w_2 t + \dots + w_d t^{d-1},$$

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.

5.2.3 Case study: Fuel efficiency

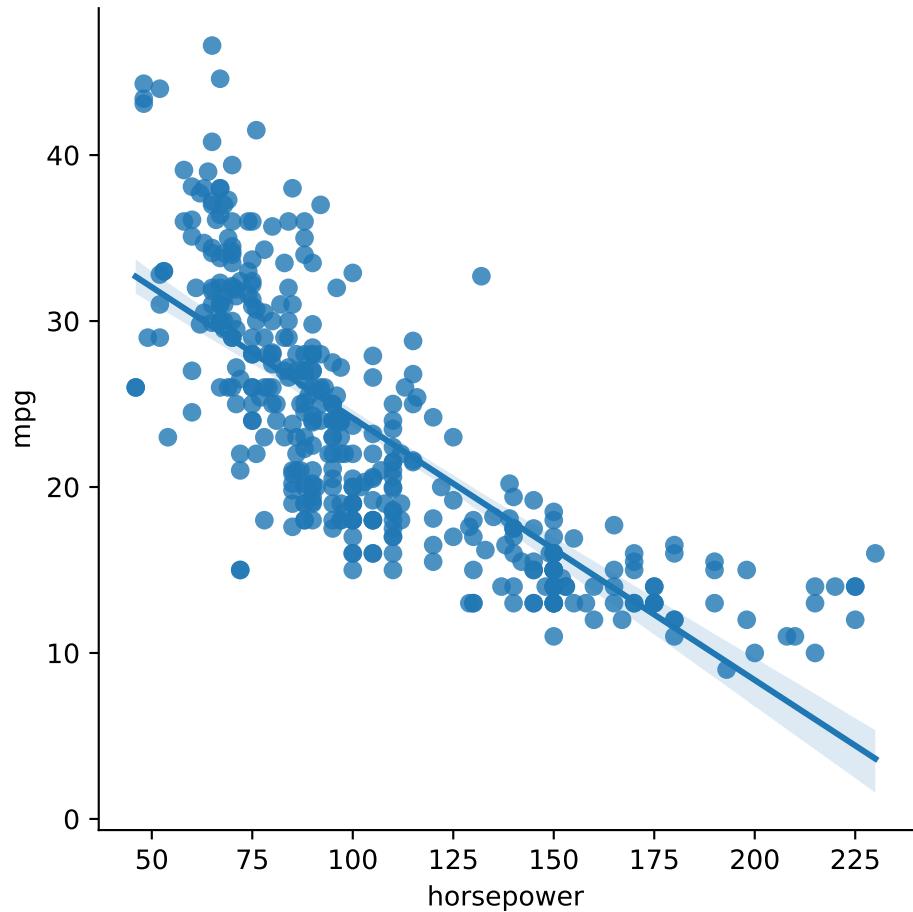
We return to the data set regarding the fuel efficiency of cars:

```
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 skylane
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel s-e
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino

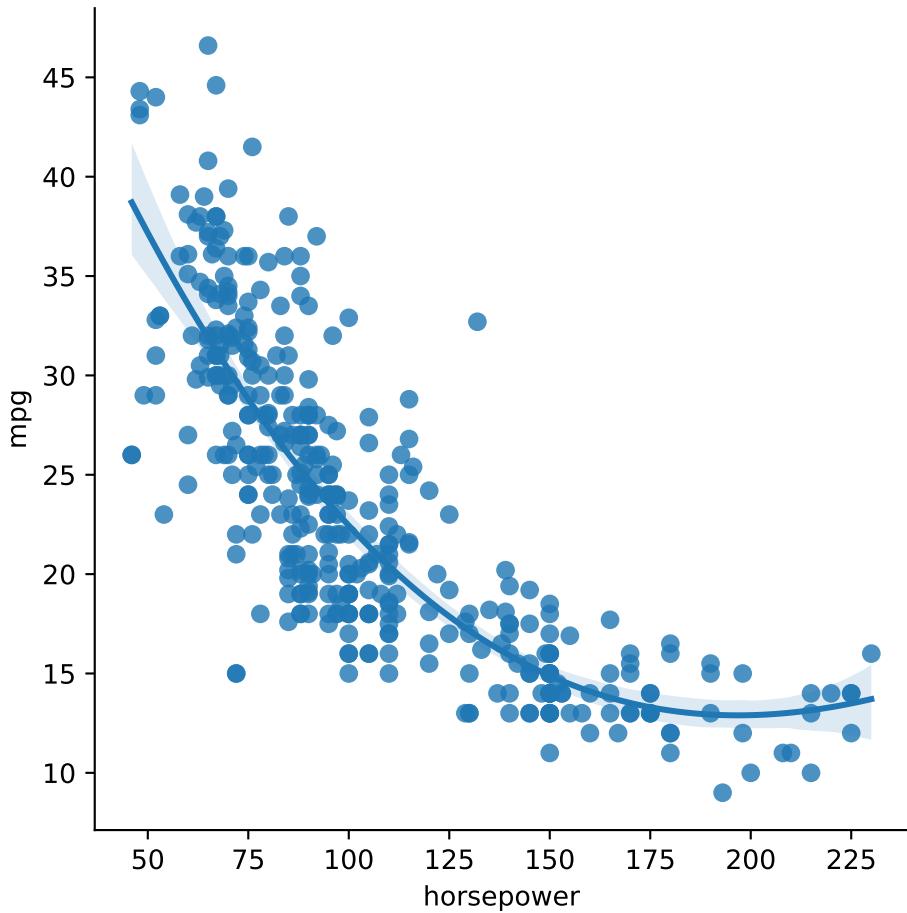
As we would expect, horsepower and miles per gallon are negatively correlated. However, the relationship is not well captured by a straight line:

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



A cubic polynomial produces a much more plausible fit, especially on the right half of the plot:

```
sns.lmplot(data=cars, x="horsepower", y="mpg", order=3);
```



In order to produce 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, we don't need to fit the intercept with the linear regressor.)

```

from sklearn.linear_model import 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)

query = pd.DataFrame([200], columns=X.columns)

```

```
print("prediction at hp=200:", cubic.predict(query))

prediction at hp=200: [12.90220247]
```

The prediction above is consistent with the earlier figure.

We can get the coefficients of the cubic polynomial from the trained regressor:

```
cubic[1].coef_
```

```
array([ 6.06847849e+01, -5.68850128e-01,  2.07901126e-03, -2.14662591e-06])
```

The coefficients go in order of increasing degree.

If a cubic polynomial can fit better than a line, it's plausible 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.

To explore the effect of degree, we split into train and test sets:

```
from sklearn.metrics import mean_squared_error

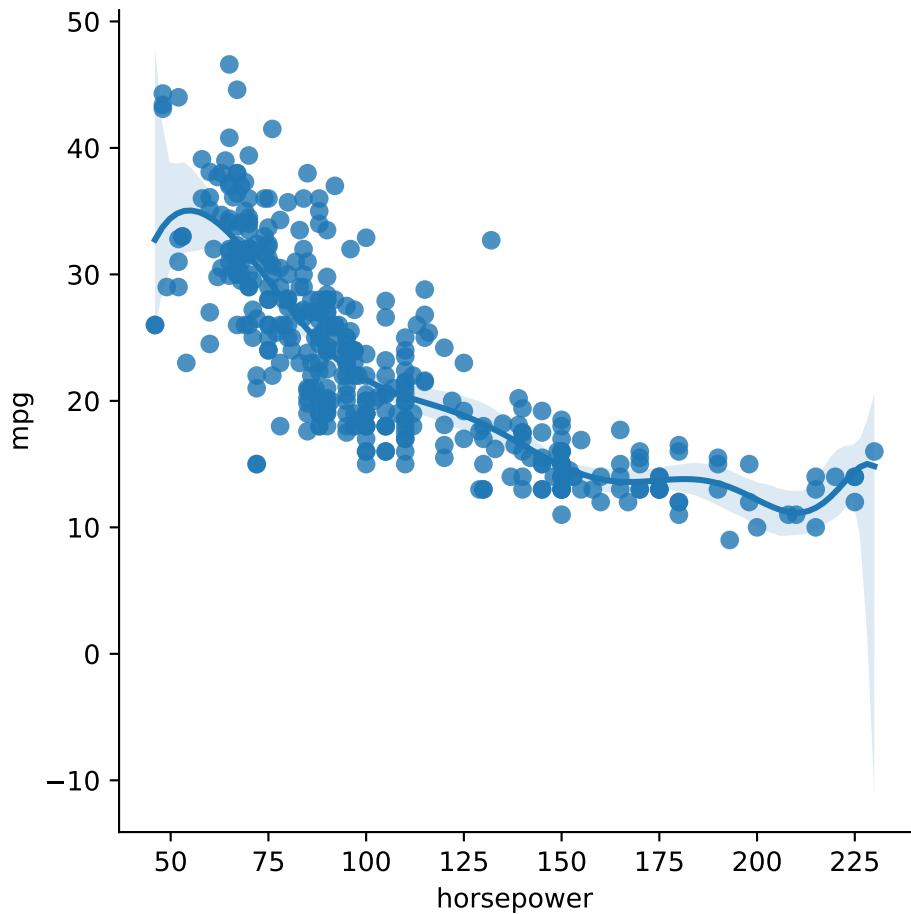
X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2, random_state=0
)

for deg in range(2,11):
    poly = make_pipeline(PolynomialFeatures(degree=deg), lm)
    poly.fit(X_tr, y_tr)
    MSE = mean_squared_error(y_te, poly.predict(X_te))
    print(f'MSE for degree {deg:2}: {MSE:.3f}')
```

```
MSE for degree  2: 16.013
MSE for degree  3: 15.911
MSE for degree  4: 15.819
MSE for degree  5: 15.653
MSE for degree  6: 15.649
MSE for degree  7: 15.593
MSE for degree  8: 18.177
MSE for degree  9: 28.510
MSE for degree 10: 55.261
```

The results above are a clear example of overfitting and the bias–variance tradeoff. A plot of the degree-10 fit shows that the polynomial becomes more oscillatory:

```
sns.lmplot(data=cars, x="horsepower", y="mpg", order=10);
```



In the above plot, note the widening of the confidence intervals near the ends of the domain, indicating increased variance in the predictions.

Next, we keep more of the original data features and pursue a multilinear fit. We chain it with a `StandardScaler` so that all columns have equal mean and scale:

```
def fitcars(model, cars, features):
    X = cars[features]
    X_tr, X_te, y_tr, y_te = train_test_split(
        X, y,
```

```

        test_size=0.2, random_state=0
    )
model.fit(X_tr, y_tr)
MSE = mean_squared_error(y_te, model.predict(X_te))
print(f"MSE: {MSE:.3f}")
return None

features = ["horsepower", "displacement", "cylinders", "weight"]
lm = LinearRegression(fit_intercept=True)
pipe = make_pipeline(StandardScaler(), lm)
fitcars(pipe, cars, features)

```

MSE: 18.624

The fit here is actually a little worse than the low-degree fits based on *horsepower* alone. However, by comparing the coefficients of the individual features, some interesting information emerges:

```
pd.Series(pipe[1].coef_, index=features)
```

	0
horsepower	-1.587584
displacement	0.191193
cylinders	-0.594598
weight	-4.771222

We now have a hypothesis that weight is the most significant negative factor for MPG, and by a wide margin.

Finally, we can combine the use of multiple features and higher degree:

```

pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=2),
    lm
)
fitcars(pipe, cars, features)

```

MSE: 14.794

This is our best regression fit so far, by a mile.

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 common 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

$$y = 1x_1 + 0x_2 + 0x_3 + 0x_4.$$

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. The common choices are **ridge regression**,

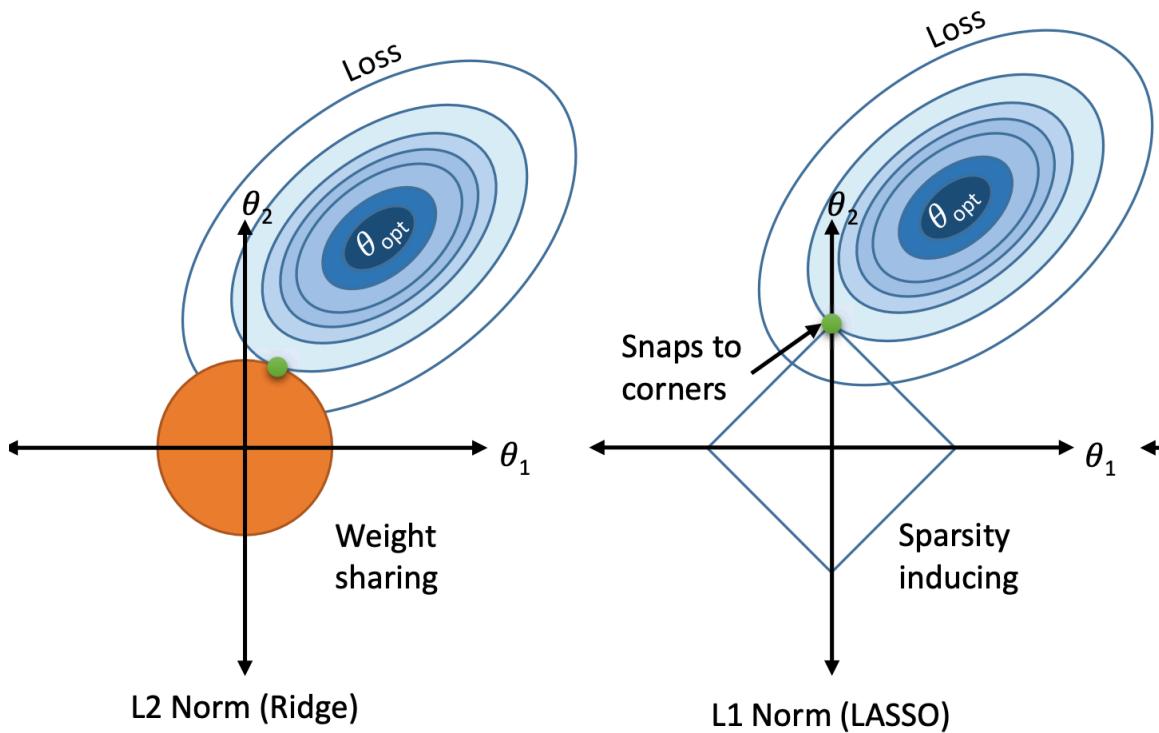
$$L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \alpha \|\mathbf{w}\|_2^2,$$

and **LASSO**,

$$L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \alpha \|\mathbf{w}\|_1.$$

As $\alpha \rightarrow 0$, both forms 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, when regression is run without these variables, there may be little effect on the bias, but a reduction in variance.

5.3.1 Case study: Diabetes

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_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2, random_state=2
)

from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(X_tr, y_tr)
print(f"linear model CoD score: {lm.score(X_te, y_te):.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_tr, y_tr)
print(f"ridge CoD score: {rr.score(X_te, y_te):.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_tr, y_tr)
    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_te, y_te):.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
```

LASSO penalizes the 1-norm of the coefficient vector. Here's a LASSO regression fit:

```
from sklearn.linear_model import Lasso
lass = Lasso(alpha=0.1)
```

```
lass.fit(X_tr, y_tr)
R2 = lass.score(X_te, y_te)
print(f"LASSO model CoD score: {R2:.4f}")
```

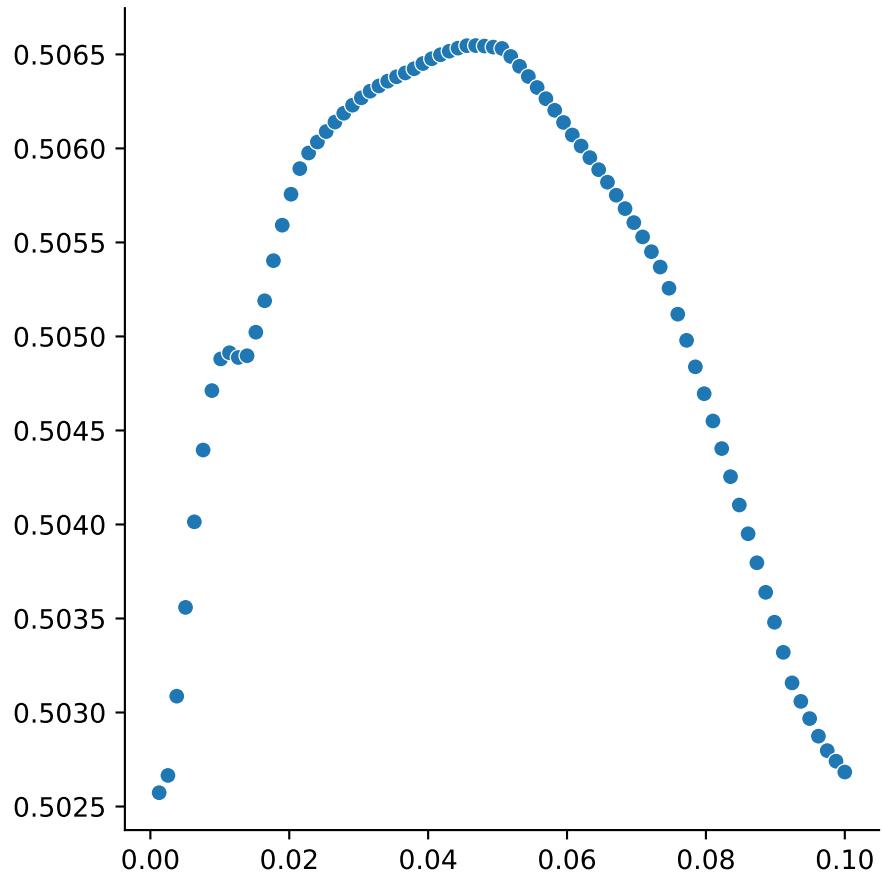
```
LASSO model CoD score: 0.4335
```

A validation curve suggests modest gains in the R^2 score as the regularization parameter is varied:

```
kf = KFold(n_splits=4, shuffle=True, random_state=0)
alpha = np.linspace(0,0.1,80)[1:] # exclude alpha=0

_,scores = validation_curve(
    lass,
    X_tr, y_tr,
    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_tr, y_tr)
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 rank the relative importance of the features by ordering them in terms of decreasing coefficient magnitude:

```
# Get the permutation that sorts values in increasing order.
order = np.argsort( np.abs(lass.coef_) )
order = order[::-1]      # reverse the order
pd.Series( order, index=X.columns )
```

	0
age	8
sex	2
bmi	3
bp	6
s1	1
s2	4
s3	9
s4	7
s5	5
s6	0

The last three features were dropped by LASSO:

```
zeroed = X.columns[order[-3:]]
print(zeroed)
```

```
Index(['s4', 's2', 'age'], dtype='object')
```

Now we can drop these features from the dataset:

```
X_tr_reduced = X_tr.drop(zeroed, axis=1)
X_te_reduced = X_te.drop(zeroed, 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_te,y_te):.4f}")
lm.fit(X_tr_reduced, y_tr)
R2 = lm.score(X_te_reduced, y_te)
print(f"reduced linear model score: {R2:.4f}")
```

```
original linear model score: 0.4399
reduced linear model score: 0.4388
```

5.4 Nonlinear regression

Multilinear regression limits the representation of the dataset to a function of the form

$$\hat{f}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}.$$

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. $\hat{f}(c\mathbf{u}) = c\hat{f}(\mathbf{u})$.

These properties are the essence of what makes a function easy to manipulate, solve for, and analyze. For our particular \hat{f} , they 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 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 relatively 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 5.3. 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

$$\hat{f}(x) = \begin{cases} 0, & x < 1, \\ \frac{1}{2}, & x = 1, \\ 1, & x > 1. \end{cases}$$

(Convince yourself that the result is the same whether the mean or the median is used.) Thus, for instance, $\hat{f}(1.2) = 1$, while $2\hat{f}(0.6) = 0$, which is not equal to $\hat{f}(2 \cdot 0.6)$.

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 5.4. 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_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2, random_state=0
)
```

```

lm = LinearRegression()
lm.fit(X_tr, y_tr)
print(f"linear model CoD: {lm.score(X_te, y_te):.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=1)
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_tr, y_tr)

print(f"best kNN CoD: {optim.score(X_te, y_te):.4f}")

```

best kNN CoD: 0.7439

As you can see above, we got some improvement over the linear regressor.

5.4.2 Decision tree

Recall that a decision tree recursively divides the examples into subsets. As with kNN, we can replace taking a classification vote over a leaf subset with taking a mean or median of the values. But the method of determining splits needs to be changed as well.

Instead of using a measure of subset impurities to determine the best split, the split is chosen to cause the greatest reduction in dispersion within the two subsets. The most common choices for the dispersion measure H are:

1. If using the mean of subset values, then let H be standard deviation.

2. If using the median of subset values, then let H be mean absolute deviation (MAD), defined as

$$\text{MAD} = \frac{1}{m} \sum_{i=1}^m |t_i - t_{\text{med}}|$$

for any list of values t_1, \dots, t_m and t_{med} equal to the median value.

As with classification, a proposal to split into subsets S and T is assigned the weighted score

$$Q = |S|H(S) + |T|H(T).$$

The split location is chosen to minimize Q .

Example 5.5. 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/MAD.

The original value set has median $\frac{1}{2}$ and gets a weighted dispersion of $\frac{5}{2}(3+3+1+1) = 20$. There are three ways to split the data, depending on where the partition falls in relation to the x_i :

- $S = \{2\}$, $T = \{-1, 1, 0\}$

$$\begin{aligned} Q &= 1 \left[\frac{1}{1} |2 - 2| \right] + 3 \left[\frac{1}{3} (|-1 - 0| + |1 - 0| + |0 - 0|) \right] \\ &= 0 + 2 = 2. \end{aligned}$$

- $S = \{2, -1\}$, $T = \{1, 0\}$

$$\begin{aligned} Q &= 2 \left[\frac{1}{2} (|2 - \frac{1}{2}| + |-1 - \frac{1}{2}|) \right] + 2 \left[\frac{1}{2} (|1 - \frac{1}{2}| + |0 - \frac{1}{2}|) \right] \\ &= 3 + 1 = 4. \end{aligned}$$

- $S = \{2, -1, 1\}$, $T = \{0\}$

$$\begin{aligned} Q &= 3 \left[\frac{1}{3} (|2 - 1| + |-1 - 1| + |1 - 1|) \right] + 1 \left[\frac{1}{1} |0 - 0| \right] \\ &= 3 + 0 = 3. \end{aligned}$$

Thus, the first split above produces the smallest total dispersion.

To predict a value for a query x , we follow the tree until ending at a leaf, where we use the mean (if dispersion is STD) or median (if dispersion is MAD) of the examples in the leaf.

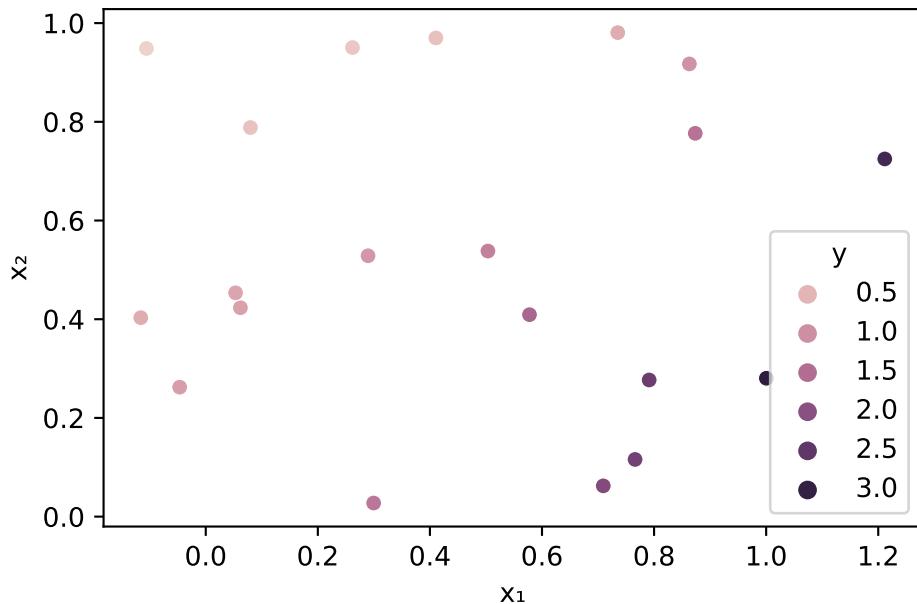
Example 5.6. Here is some simple 2D data:

```

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");

```



The default in `sklearn` is to use STD as the dispersion measure (called `squared_error` in `sklearn`). 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)

```

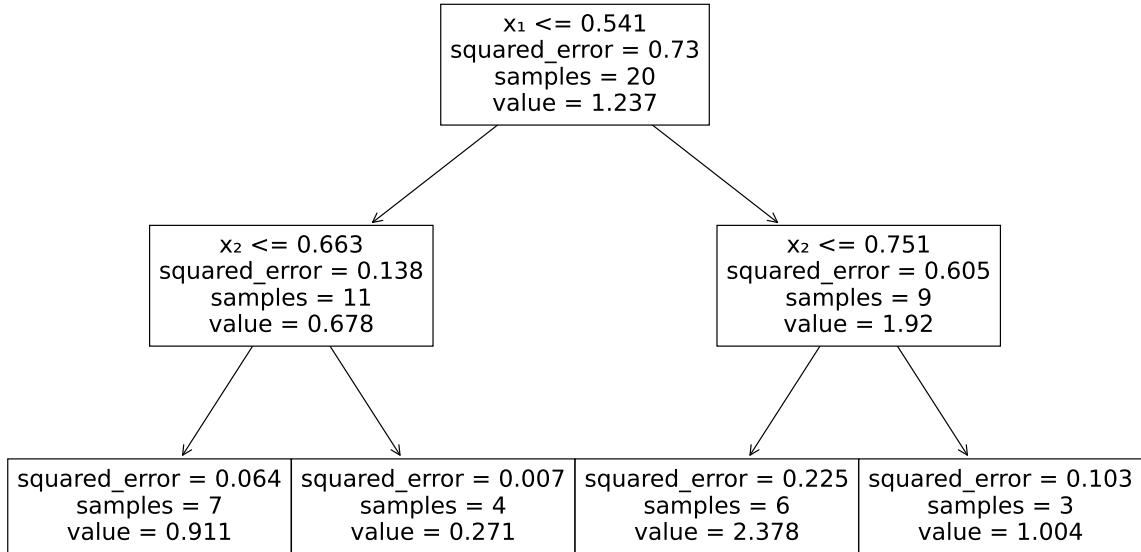
```
DecisionTreeRegressor(max_depth=2)
```

```

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 5.7. Continuing with the data from Example 5.4, we find that we can do even better with a random forest of decision tree regressors:

```
X = cars[features]
y = cars["mpg"]

X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2, random_state=0
)

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_tr, y_tr)

print(f"best forest CoD: {optim.score(X_te, y_te):.4f}")

```

best forest CoD: 0.7779

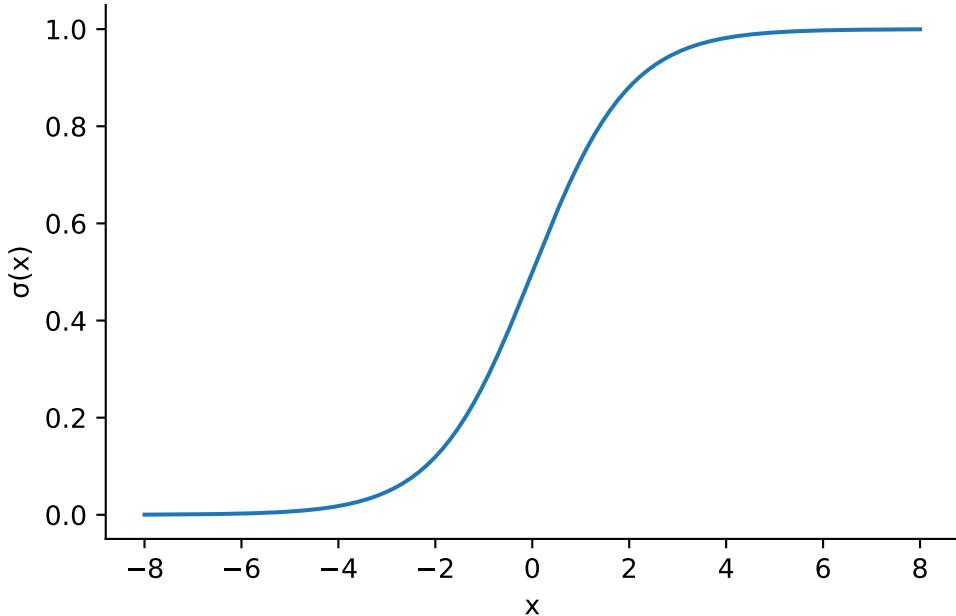
5.5 Logistic regression

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 using the **logistic function**, defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

The logistic function has the real line as its domain and takes the form of a smoothed step increasing from 0 to 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

$$\sum_{i=1}^n (\hat{p}_i - y_i)^2.$$

For binary classification, however, it's more common to use the **cross-entropy** loss function

$$L(\mathbf{w}) = - \sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]. \quad (5.7)$$

(The logarithms in Equation 5.7 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 $\hat{p}_i \rightarrow 0^+$. In words, there is a steep penalty for being almost completely wrong about an observation.

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.

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

$$\tilde{L}(\mathbf{w}) = C L(\mathbf{w}) + \|\mathbf{w}\|,$$

where C is a positive hyperparameter and the vector norm is either the 2-norm (ridge) or 1-norm (LASSO).

! Important

The parameter C functions like the inverse of the regularization parameter α we used in the linear regressor. It's just a different convention chosen historically. As C decreases, the regularization strength increases.

5.5.3 Case study: Personal spam filter

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.

```
spam = pd.read_csv("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	

We create a feature matrix and label vector, and split into train/test sets:

```
X = spam.drop("class", axis="columns")
y = spam["class"]

X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=1
)
```

When using norm-based regularization, it's good practice to standardize the variables, so we will use scaling pipelines. First we use a large value of C to emphasize the regressive loss over the regularization penalty:

```

from sklearn.linear_model import LogisticRegression

logr = LogisticRegression(C=100, solver="liblinear")
pipe = make_pipeline(StandardScaler(), logr)
pipe.fit(X_tr, y_tr)
print("accuracy:", pipe.score(X_te, y_te))

```

accuracy: 0.9337676438653637

💡 Tip

The use of the `solver` keyword is optional, but the solver used above seems to be far faster and more reliable for small datasets than the default.

Let's look at the most extreme regression coefficients, associating them with the feature names and then sorting the results:

```

coef = pd.Series(logr.coef_[0], index=X.columns).sort_values()
print("most hammy features:")
print(coef[:4])
print()
print("most spammy features:")
print(coef[-4:])

```

```

most hammy features:
word_freq_george      -24.055810
word_freq_cs           -8.573920
word_freq_hp            -3.512677
word_freq_meeting       -1.940989
dtype: float64

most spammy features:
char_freq_%23          1.155189
char_freq_%24          1.262551
capital_run_length_longest 1.974990
word_freq_3d             2.112905
dtype: float64

```

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 closer to 1.

The ultimate predictions by the regressor are all either 0 or 1. But we can also see the forecasted probabilities before thresholding:

```
print("predicted classes:")
print( pipe.predict(X_tr.iloc[:5,:]) )
print("\nprobabilities:")
print( pipe.predict_proba(X_tr.iloc[:5,:]) )

predicted classes:
[0 0 0 0 0]

probabilities:
[[0.53768185 0.46231815]
 [0.99694715 0.00305285]
 [0.63975941 0.36024059]
 [0.99634195 0.00365805]
 [0.93740669 0.06259331]]
```

The probabilities might be useful for making decisions based on the results. For example, the first instance above was much less certain about the classification than the second. A more skeptical threshold greater than 0.54 would change the class to 1. As in Section 3.5, the probability matrix can be used to create an ROC curve showing the tradeoffs over all thresholds.

For a validation-based selection of the best regularization parameter value, we can use `LogisticRegressionCV`, which is a convenience method for a grid search. You can specify which values of C to search over, or just say how many, as we do here:

```
from sklearn.linear_model import LogisticRegressionCV

logr = LogisticRegressionCV(
    Cs=40,      # 40 automatically chosen values of C
    cv=5,
    solver="liblinear",
    n_jobs=-1, random_state=0
)
pipe = make_pipeline(StandardScaler(), logr)
pipe.fit(X_tr, y_tr)

print(f"best C value: {logr.C_[0]:.3g}")
```

```

print(f"accuracy score: {pipe.score(X_te,y_te):.5f}")

best C value: 21.5
accuracy score: 0.93485

```

5.5.4 Multiclass case

When there are more than two unique labels possible, 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, giving K different coefficient vectors, \mathbf{w}_k . Now for a query point \mathbf{x} , we can predict probabilities for it being in each class:

$$\hat{q}_k(\mathbf{x}) = \sigma(\mathbf{x}^T \mathbf{w}_k), \quad k = 1, \dots, K.$$

Since the K OVR regressors are done independently, there is no reason to think these probabilities will sum to 1 over all the classes. So we must normalize them:

$$\hat{p}_k(bfx) = \frac{\hat{q}_k(\mathbf{x})}{\sum_{k=1}^K \hat{q}_k(\mathbf{x})}.$$

Computed over a testing set, we get a matrix of probabilities. Each of the rows gives the class probabilities at a single query point, and each of the K columns gives the probability of one class at all the points.

Example 5.8. As a multiclass example, we use a data set about gas sensors recording values over long periods of time:

```

gas = pd.read_csv("gas_drift.csv")
y = gas["Class"]
X = gas.drop("Class", axis="columns")
X_tr, X_te, y_tr, y_te = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True, random_state=1
)

logr = LogisticRegression(solver="liblinear")
pipe = make_pipeline(StandardScaler(), logr)

```

```

pipe.fit(X_tr, y_tr)
print("accuracy score:", pipe.score(X_te, y_te))

```

```
accuracy score: 0.98705966930266
```

We can now look at probabilistic predictions for each class:

```

p_hat = pipe.predict_proba(X_te)
cols = ["Class "+str(i) for i in range(1,7)]
pd.DataFrame(p_hat, columns=cols).head(10)

```

	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6
0	0.000154	5.324116e-06	0.020485	4.207072e-03	0.003196	9.719529e-01
1	0.000004	9.999198e-01	0.000074	6.812414e-07	0.000002	5.869686e-17
2	0.008624	3.847762e-03	0.000017	1.282243e-03	0.001295	9.849346e-01
3	0.237558	5.503335e-08	0.000020	7.222371e-01	0.000054	4.013053e-02
4	0.016611	3.643991e-02	0.010126	2.032083e-01	0.032053	7.015607e-01
5	0.002096	9.976914e-01	0.000163	4.242972e-05	0.000007	7.264576e-11
6	0.000212	2.805245e-05	0.003276	5.669049e-04	0.017867	9.780503e-01
7	0.080161	7.056994e-08	0.000025	8.963731e-01	0.000091	2.334934e-02
8	0.002059	9.640239e-06	0.000399	7.548506e-02	0.002079	9.199678e-01
9	0.979774	1.129101e-03	0.000001	1.136161e-02	0.000003	7.730162e-03

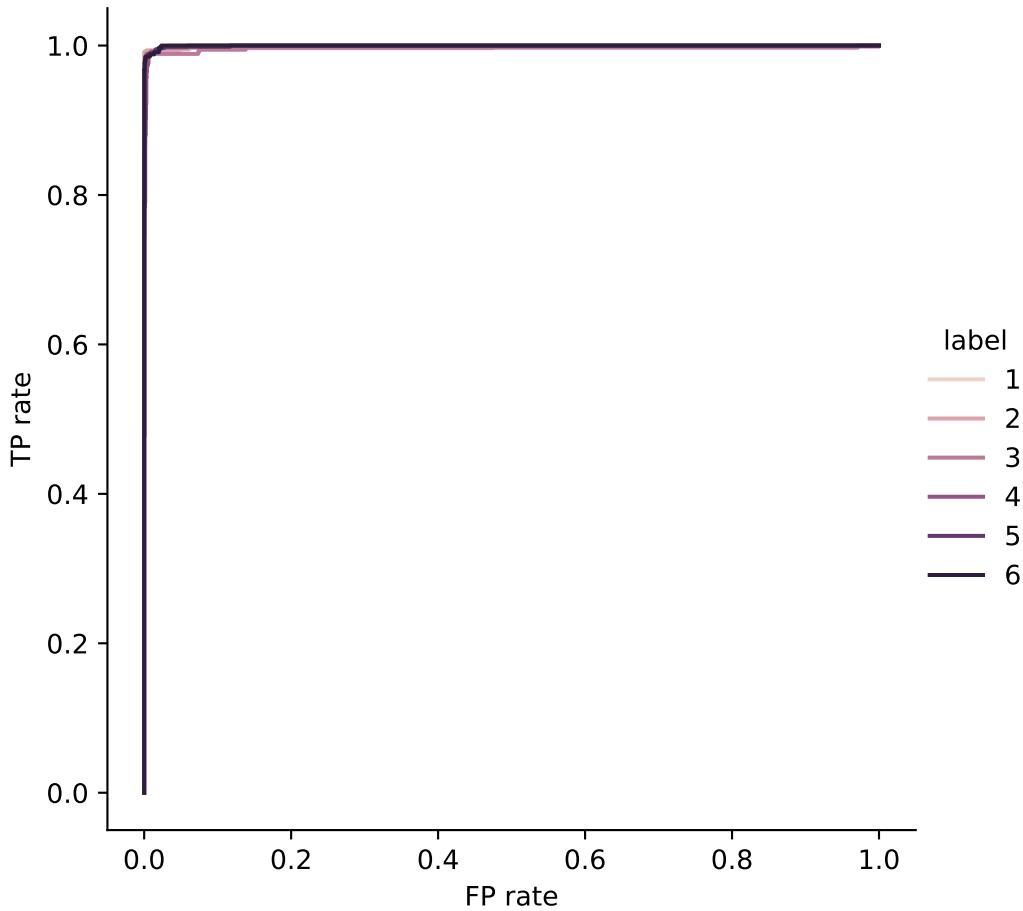
This allows us to see that the ROC curves are nearly perfect:

```

results = []
for i, label in enumerate(pipe.classes_):
    actual = (y_te==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
            );

```



Based on the ROC curves, we could choose a high decision threshold to cut down on false positives without losing many true positives.

Exercises

Exercise 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, $\hat{f}(x) = \alpha x$. Use calculus to find a formula for the value of α that minimizes the sum of squared residuals,

$$r = \sum_{i=1}^n (f(x_i) - y_i)^2.$$

Exercise 5.2. Suppose that $x_1 = -2$, $x_2 = 1$, and $x_3 = 2$. Define α as in Exercise 5.1, and define the predicted values $\hat{y}_k = \alpha x_k$ for $k = 1, 2, 3$. Express each \hat{y}_k as a combination of the three values y_1 , y_2 , and y_3 , which remain arbitrary. (This is a special case of a general fact about linear regression: each prediction is a linear combination of the training values.)

Exercise 5.3. 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 $f(\bar{x}) = \bar{y}$. This can be done without first solving for a and b , which is a bit tedious to write out.)

Exercise 5.4. Suppose that values y_i for $i = 1, \dots, n$ are to be fit to features (u_i, v_i) using a multilinear function $f(u, v) = \alpha u + \beta v$. Define the sum of squared residuals

$$r = \sum_{i=1}^n (f(u_i, v_i) - y_i)^2.$$

Show that by holding α constant and taking a derivative with respect to β , and then holding β constant and taking a derivative with respect to α , at the minimum residual we must have

$$\begin{aligned} \left(\sum u_i^2 \right) \alpha + \left(\sum u_i v_i \right) \beta &= \sum u_i y_i, \\ \left(\sum u_i v_i \right) \alpha + \left(\sum v_i^2 \right) \beta &= \sum v_i y_i. \end{aligned}$$

:::{#exr-regression-regular-no-intercept} Repeat Exercise 5.1, but using the regularized residual

$$\tilde{r} = C\alpha^2 + \sum_{i=1}^n (f(x_i) - y_i)^2.$$

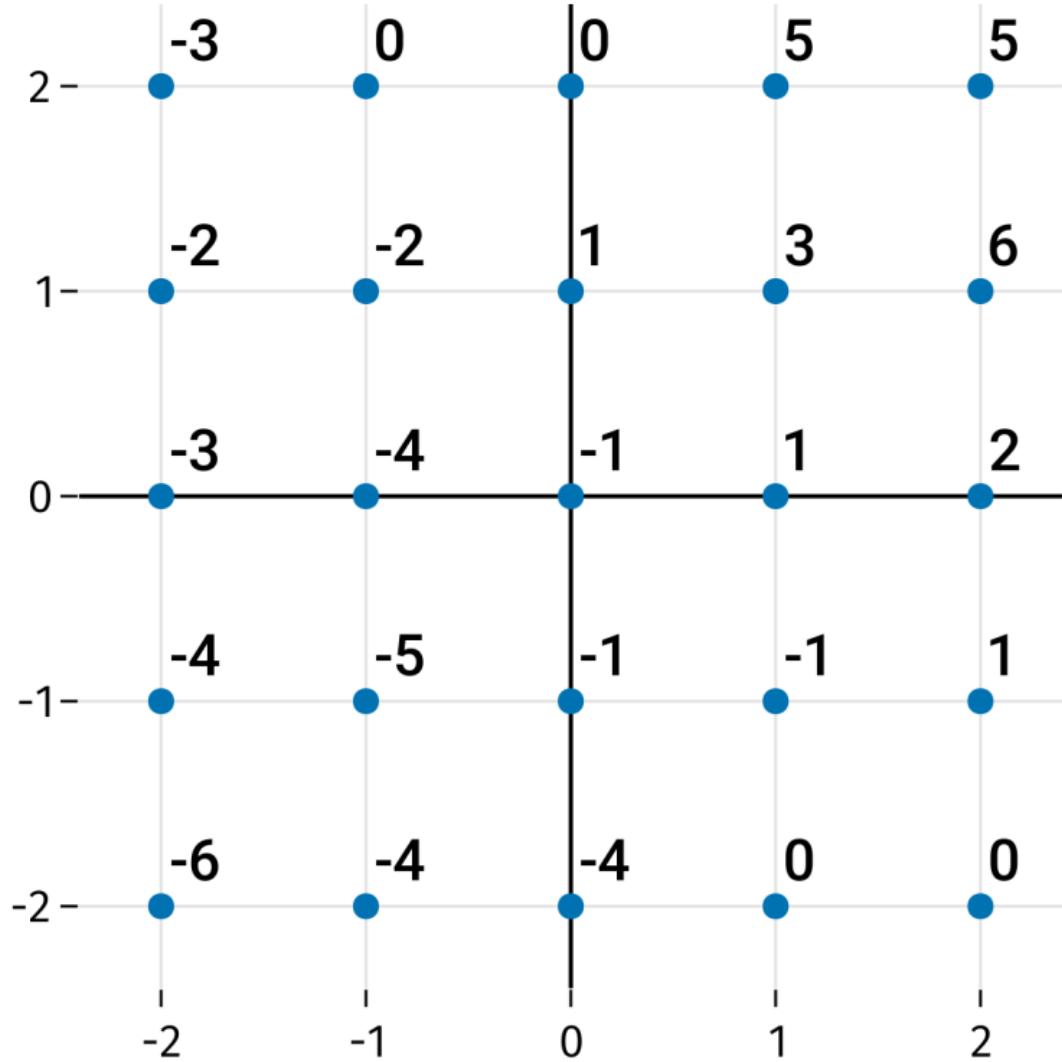
Exercise 5.5. Repeat Exercise 5.4, but using the regularized residual

$$\tilde{r} = C(\alpha^2 + \beta^2) + \sum_{i=1}^n (f(u_i, v_i) - y_i)^2.$$

Exercise 5.6. 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) $x \leq 3.5$

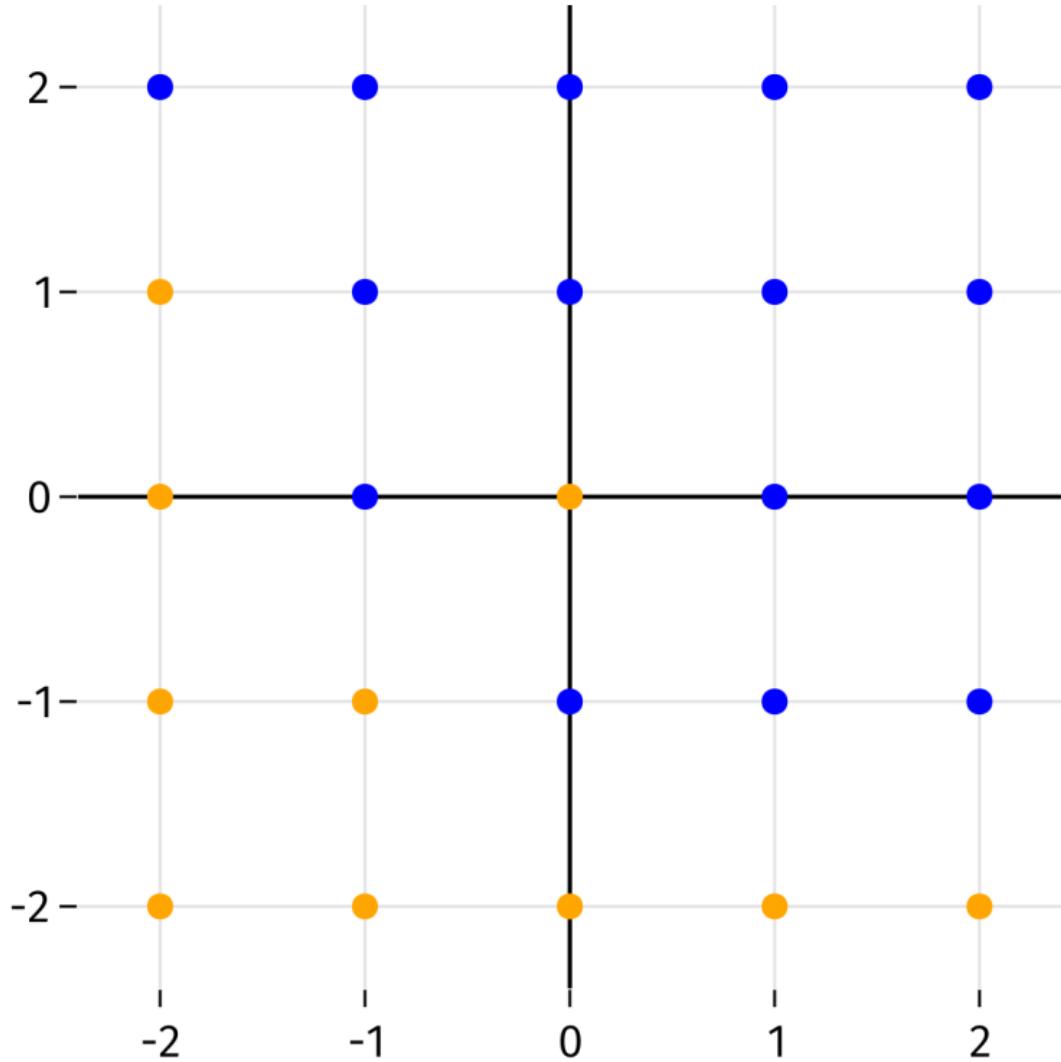
Exercise 5.7. Here are values on an integer lattice.



Let $\hat{f}(x_1, x_2)$ be a kNN regressor with $k = 4$, Euclidean metric, and mean averaging. Carefully sketch a one-dimensional plot of \hat{f} along the given line.

- (a) $\hat{f}(1.2, t)$ for $2 \leq t \leq 2$
- (b) $\hat{f}(t, -0.75)$ for $2 \leq t \leq 2$
- (c) $\hat{f}(t, 1.6)$ for $2 \leq t \leq 2$
- (d) $\hat{f}(-0.25, t)$ for $2 \leq t \leq 2$

Exercise 5.8. Here are blue/orange labels on an integer lattice.



Let $\hat{f}(x_1, x_2)$ be a kNN probabilistic classifier with $k = 4$, Euclidean metric, and mean averaging. Carefully sketch a one-dimensional plot of the probability of the blue class along the given line.

- (a) $\hat{f}(1.2, t)$ for $2 \leq t \leq 2$
- (b) $\hat{f}(t, -0.75)$ for $2 \leq t \leq 2$
- (c) $\hat{f}(t, 1.6)$ for $2 \leq t \leq 2$
- (d) $\hat{f}(-0.25, t)$ for $2 \leq t \leq 2$

Exercise 5.9. Here are some label values and probabilistic predicted categories for them.

$$\begin{aligned} y &: [0, 0, 1, 1] \\ \hat{p} &: [\frac{1}{4}, 0, \frac{1}{2}, 1] \end{aligned}$$

Using base-2 logarithms, calculate the cross-entropy loss for these predictions.

Exercise 5.10. Let $\mathbf{x} = [-1, 0, 1]$ and $\mathbf{y} = [0, 1, 0]$. This is to be fit to a probabilistic predictor $\hat{p}(x) = \sigma(ax)$ for parameter a .

(a) Show that the cross-entropy loss function $L(a)$, using natural logarithms, satisfies

$$L'(a) = \frac{e^a - 1}{e^a + 1}.$$

(b) Explain why part (a) implies that $a = 0$ is the global minimizer of the loss L .

(c) Using the result of part (b), simplify the optimum predictor function \hat{p} .

Exercise 5.11. Let $\mathbf{x} = [-1, 1]$ and $\mathbf{y} = [0, 1]$. This is to be fit to a probabilistic predictor $\hat{p}(x) = \sigma(ax)$ for parameter a . Without regularization, the best fit takes $a \rightarrow \infty$, 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.,

$$L(a) = \ln[1 - \hat{p}(-1)] - \ln[\hat{p}(1)] + C|a|,$$

for a positive regularization constant C .

- (a) Show that L' is never zero for $a < 0$.
- (b) Show that if $0 < C < 1$, then L' has a zero at

$$a = \ln\left(\frac{2}{C} - 1\right).$$

Assume that this value minimizes L .

- (c) Show that the minimizer above is a decreasing function of C . (Therefore, increasing C 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. 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
    )

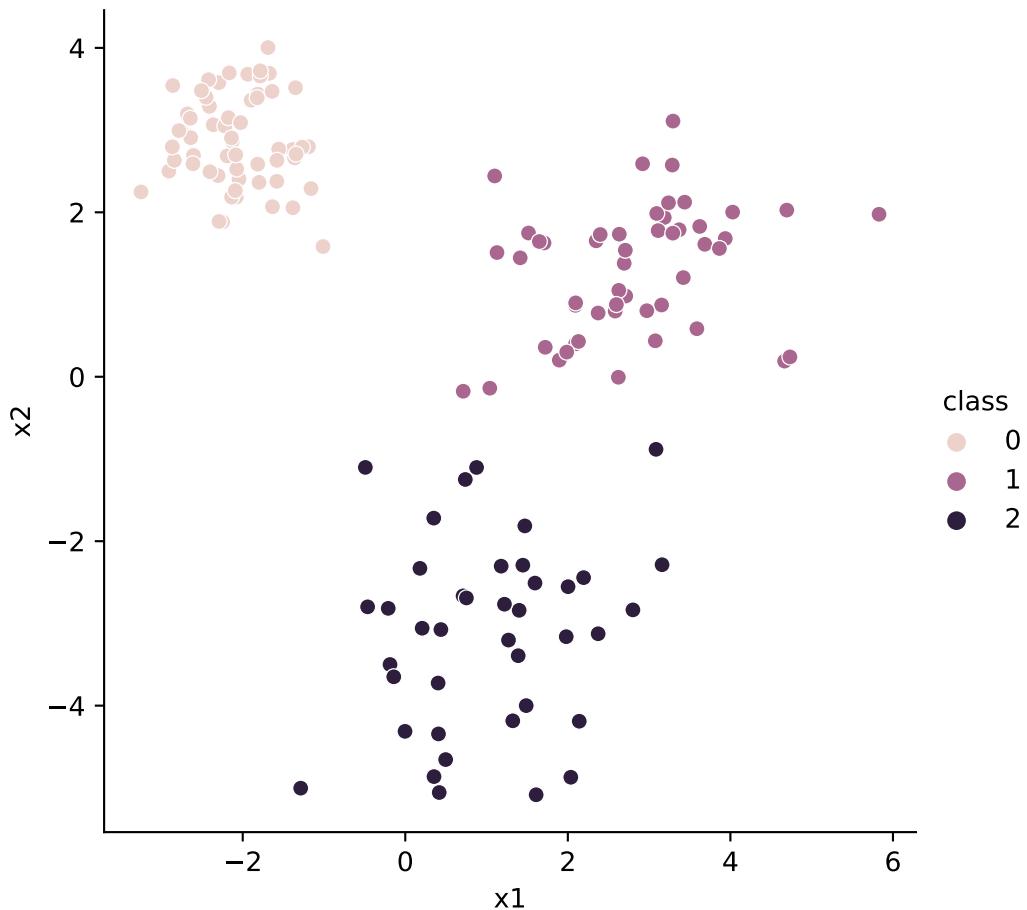
    return pd.DataFrame({"x1":X[:,0], "x2":X[:,1], "class":y})

```

```

blobs = blobs_data()
sns.relplot(data=blobs, x="x1", y="x2", hue="class");

```

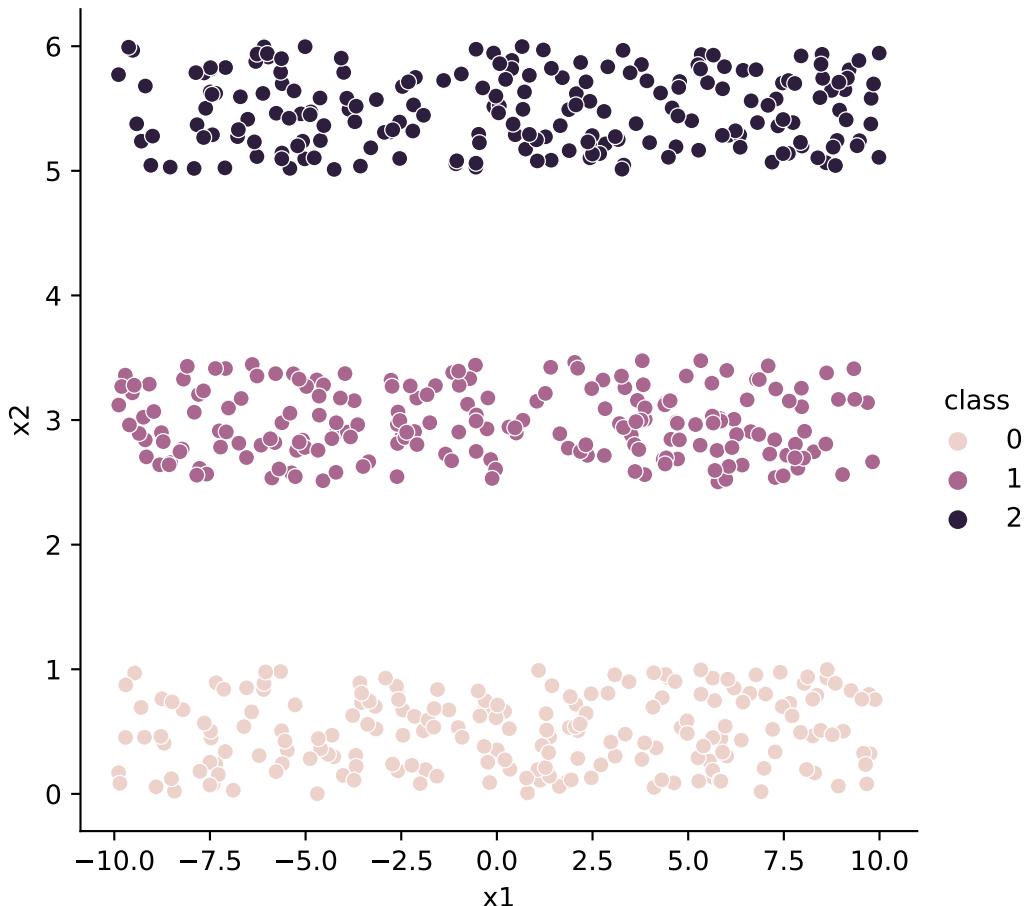


- **stripes** In this dataset, there is clear separation along one axis and a continuous blob

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, "class": cls})

stripes = stripes_data()
sns.relplot(data=stripes, x="x1", y="x2", hue="class");
```



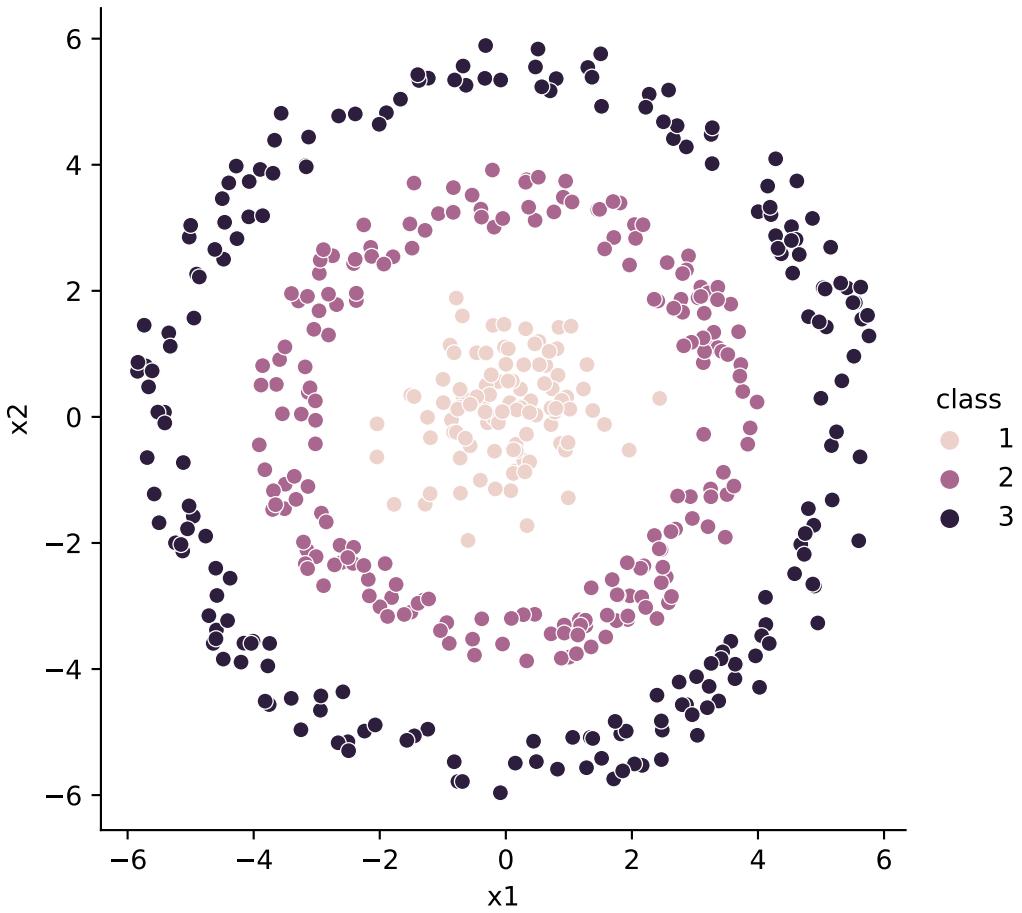
- **bullseye** This is the most challenging dataset, because the clusters can't be separated by anything like straight lines:

```

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))
    bullseye = pd.DataFrame( np.vstack((inner,middle,outer)), columns=["x1", "x2"] )
    bullseye["class"] = cls
    return bullseye

bullseye = bullseye_data()
p = sns.relplot(data=bullseye,
                  x="x1", y="x2",
                  hue="class"
                 )
p.set(aspect=1);

```



6.1 Similarity and distance

Given an $n \times d$ feature matrix, we want to define disjoint subsets of the \mathbf{x}_i such that the samples within a subset, or **cluster**, are more similar to one another than they are to members of other clusters.

The first decision we have to make is how to measure *similarity*. When a distance metric is available, we consider similarity to be inversely related to distance. For example, if we have defined a distance function between pairs of vectors as $\text{dist}(\mathbf{x}, \mathbf{y})$, then we could define similarity as

$$\text{sim}(\mathbf{x}, \mathbf{y}) = \exp \left[-\frac{\text{dist}(\mathbf{x}, \mathbf{y})^2}{2\sigma^2} \right].$$

Thus a distance of zero implies a similarity of 1, while the similarity tends to zero as distance increases. The scaling parameter σ controls the rate of decrease; for instance, when the distance is σ , the similarity is $e^{-1/2} \approx 0.6$.

There are ways to define similarity without making use of a distance, but we won't be using them.

6.1.1 Distance metrics

Definition 6.1. A **distance metric** is a function dist on pairs of vectors that satisfies the following properties for all vectors:

1. $\text{dist}(\mathbf{x}, \mathbf{y}) = 0$ if and only if $\mathbf{x} = \mathbf{y}$,
2. $\text{dist}(\mathbf{x}, \mathbf{y}) = \text{dist}(\mathbf{y}, \mathbf{x})$, and
3. $\text{dist}(\mathbf{x}, \mathbf{y}) \leq \text{dist}(\mathbf{x}, \mathbf{z}) + \text{dist}(\mathbf{z}, \mathbf{y})$, known as the triangle inequality.

These are considered the essential axioms of a distance metric. From them, you can also deduce that the distance function is always nonnegative.

Danger

The term *distance metric* isn't always used carefully to mean a function satisfying the three axioms, however, and some applications use a metric that does not satisfy the triangle inequality.

We already have the distance metric defined by

$$\text{dist}(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|$$

for any vector norm.

Another proper distance metric is **angular distance**. Generalizing from 2D and 3D vector geometry, we define the angle θ between vectors \mathbf{u} and \mathbf{v} in \mathbb{R}^d by

$$\cos(\theta) = \frac{\mathbf{u}^T \mathbf{v}}{\|\mathbf{u}\|_2 \|\mathbf{v}\|_2}. \quad (6.1)$$

Then the quantity θ/π is a distance metric. Because \arccos is a relatively expensive computational operation, though, it's common to use **cosine similarity**, defined as $\cos(\theta)$, and the related **cosine distance** $\frac{1}{2}[1 - \cos(\theta)]$, even though the latter does not satisfy the triangle inequality.

Categorical variables can be included in distance metrics. An ordinal variable is easily converted to equally spaced numerical values, which then may get a standard treatment. Nominal features are often compared using **Hamming distance**, which is just the total number of features that have different values in the two vectors.

6.1.2 Probability distributions

Definition 6.2. A **discrete probability distribution** is a vector \mathbf{x} whose components are nonnegative and satisfying $\|\mathbf{x}\|_1 = 1$.

Such a vector can be interpreted as frequencies or probabilities of observing different classes, for example. We already encountered one way to measure the dissimilarity of two probability distributions, the *cross-entropy*:

$$\text{CE}(\mathbf{x}, \mathbf{y}) = - \sum_{i=1}^d x_i \log(y_i).$$

A related measure is the **Kullback–Leibler (KL) divergence** or *relative entropy*,

$$\text{KL}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^d x_i \log\left(\frac{x_i}{y_i}\right).$$

Whenever $0 \cdot \log(0)$ is encountered in the CE or KL definitions, it equals zero, in accordance with its limiting value from calculus.

Neither cross-entropy nor KL divergence are symmetric in their arguments. But there is a related value called **information radius**, defined as

$$\text{IR}(\mathbf{u}, \mathbf{v}) = \frac{1}{2} [\text{KL}(\mathbf{u}, \mathbf{z}) + \text{KL}(\mathbf{v}, \mathbf{z})]$$

where $\mathbf{z} = (\mathbf{u}+\mathbf{v})/2$. Typically one uses a base-2 logarithm, in which case IR ranges between 0 and 1. The square root of IR is a distance metric.

Example 6.1. Let $\mathbf{u} = \frac{1}{4}[1, 3]$ and $\mathbf{v} = \frac{1}{4}[3, 1]$. Then $\mathbf{z} = (\mathbf{u} + \mathbf{v})/2 = [\frac{1}{2}, \frac{1}{2}]$, and

$$\begin{aligned}\text{KL}(\mathbf{u}, \mathbf{z}) &= \frac{1}{4} \cdot \log\left(\frac{1/4}{1/2}\right) + \frac{3}{4} \cdot \log\left(\frac{3/4}{1/2}\right) \\ &= \frac{1}{4} \cdot \log\left(\frac{1}{2}\right) + \frac{3}{4} \cdot \log\left(\frac{3}{2}\right) = -\frac{1}{4} + \frac{3}{4}(\log(3) - 1), \\ \text{KL}(\mathbf{v}, \mathbf{z}) &= \frac{3}{4}(\log(3) - 1) - \frac{1}{4}, \\ \text{IR}(\mathbf{u}, \mathbf{u}) &= \frac{3}{4}(\log(3) - 1) - \frac{1}{4} \approx 0.1887.\end{aligned}$$

6.1.3 Distance matrix

Definition 6.3. 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**

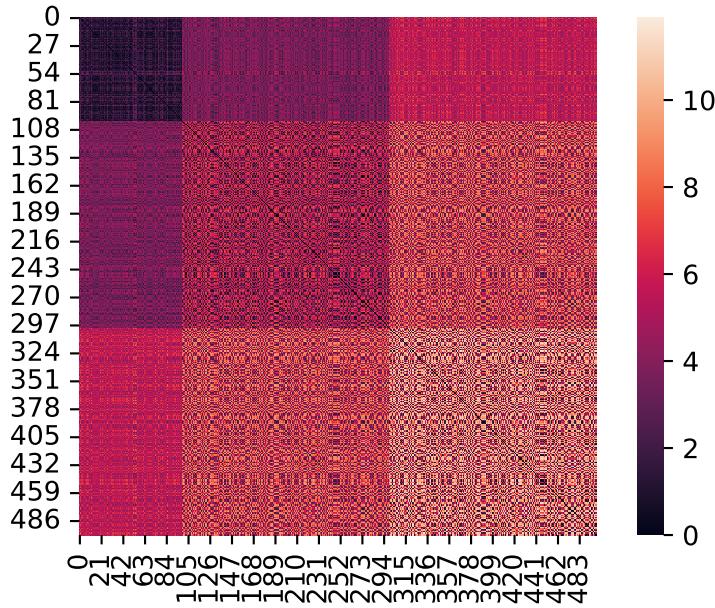
$$D_{ij} = \text{dist}(\mathbf{x}_i, \mathbf{x}_j).$$

Note that $D_{ii} = 0$ and $D_{ji} = D_{ij}$. Many clustering algorithms allow supplying \mathbf{D} in lieu of the feature vectors.

One can analogously define a *similarity matrix* using the Gaussian kernel. An advantage of similarity is that small values can be rounded down to zero. This has little effect on the results, but can create big gains in execution time and memory usage.

Example 6.2. The distance matrix of our bullseye dataset has some interesting structure:

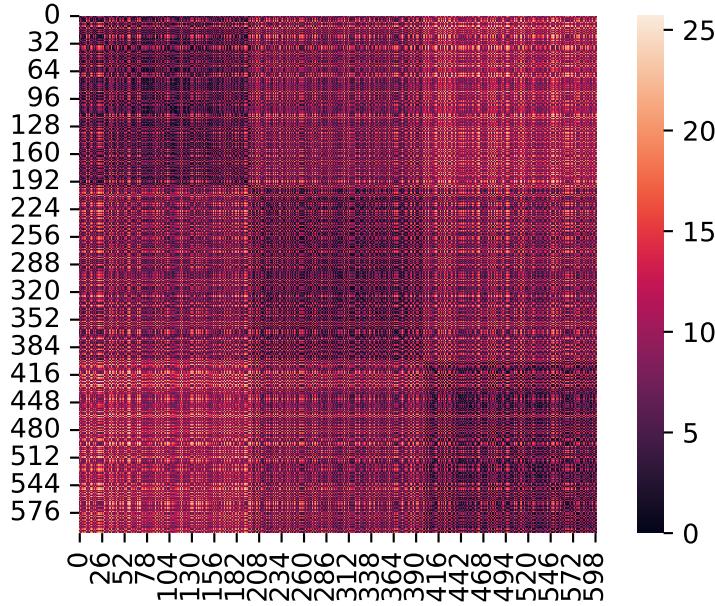
```
from sklearn.metrics import pairwise_distances
X = bullseye_data()[["x1", "x2"]]
D2 = pairwise_distances(X, metric="euclidean")    # use 2-norm metric
ax = sns.heatmap(D2)
ax.set_aspect(1);
```



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 a little interesting:

```
X = stripes_data()[["x1", "x2"]]
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.

6.1.4 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

$$x_1^2 + x_2^2 + \cdots + x_d^2 \leq 1,$$

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 unexpectedness 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 advice is easy to overstate. The curse is essentially about *randomly* chosen points, and it is correct that dimensions of noisy or irrelevant features will make many learning algorithms less effective. But if features carry useful information, adding them usually makes matters better, not worse.

6.2 Performance measures

Before we start generating clusterings, we need to decide how we will evaluate them. Recall that a clustering is simply a partitioning of the sample points into disjoint subsets. If a classification of the samples is available, then it automatically implies a clustering: divide the samples into subsets determined by class membership.

We will use some nonstandard terminology that makes the definitions a bit easier to state and read.

Definition 6.4. 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

If a trusted or reference clustering is available, then we can compare it to any other clustering result. This allows us to use classification datasets as proving grounds for clustering, although the problems of classification and clustering have different goals (separation versus similarity).

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. Noting that there are $\binom{n}{2}$ distinct pairs of n sample points, we define the **Rand index** by

$$\text{RI} = \frac{b + s}{\binom{n}{2}}.$$

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 the classifier over all pairs of samples.

Example 6.3. Suppose that samples x_1, x_2, x_4 are classified as blue, and x_3, x_5 are classified as red. Let's compute the Rand index relative to the reference classification for the clustering $A = \{x_1, x_2\}$ and $B = \{x_3, x_4, x_5\}$.

Here is a table showing which pairs of samples are buddies in both clusterings (indicated as TP), strangers in both (TN), or neither (F).

	x_1	x_2	x_3	x_4	x_5
x_1		TP	TN	F	TN
x_2			TN	F	TN
x_3				F	TP
x_4					F
x_5					

Hence the Rand index is $6/10 = 0.6$.

The Rand index has some attractive features:

- 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.
- The value is between 0 (complete disagreement) and 1 (complete agreement).

A weakness of the Rand index is that it can be fairly close to 1 even for a random clustering. The **adjusted Rand index** is

$$\text{ARI} = \frac{\text{RI} - E[\text{RI}]}{\max(\text{RI}) - E[\text{RI}]},$$

where the mean and max operations are taken over all possible clusterings. (These values can be worked out exactly by combinatorics.) The value can be negative. An ARI of 0 indicates no better agreement than a random clustering, and an ARI of 1 is complete agreement.

6.2.2 Silhouettes

If no reference clustering is available, then we must use an intrinsic measurement to assess quality. Suppose \mathbf{x}_i is a sample point. 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

$$s_i = \frac{\bar{r}_i - \bar{b}_i}{\max\{\bar{r}_i, \bar{b}_i\}}.$$

This value is between -1 (bad) and 1 (good) for every sample point. A **silhouette score** is derived by taking a mean of the silhouette values, either per cluster or overall depending on the usage.

Example 6.4. 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, the silhouette scores of the clusters, and the overall silhouette score.

x_i	\bar{b}_i	\bar{r}_i	s_i
-4	$\frac{3+5}{2}$	$\frac{6+10}{2}$	$\frac{8-4}{8} = \frac{1}{2}$
-1	$\frac{3+2}{2}$	$\frac{3+7}{2}$	$\frac{5-2.5}{5} = \frac{1}{2}$
1	$\frac{5+2}{2}$	$\frac{1+5}{2}$	$\frac{3-3.5}{3.5} = -\frac{1}{7}$
2	$\frac{4}{1}$	$\frac{6+3+1}{3}$	$\frac{(10/3)-4}{(10/3)} = -\frac{1}{6}$
6	$\frac{4}{1}$	$\frac{10+7+5}{3}$	$\frac{(22/3)-4}{(22/3)} = \frac{5}{11}$

The silhouette score of cluster A is

$$\frac{1}{3} \left(\frac{1}{2} + \frac{1}{2} - \frac{1}{7} \right) \approx 0.286,$$

and of cluster B is

$$\frac{1}{2} \left(\frac{5}{11} - \frac{1}{6} \right) \approx 0.144.$$

The overall score is the mean of the five values in the last column, which is about 0.229 .

The silhouette score is fairly easy to understand and use. However, it relies on distances and tends to favor convex, compact clusters.

Example 6.5. 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:

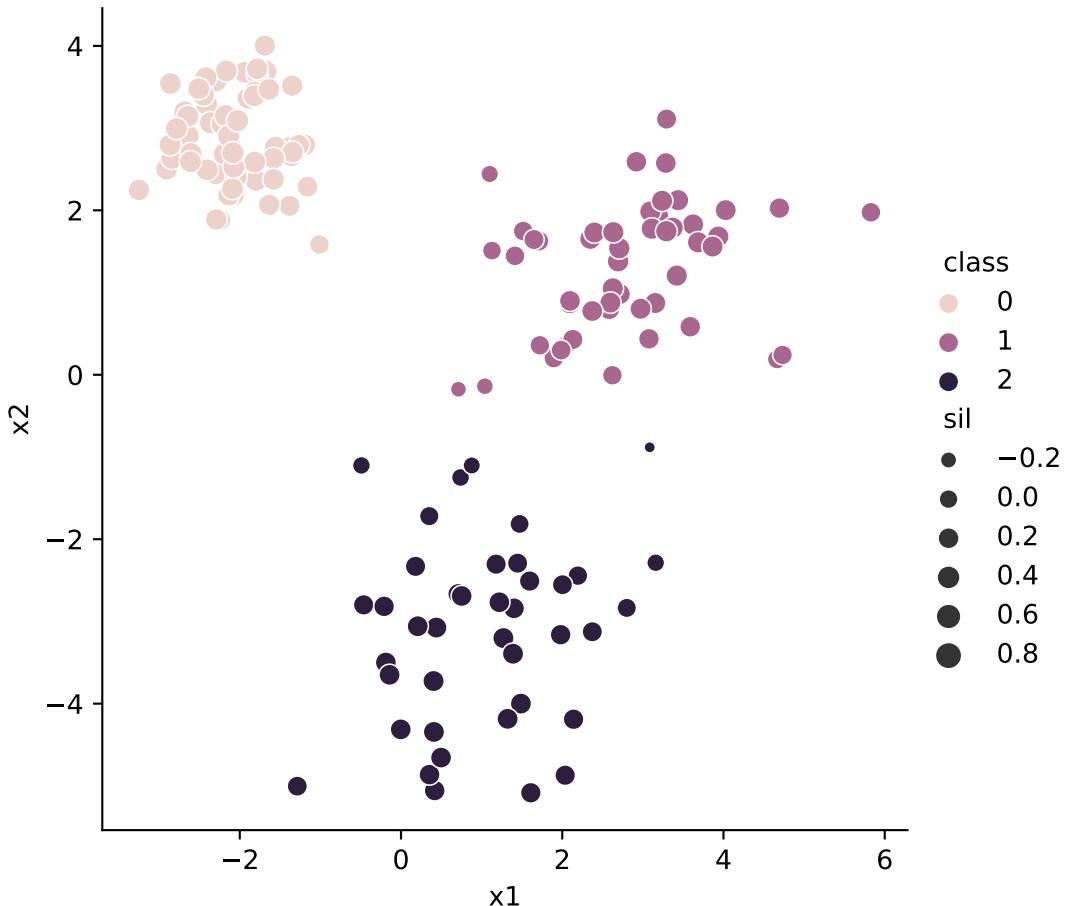
```
from sklearn.metrics import silhouette_samples
```

```

blobs = blobs_data()
X = blobs.drop("class", axis=1)

blobs["sil"] = silhouette_samples(X, blobs["class"])
sns.relplot(data=blobs,
            x="x1", y="x2",
            hue="class", size="sil"
            );

```



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:

```

blobs.groupby("class")["sil"].mean()

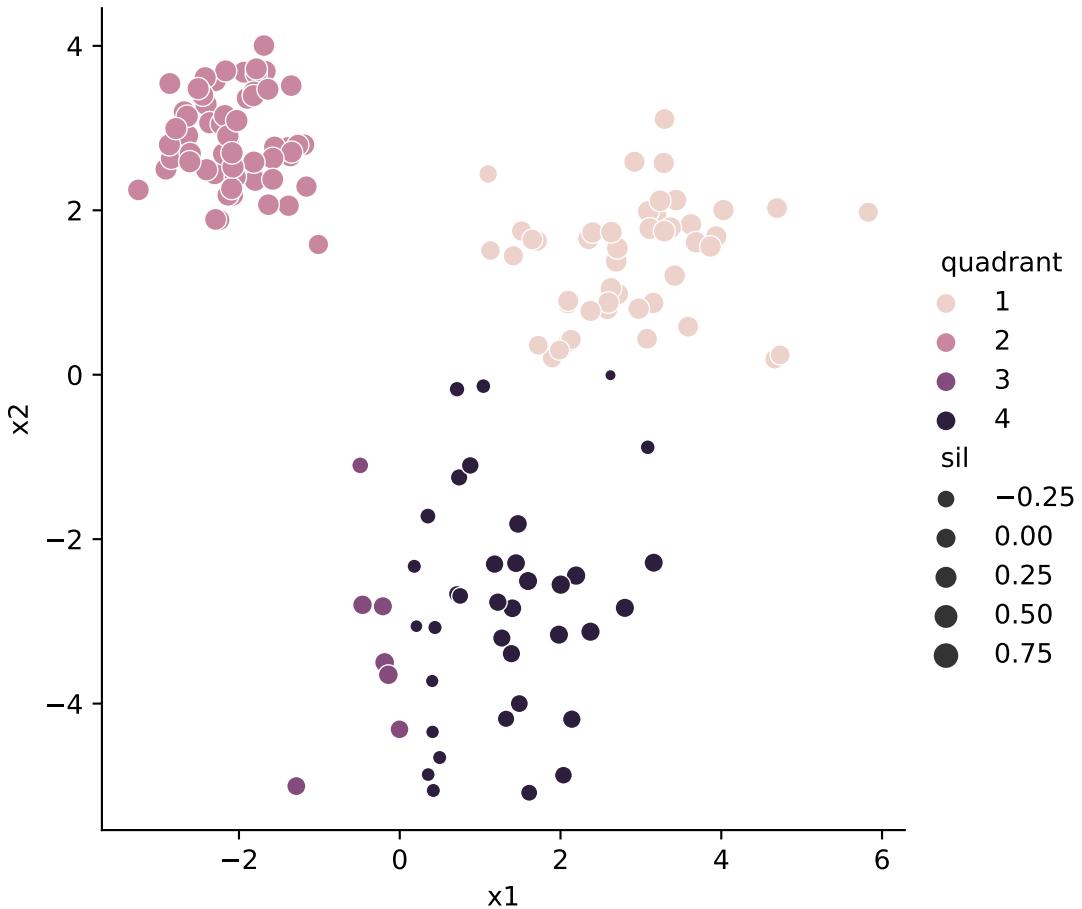
```

	sil
class	
0	0.815086
1	0.641591
2	0.582871

These values are ordered as we would expect. Now let's create another clustering based on the quadrants of the plane:

```
def quad(x,y):
    if x > 0:
        if y > 0: return 1
        else: return 4
    else:
        if y > 0: return 2
        else: return 3

blobs["quadrant"] = [quad(x,y) for (x,y) in zip(blobs.x1, blobs.x2)]
blobs["sil"] = silhouette_samples(X, blobs["quadrant"])
sns.relplot(data=blobs,
            x="x1", y="x2",
            hue="quadrant", size="sil"
            );
```



```
blobs.groupby("quadrant")["sil"].mean()
```

quadrant	sil
1	0.654031
2	0.816362
3	0.357247
4	0.095618

Even though the original clustering had three classes, and there are four quadrants, we can still compare them by adjusted Rand index:

```
from sklearn.metrics import adjusted_rand_score
adjusted_rand_score(blobs["class"], blobs["quadrant"])
```

```
0.904092765401111
```

Not surprisingly, they are seen as fairly similar.

Example 6.6. `sklearn` has a well-known dataset that contains labeled handwritten digits. Let's extract the examples for just the numerals 4, 5, and 6:

```
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()
```

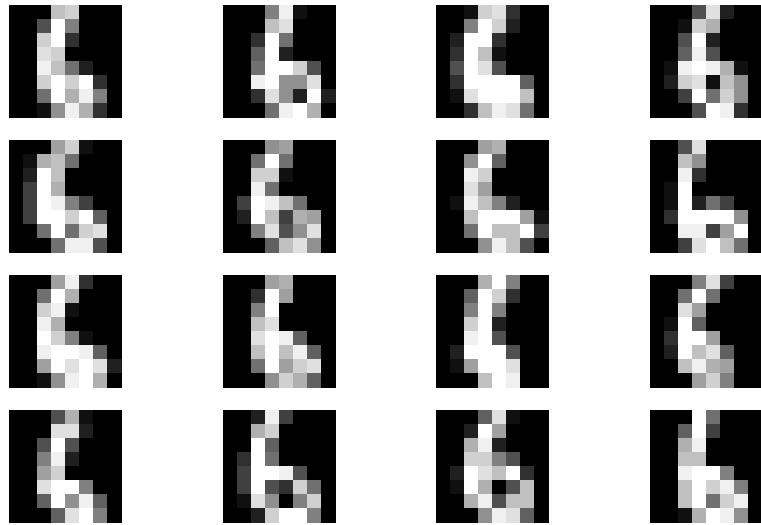
target	
5	182
4	181
6	181

We can visualize the raw data. Here are some of the 6s:

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

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.iloc[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==6])
```



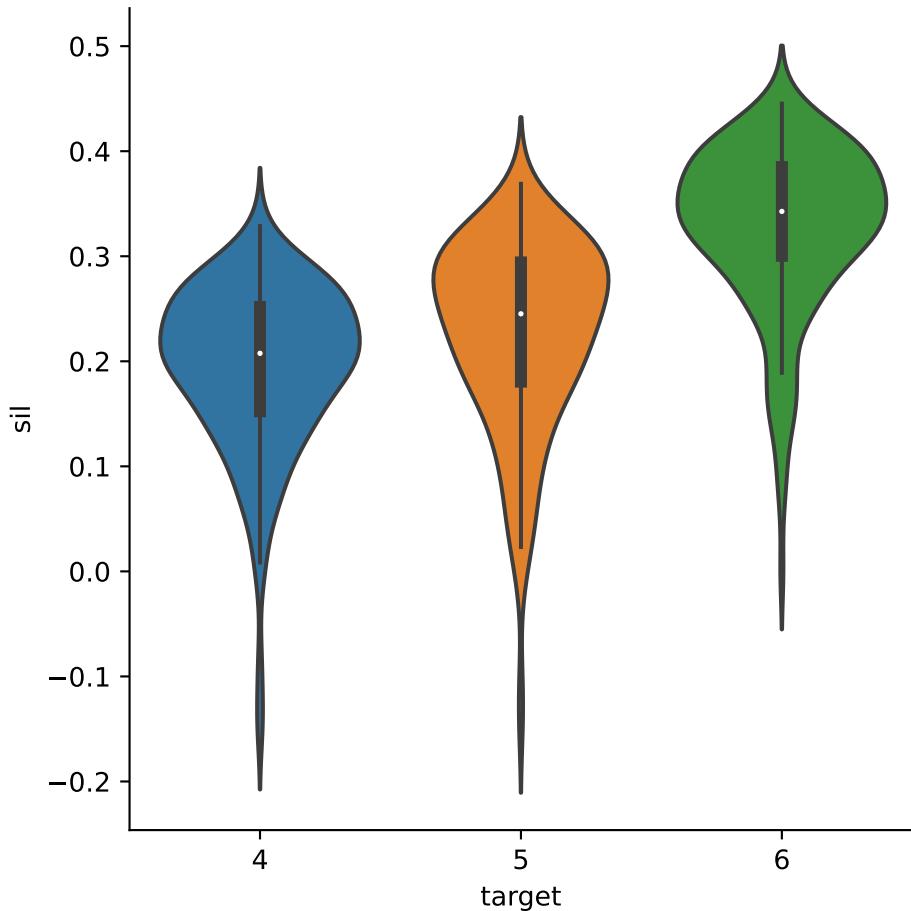
A clustering method won't be able to learn from the ground truth labels. In order to set expectations, we should see how well the originally labels cluster the samples. Here are the mean silhouette scores for the clusters.

```
digits["sil"] = silhouette_samples(X,y)
digits.groupby("target")["sil"].mean()
```

	sil
target	
4	0.194477
5	0.225677
6	0.327088

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. The existence of values close to and below zero suggest that a clustering algorithm might reconstruct the classification to some extent, but the ground truth may represent something more than geometric distances in feature space.

! 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 just requires us to separate different classes of examples, clustering is more specific and more demanding: examples 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, even within the training data, will look ambiguous.

6.3 k-means

The **k -means algorithm** is one of the best-known and most widely used clustering methods, although it has some serious limitations and drawbacks.

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 preselected hyperparameter. Cluster j has a **centroid** μ_j , which is the mean of the points in C_j . Define the **inertia** of C_j as

$$I_j = \sum_{\mathbf{x} \in C_j} \|\mathbf{x} - \mu_j\|_2^2.$$

The goal of the algorithm is to choose the clusters in order to minimize the total inertia,

$$I = \sum_{j=1}^k I_j.$$

Example 6.7. Let $k = 2$. Given the values $-3, -2, -1, 2, 5, 7$, we might cluster $\{-3, -2, -1\}$ and $\{2, 5, 7\}$. The total inertia is then

$$[(-3+2)^2 + (-2+2)^2 + (-1+2)^2] + \left[\left(2 - \frac{14}{3}\right)^2 + \left(5 - \frac{14}{3}\right)^2 + \left(7 - \frac{14}{3}\right)^2 \right] = 2 + \frac{124}{9} = 15.78.$$

If we instead cluster as $\{-3, -2, -1, 2\}$ and $\{5, 7\}$, then the 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 from a starting clustering.

6.3.1 Lloyd's algorithm

The standard method is known as **Lloyd's algorithm**. Starting with values for the k centroids, there is an iteration consisting of two steps:

- **Assignment** Each sample point is assigned to the cluster whose centroid is the nearest. (Ties are broken randomly.)
- **Update** Recalculate the centroids based on the cluster assignments:

$$\mu_j^+ = \frac{1}{|C_j|} \sum_{\mathbf{x} \in C_j} \mathbf{x}.$$

The algorithm stops when 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](#)

While Lloyd's algorithm will find a local minimum of total inertia, in the sense that small changes cannot decrease it, there is no guarantee of converging to the global minimum.

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 better/more reliable heuristics, such as *k-means++*, have since become more 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, and the run with the lowest inertia is kept.
- **Selection of k .** The algorithm treats k as a hyperparameter. 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."
- **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.
- **Shape effects.** Because of the dependence on the norm, the inertia criterion disfavors long, skinny clusters and clusters of unequal dispersion. Basically, it wants to find spherical blobs (as defined by the metric) of roughly equal size.

Example 6.8. Let's generate some test blobs:

We start k -means with $k = 2$ clusters, not presupposing prior knowledge of how the samples were created.

```

from sklearn.cluster import KMeans

X = blobs_data() [["x1", "x2"]]
km2 = KMeans(n_clusters=2, n_init="auto")
km2.fit(X)

KMeans(n_clusters=2, n_init='auto')

```

The fitted clustering object can tell how many iterations were required, and what the final inertia and cluster centroids are:

```

print("k=2 took", km2.n_iter_, "iterations")
print(f"final inertia: {km2.inertia_:.5g}")
print("cluster centroids:")
print(km2.cluster_centers_)

```

```

k=2 took 4 iterations
final inertia: 919.12
cluster centroids:
[[ 1.22650329 -2.88445195]
 [ 0.03501811  2.21111415]]

```

There is a `predict` method that can make cluster assignments for arbitrary points in feature space. In k-means, this just tells you which centroid is closest, i.e., the cluster membership:

```

km2.predict([ [-2,-1], [1,2] ])

array([0, 1], dtype=int32)

```

For the training samples we don't need to call `predict`. Every fitted clustering object has a `labels_` property that lists the cluster index values. We can use those labels to compute silhouette scores:

```

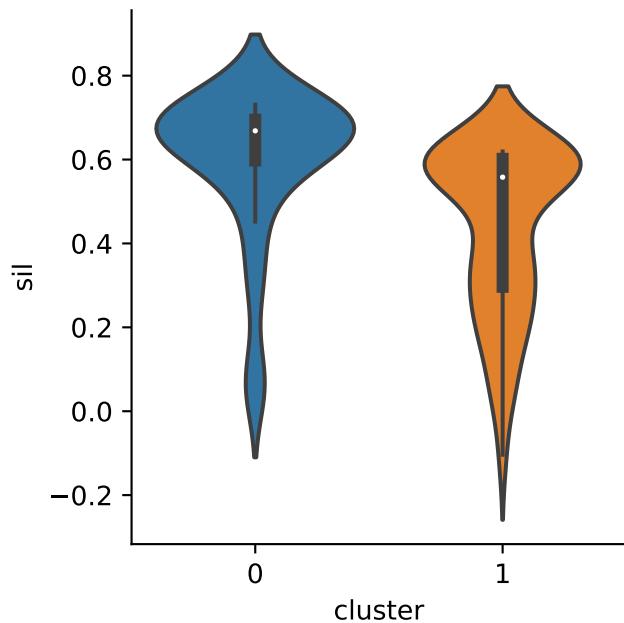
def report(clustering):
    blobs["cluster"] = clustering.labels_
    blobs["sil"] = silhouette_samples(X, blobs["cluster"])
    print(f"inertia: {clustering.inertia_:.5g}")
    print(f"overall silhouette score: {blobs['sil'].mean():.5g}")
    sns.catplot(data=blobs,
                x="cluster", y="sil",

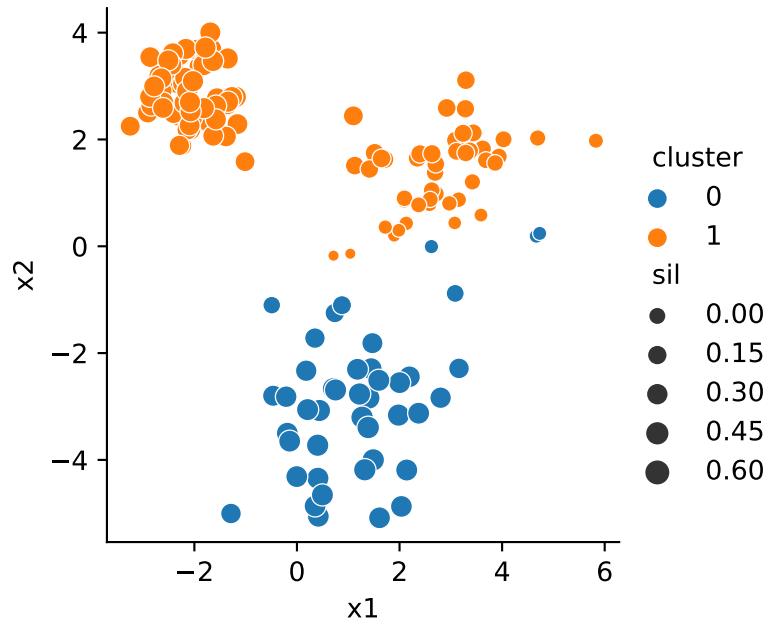
```

```
        kind="violin", height=3.5
    );
sns.relplot(data=blobs,
    x="x1", y="x2",
    hue="cluster", size=blobs["sil"], height=3.5
);
return blobs

report(km2);
```

```
inertia: 919.12
overall silhouette score: 0.48147
```

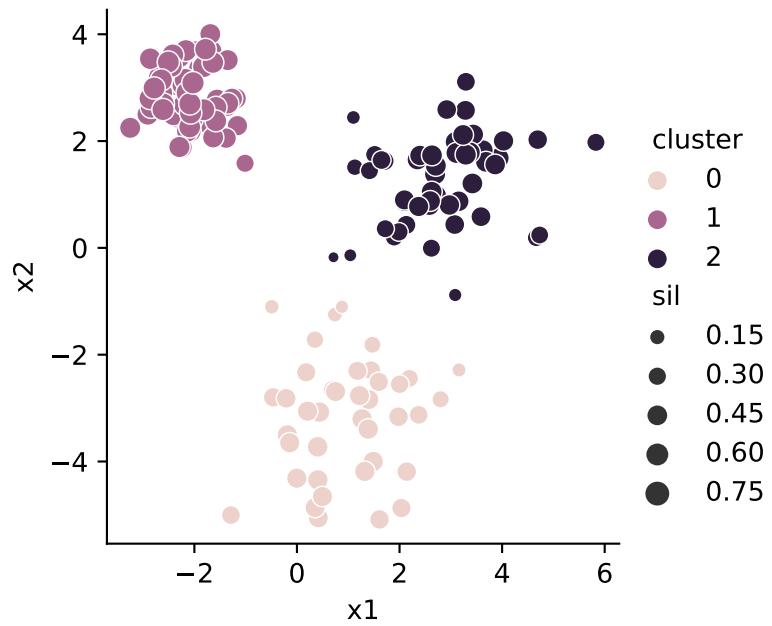
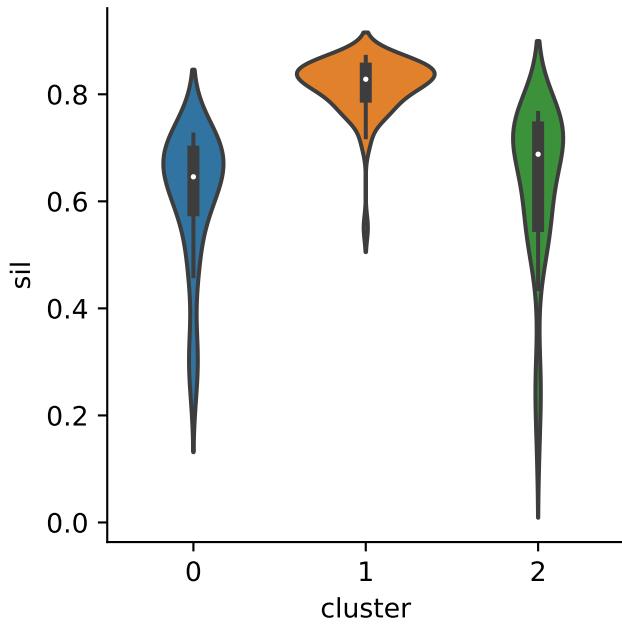




It's clear in both plots that cluster 0 is more tightly packed than cluster 1. Let's repeat the computation for $k = 3$ clusters:

```
km3 = KMeans(n_clusters=3, n_init="auto")
km3.fit(X)
report(km3);
```

```
inertia: 203.3
overall silhouette score: 0.69987
```

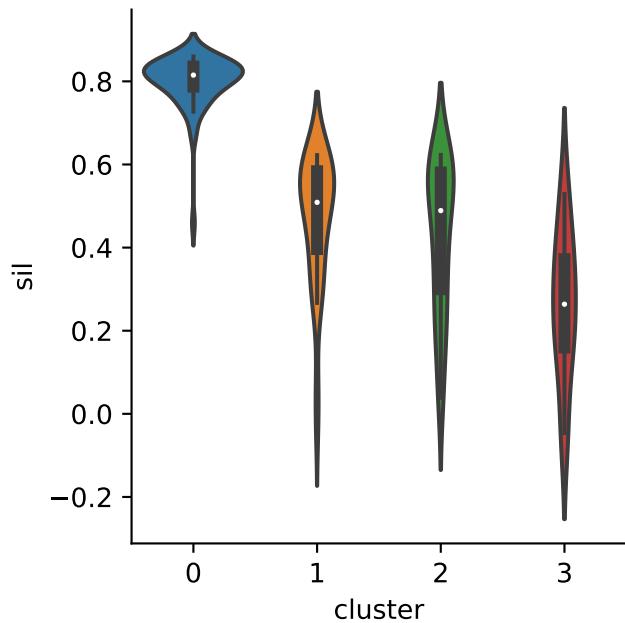


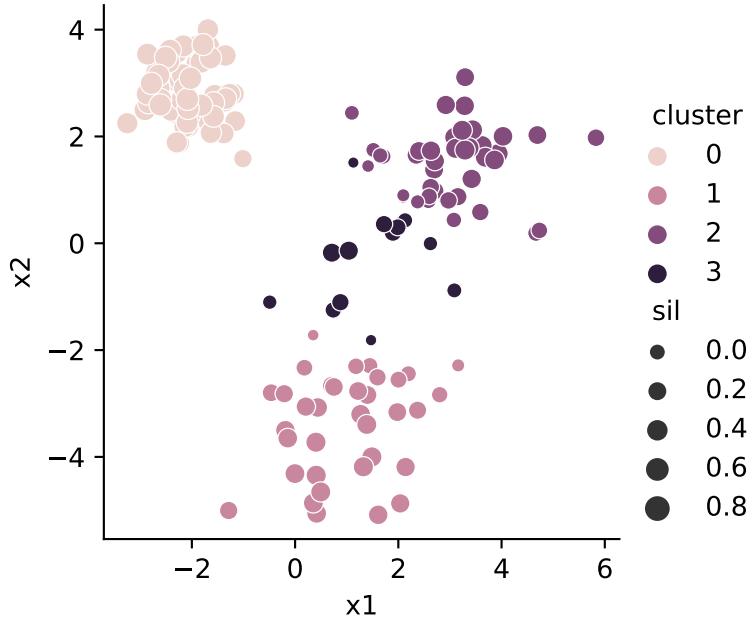
This result shows a modest reduction in silhouette scores for original good cluster, but improvement for the problematic one.

Moving on to $k = 4$ clusters shows clear degradation of the silhouette scores:

```
km4 = KMeans(n_clusters=4, n_init="auto")
km4.fit(X)
report(km4);
```

```
inertia: 172.8
overall silhouette score: 0.56995
```





Based on silhouette scores, then, we would probably stop at $k = 3$ clusters.

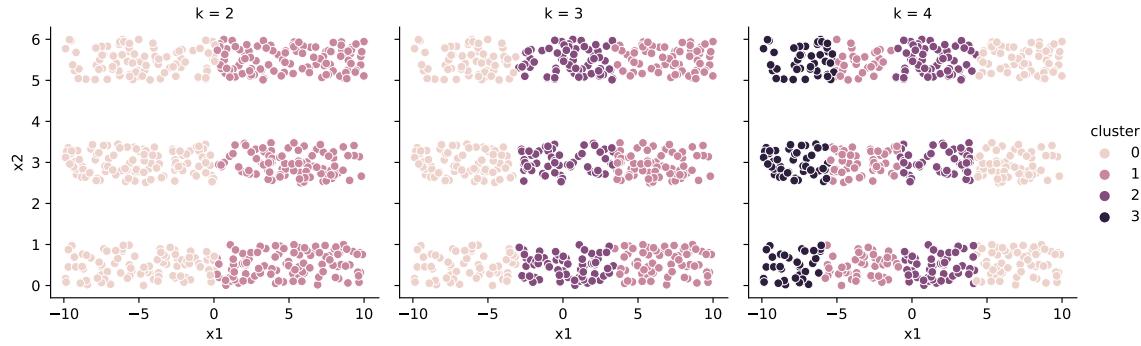
Example 6.9. K-means is expecting to find roughly spherical clusters. When the data do not conform to that model, it tends to perform poorly:

```

stripes = stripes_data()
X = stripes[["x1", "x2"]]
results = pd.DataFrame()
for k in [2,3,4]:
    km = KMeans(n_clusters=k, n_init="auto")
    km.fit(X)
    stripes["cluster"] = km.labels_
    stripes["k"] = k
    results = pd.concat( (results, stripes) )

sns.relplot(data=results,
             x="x1", y="x2",
             hue="cluster", col="k", height=3.5
            );

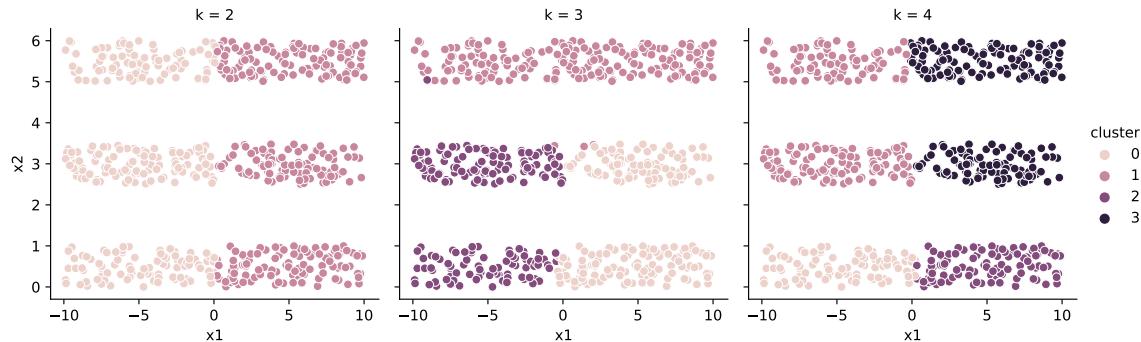
```



It's not a bad idea to standardize the data. But that's no panacea:

```
results = pd.DataFrame()
for k in [2,3,4]:
    km = make_pipeline(StandardScaler(), KMeans(n_clusters=k, n_init="auto"))
    km.fit(X)
    stripes["cluster"] = km[1].labels_
    stripes["k"] = k
    results = pd.concat( (results, stripes) )

sns.relplot(data=results,
             x="x1", y="x2",
             hue="cluster", col="k", height=3.5
            );
```



Clustering is hard!

Example 6.10. 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 fit 3 clusters to the feature matrix:

```

km = KMeans(n_clusters=3)
km.fit(X)
digits["kmeans3"] = km.labels_
digits[["target", "kmeans3"]].head(9)

```

	target	kmeans3
4	4	1
5	5	2
6	6	0
14	4	1
15	5	2
16	6	0
24	4	1
25	5	2
26	6	0

The adjusted Rand index suggests that we have reproduced the classification very well:

```

ARI = adjusted_rand_score(y, digits["kmeans3"])
print(f"ARI: {ARI:.4f}")

```

ARI: 0.9618

However, that conclusion benefits from our prior knowledge. What if we did not know how many clusters to look for? Let's look over a range of k values, recording the final total inertia and the mean silhouette score for each

```

from sklearn.metrics import silhouette_score
results = []
for k in range(2,8):
    km = KMeans(n_clusters=k, random_state=0)
    km.fit(X)

```

```

sil = silhouette_score(X, km.labels_)
results.append( [k, sil] )

pd.DataFrame(results, columns=["k", "mean silhouette"])

```

	k	mean silhouette
0	2	0.226800
1	3	0.251904
2	4	0.245855
3	5	0.235199
4	6	0.188324
5	7	0.170556

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 fairly close, 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 vector. Groups of similar samples are connected as nearby relatives in the tree, with less-similar groups located as more distant relatives.

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., $C_i = \{\mathbf{x}_i\}$. Then, the similarity or distance between each pair of clusters is determined. The pair with the minimum distance is merged, and the process repeats.

Common ways to define the distance between two clusters C_i and C_j are:

- **single linkage** (also called *minimum linkage*)

$$\min_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \{\|\mathbf{x} - \mathbf{y}\|\} \quad (6.2)$$

- **complete linkage** (also called *maximum linkage*)

$$\max_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \{\|\mathbf{x} - \mathbf{y}\|\} \quad (6.3)$$

- **average linkage**

$$\frac{1}{|C_i||C_j|} \sum_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \|\mathbf{x} - \mathbf{y}\| \quad (6.4)$$

- **Ward linkage** The increase in inertia resulting from merging C_i and C_j , equal to

$$\frac{|C_i||C_j|}{|C_i| + |C_j|} \|\mu_i - \mu_j\|_2^2, \quad (6.5)$$

where μ_i and μ_j are the centroids of C_i and C_j .

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.

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 clusters packed tightly together. Average linkage is a compromise between these extremes. All three of these options can work with a distance matrix in lieu of the original feature matrix.

Example 6.11. Given clusters $C_1 = \{-3, -2, -1\}$ and $C_2 = \{3, 4, 5\}$, we find the different linkages between them:

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 3×3 matrix:

	-3	-2	-1
3	6	5	4
4	7	6	5
5	8	7	6

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 6.12. 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.          , 4.12310563, 3.35410197, 3.60555128, 2.23606798],
       [4.12310563, 0.          , 2.6925824 , 4.47213595, 4.24264069],
       [3.35410197, 2.6925824 , 0.          , 1.80277564, 2.06155281],
       [3.60555128, 4.47213595, 1.80277564, 0.          , 1.41421356],
       [2.23606798, 4.24264069, 2.06155281, 1.41421356, 0.        ]])
```

The minimum value in the upper triangle of the distance matrix is in row 3, column 4 (starting index at 0). So our first merge results in the cluster $C_1 = \{\mathbf{x}_3, \mathbf{x}_4\}$. The next-smallest entry in the upper triangle is at position (2, 3), so we want to merge those samples together next, resulting in

$$C_1 = \{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}, C_2 = \{\mathbf{x}_0\}, C_3 = \{\mathbf{x}_1\}.$$

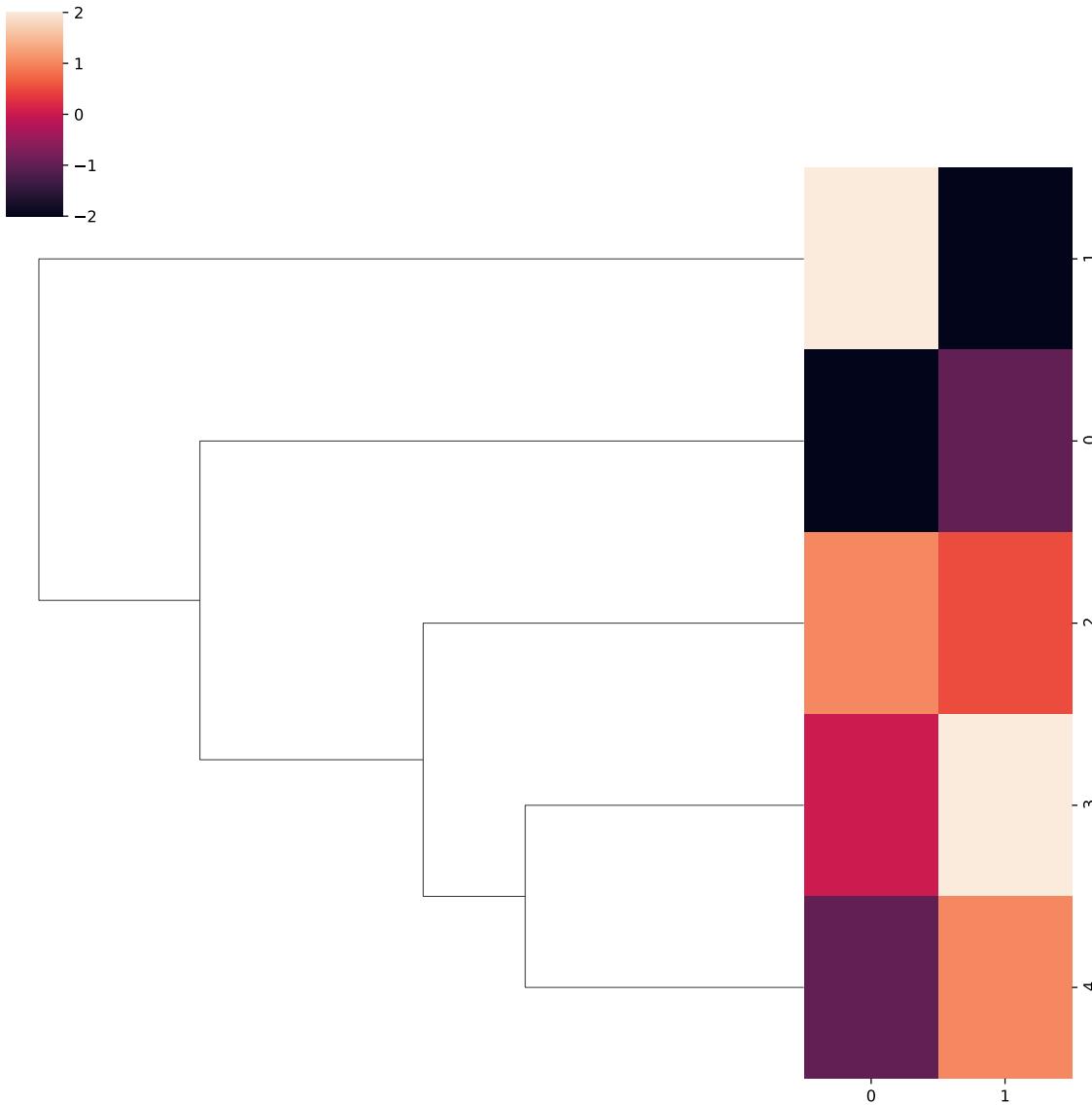
The next-smallest element in the matrix is at (2, 4), but those points are already merged, so we move on to position (0, 4). Now we have

$$C_1 = \{\mathbf{x}_0, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}, C_2 = \{\mathbf{x}_1\}.$$

The final merge is to combine these.

The entire dendrogram can be visualized with seaborn:

```
sns.clustermap(X,
                 col_cluster=False,
                 dendrogram_ratio=(.75,.15)
                 );
```



The horizontal position in the dendrogram above indicates the linkage strength. Note on the right that the ordering of the samples has been changed (so that the lines won't cross each other). The two colored columns show a heatmap of the two features of the sample points. Working from right to left, we see the merger of samples 3 and 4, which are then merged with sample 2, etc.

In effect, we get an entire family of clusterings by stopping at any linkage value we want. If we chose to stop at value 2.5, for instance, we would have two clusters of size 4 and 1. Or, if we predetermine that we want k clusters, we can stop after $n - k$ merge steps.

Example 6.13. We define a function that allows us to run all three linkages for a dataset:

```
from sklearn.cluster import AgglomerativeClustering

def run_experiment(data):
    results = pd.DataFrame()
    for linkage in ["single", "complete", "ward"]:
        agg = AgglomerativeClustering(n_clusters=3, linkage=linkage)
        agg.fit( data[["x1", "x2"]] )
        data["cluster"] = agg.labels_
        data["linkage"] = linkage
        results = pd.concat( (results, data) )
    return results
```

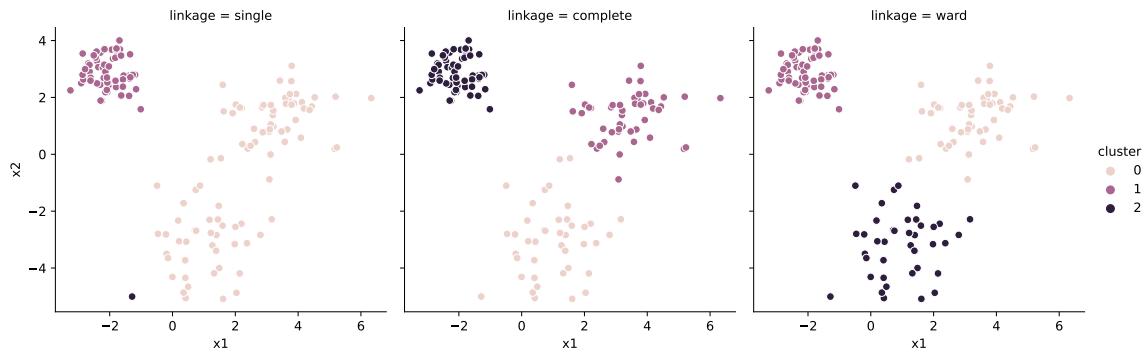
We first try the blobs seen previously:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(
    n_samples=[60, 50, 40],
    centers=[[ -2,3], [3.5,1.5], [1,-3] ],
    cluster_std=[0.5, 0.9, 1.2],
    random_state=19716
)
blobs = pd.DataFrame( {"x1": X[:,0], "x2": X[:,1], "class": y} )
blobs.head()
```

	x1	x2	class
0	4.436817	1.681397	1
1	3.193103	1.379978	1
2	1.400972	-2.840545	2
3	-2.147649	2.716864	0
4	0.438409	-3.074790	2

```
results = run_experiment(blobs)
sns.relplot(data=results,
             x="x1", y="x2",
             hue="cluster", col="linkage", height=4
            )
```

```
<seaborn.axisgrid.FacetGrid at 0x154bfda10>
```

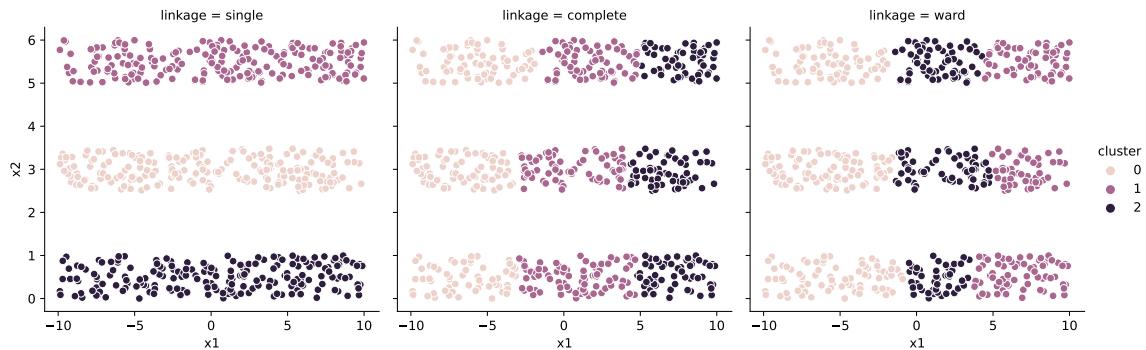


As you can see, the simple linkage was confused by the two blobs that nearly run together.

Next, we try data points lying in three distinct stripes:

```
stripes = stripes_data()
results = run_experiment(stripes)
sns.relplot(data=results,
            x="x1", y="x2",
            hue="cluster", col="linkage", height=4
            )
```

<seaborn.axisgrid.FacetGrid at 0x154794350>



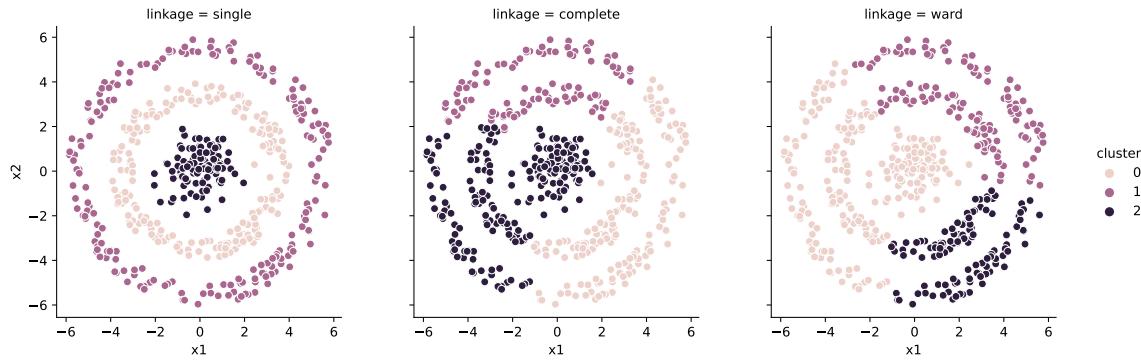
Both the complete and Ward linkages are committed to finding compact, roughly spherical clusters. They group together points across stripes rather than clusters extending lengthwise. The single linkage has more flexibility.

Finally, we try the most demanding test, points that are arranged in rings:

```

bullseye = bullseye_data()
results = run_experiment(bullseye)
p = sns.relplot(data=results,
                 x="x1", y="x2",
                 hue="cluster", col="linkage", height=4
                )
p.set(aspect=1);

```

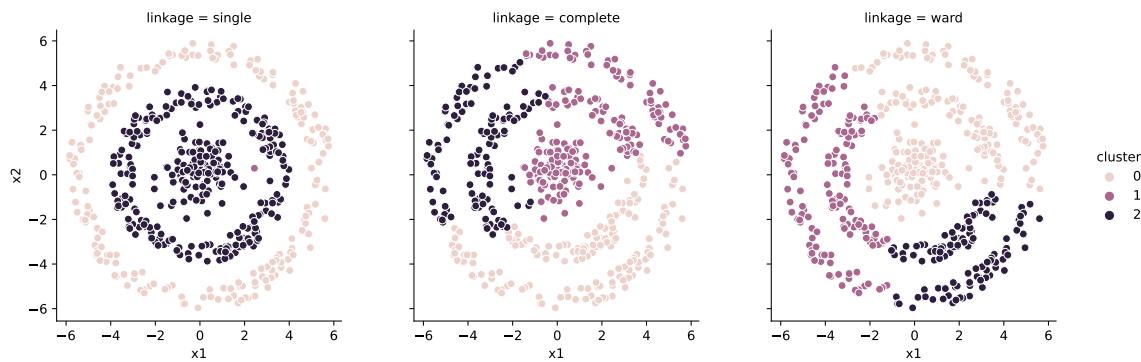


The single linkage is the only one with enough geometric flexibility to cluster the rings properly. However, it's a delicate situation, and it can be sensitive to individual samples. Here, we add just one point to the bullseye picture and get a big change:

```

bullseye = pd.concat( ( bullseye, pd.DataFrame({"x1": [0], "x2": [2.25]}) ) )
results = run_experiment(bullseye)
p = sns.relplot(data=results,
                 x="x1", y="x2",
                 hue="cluster", col="linkage", height=4
                )
p.set(aspect=1);

```



6.4.1 Case study: Penguins

Let's try agglomerative clustering to discover the species of the penguins. First, let's recall how many of each species we have.

```
penguins = sns.load_dataset("penguins").dropna()  
features = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]  
X = penguins[features]  
penguins["species"].value_counts()
```

species	
Adelie	146
Gentoo	119
Chinstrap	68

Our first attempt is single linkage. Because 2-norm distances are involved, we will 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)  
penguins["single"] = single.labels_ # cluster assignments  
penguins.loc[::-24,[ "species", "single"]] # print out some rows
```

	species	single
0	Adelie	0
29	Adelie	0
54	Adelie	0
78	Adelie	0
102	Adelie	0
126	Adelie	0
150	Adelie	0
174	Chinstrap	0
198	Chinstrap	0
222	Gentoo	2
247	Gentoo	2
271	Gentoo	2
296	Gentoo	2
320	Gentoo	2

It seems that 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 toy datasets in Example 6.13, the 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(penguins["species"], penguins["single"])
print(f"single linkage ARI: {ARI:.4f}")
```

single linkage ARI: 0.6506

Now let's try it with Ward linkage (the default):

```
ward = AgglomerativeClustering(n_clusters=2, linkage="ward")
pipe = make_pipeline(StandardScaler(), ward)
pipe.fit(X)
penguins["ward"] = ward.labels_

print("Ward linkage results:")
print(penguins["ward"].value_counts())
```

Ward linkage results:

0	214
1	119

Name: ward, dtype: int64

This result looks more promising. The ARI confirms that hunch:

```

ARI = adjusted_rand_score(penguins["species"], penguins["ward"])
print(f"Ward linkage ARI: {ARI:.4f}")

```

Ward linkage ARI: 0.6486

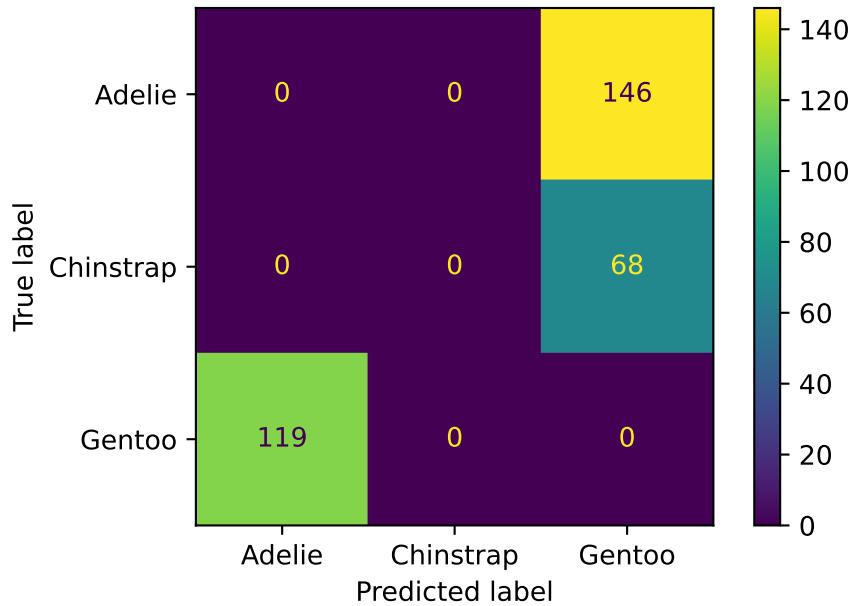
If we guess at the likely correspondence between the cluster numbers and the different species, then we can find the confusion matrix:

```

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
y = penguins["species"]
# Convert cluster numbers into labels:
y_hat = penguins["ward"].replace({1:"Adelie",0:"Gentoo",2:"Chinstrap"})

ConfusionMatrixDisplay(confusion_matrix(y,y_hat), display_labels=y.unique()).plot();

```



Exercises

Exercise 6.1. Using only the three axioms of a distance metric, prove that $\text{dist}(\mathbf{x}, \mathbf{y}) \geq 0$ for all vectors \mathbf{x} and \mathbf{y} . (Hint: apply the triangle inequality to go from \mathbf{x} to \mathbf{y} and back again.)

Exercise 6.2. Prove that the angular distance between any nonzero vector and itself is zero.

Exercise 6.3. Find a counterexample showing that cosine distance does not satisfy the triangle inequality. (Hint: it's enough to consider some simple vectors in two dimensions.)

Exercise 6.4. Let c be a positive number, and consider the 12 sample points $\{(\pm c, \pm j) : j = 1, 2, 3\}$. One way to cluster the sample points, which we designate as clustering α , is to split according to the sign of x_1 . Another way, which we designate as clustering β , is to split according to the sign of x_2 . Compute the inertia of both clusterings. For which values of c , if any, does clustering α have less inertia than clustering β ?

Exercise 6.5. Here is a distance matrix for points $\mathbf{x}_1, \dots, \mathbf{x}_5$.

$$\begin{bmatrix} 0 & 2 & 4 & 5 & 6 \\ 2 & 0 & 2 & 3 & 4 \\ 4 & 2 & 0 & 1 & 2 \\ 5 & 3 & 1 & 0 & 1 \\ 6 & 4 & 2 & 1 & 0 \end{bmatrix}$$

Compute the average linkage between the clusters with index sets $C_1 = \{1, 3\}$ and $C_2 = \{2, 4, 5\}$.

Exercise 6.6. Perform by hand an agglomerative clustering for the values 2, 4, 5, 8, 12 using single linkage. This means finding the four merge steps needed to convert five singleton clusters into one global cluster.

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
```

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**. A graph is a collection of **nodes** (also called *vertices*) and **edges** that connect pairs of nodes. A basic distinction in graph theory is between an **undirected graph**, in which the edge (a, b) is identical to (b, a) , and a **directed graph** or **digraph**, in which (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 seek to go beyond them. We also will not consider graphs that allow a node to link to itself.

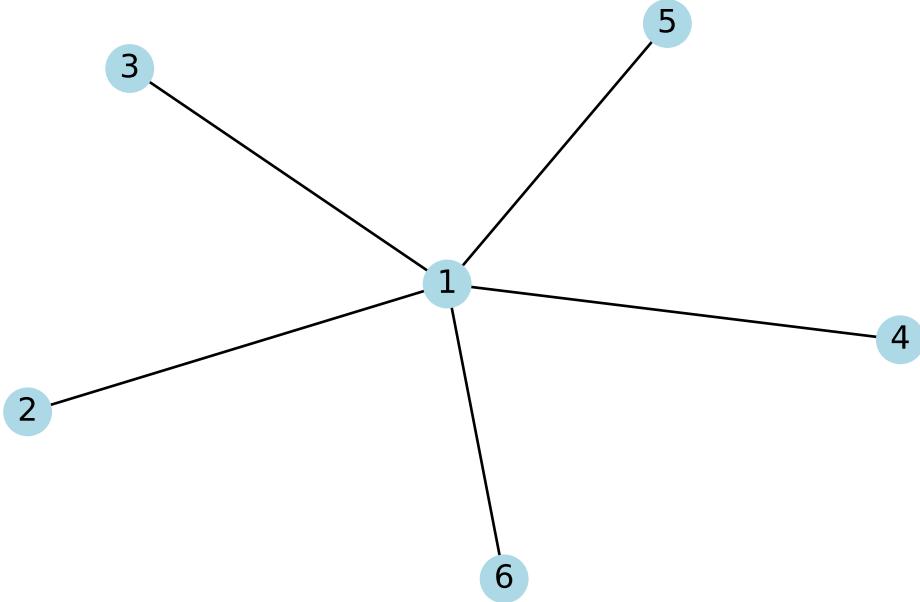
7.1.1 NetworkX

We will use the NetworkX package to work with graphs.

```
import networkx as nx
```

One way to create a graph is from a list of edges.

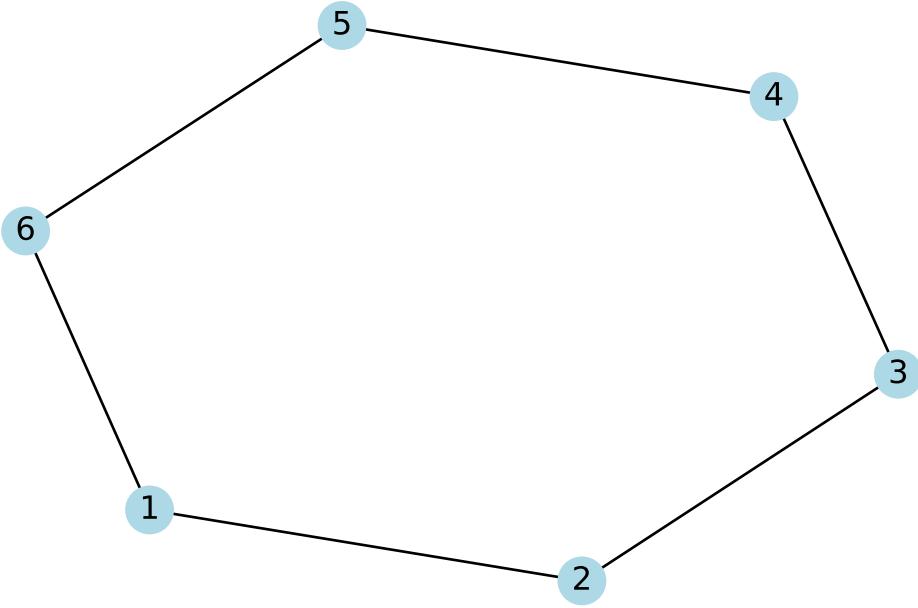
```
star = nx.Graph( [ (1,2),(1,3),(1,4),(1,5),(1,6) ] )
nx.draw(star, with_labels=True, node_color="lightblue")
```



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



We can conversely deconstruct a graph object into its nodes and edges. The results have special types that may need to 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 find out which nodes are **adjacent** to a given node, i.e., connected to it by an edge. The result is that node's list of **neighbors**.

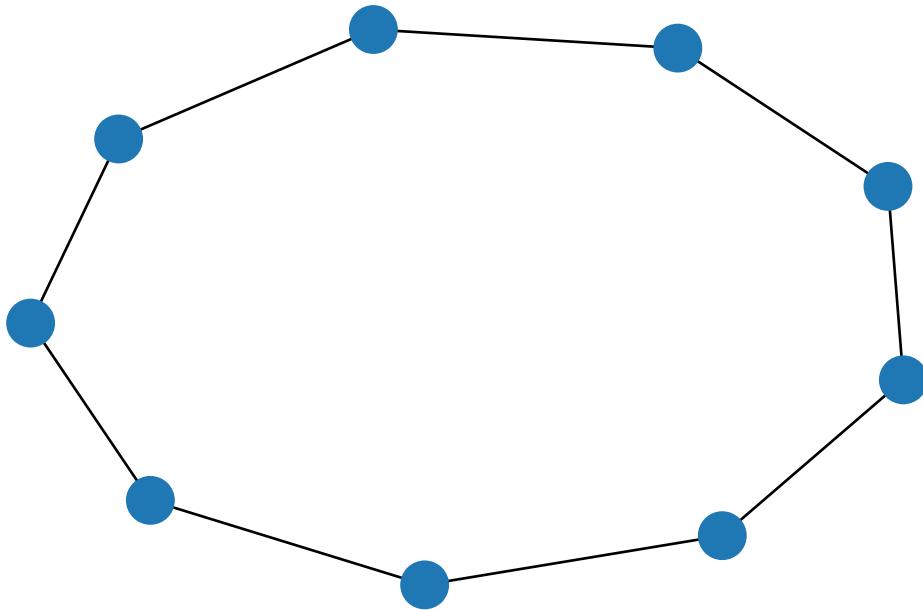
```
print( "Neighbors of node 3 in graph H:", list(H[3]) )
```

```
Neighbors of node 3 in graph H: [2, 4]
```

7.1.2 Common graph types

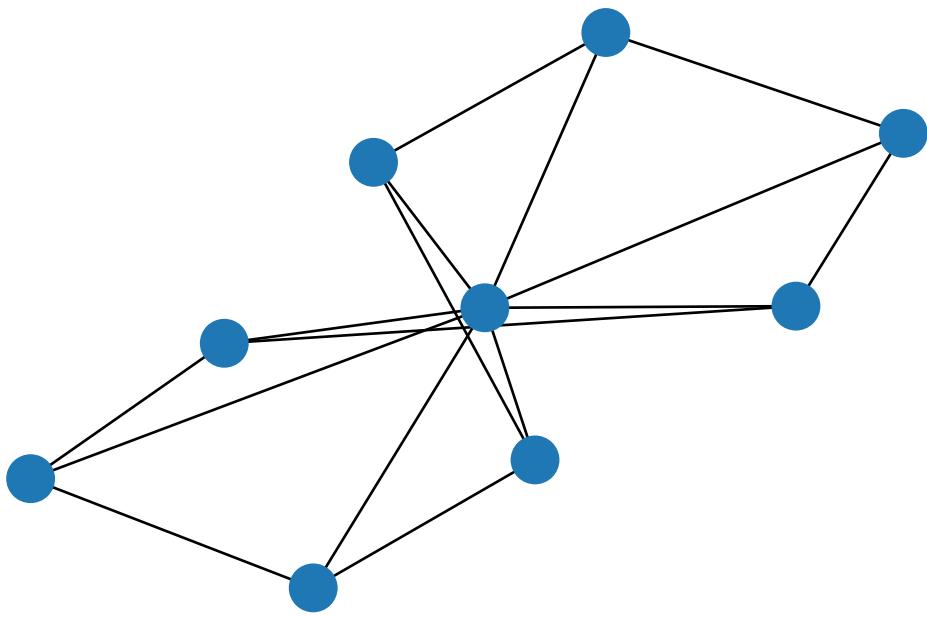
There are functions that generate different well-studied types of graphs. The first graph constructed above is a **star graph**, and the graph H above is a **cycle graph**.

```
nx.draw(nx.cycle_graph(9))
```



A cross between the star and the cycle is a **wheel graph**.

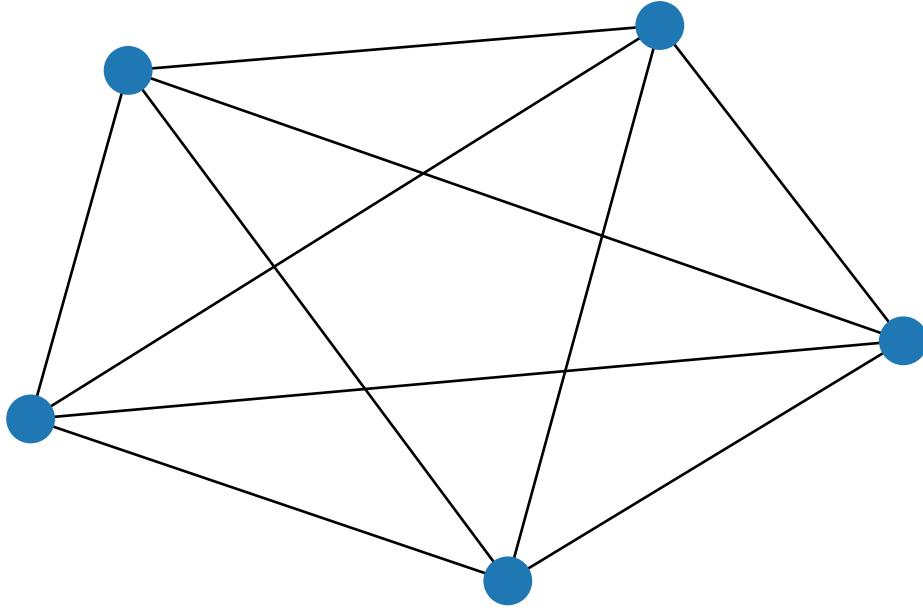
```
nx.draw(nx.wheel_graph(9))
```



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

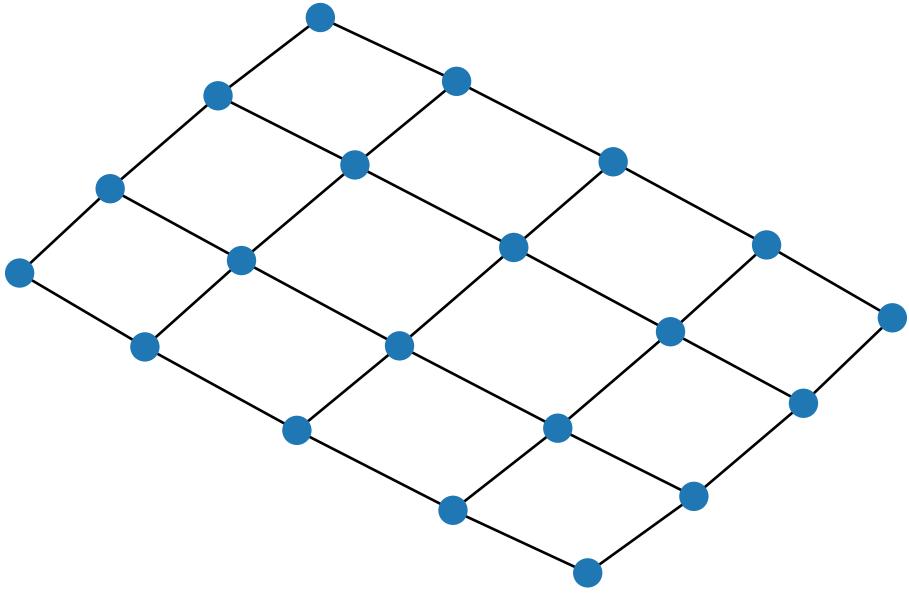
$$\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$$

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



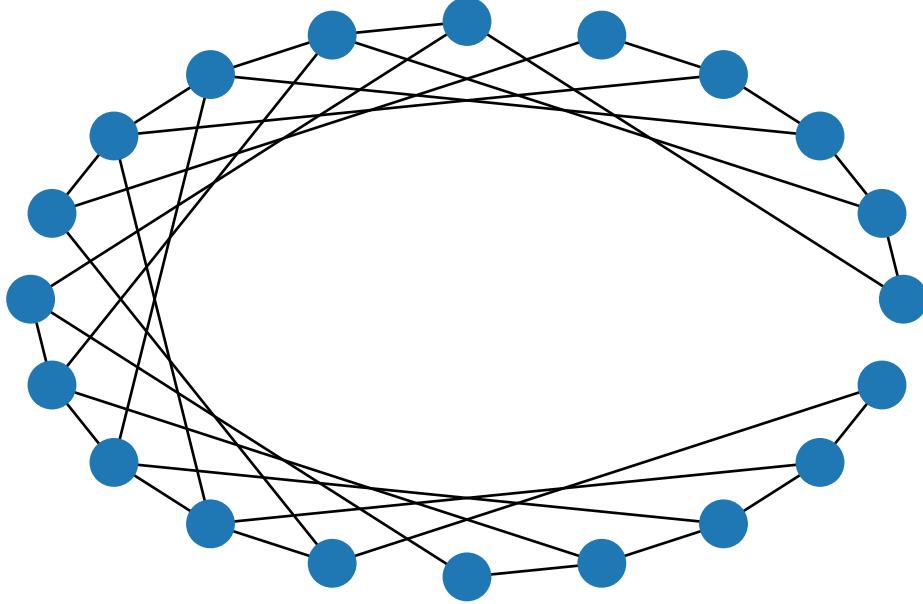
In an $m \times n$ lattice graph, there are $m - 1$ edges in one direction repeated n times, plus $n - 1$ edges in the other direction, repeated m times. Thus there are

$$(m - 1)n + (n - 1)m = 2mn - (m + n)$$

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)
```



As you can see, it's not easy to tell how similar two graphs are by comparing renderings of them.

7.1.3 Adjacency matrix

Every graph can be associated with an **adjacency matrix**. Suppose the nodes are numbered from 0 to $n - 1$. The adjacency matrix is $n \times n$ and has a 1 at position (i, j) if node i and node j are adjacent, and a 0 otherwise.

```
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],
       [1, 0, 0, 0, 0, 0]])
```

In an undirected graph, we have $A_{ij} = A_{ji}$ everywhere, and we say that A is *symmetric*.

7.1.4 Importing networks

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("musae_edges.csv", delimiter=',', nodetype=int)
```

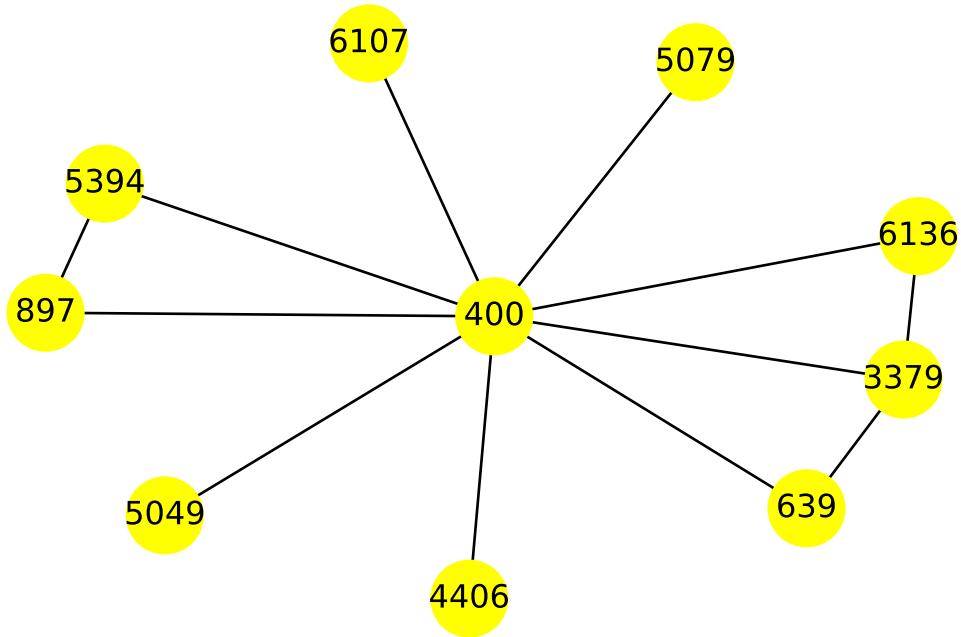
The file just imported has a pair of nodes representing one edge on each line. The nodes can have any names at all; by default they are interpreted as strings, which we overrode above to get integer node labels.

```
print("Twitch network has",
      twitch.number_of_nodes(),
      "nodes and",
      twitch.number_of_edges(),
      "edges"
     )
```

```
Twitch network has 7126 nodes and 35324 edges
```

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")
```

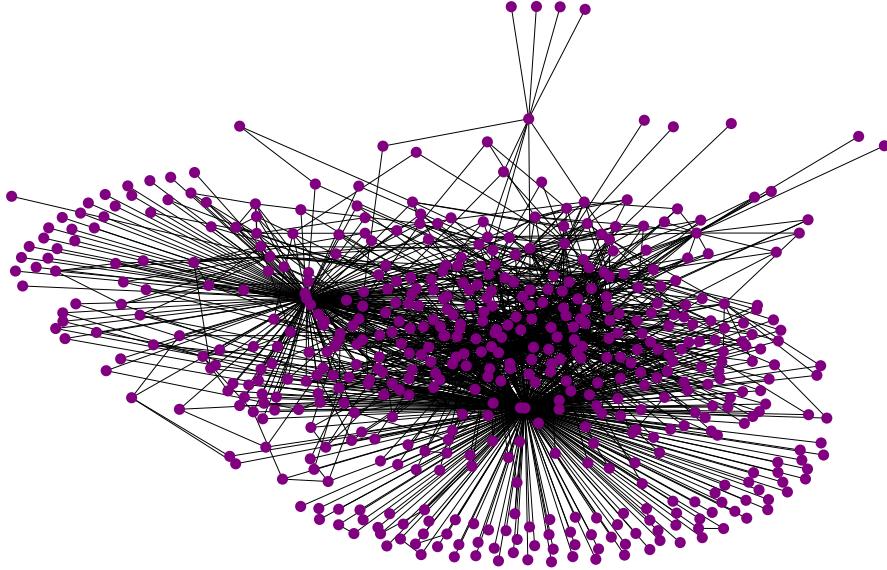


Notice that the nodes of the ego network have the same labels as they did in the graph that it was taken from. We can widen the ego graph to include the ego graphs of all the neighbors:

```
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.

7.1.5 Degree and average degree

The **degree** of a node is the number of edges that have the node as an endpoint. Equivalently, it is the number of nodes in its ego graph, minus the original node itself. The **average degree** of a graph is the mean of the degrees of all of its nodes.

The `degree` property of a graph gives a dictionary-style object of all nodes with their degrees.

```
ego.degree
```

```
DegreeView({897: 2, 400: 9, 5394: 2, 3379: 3, 4406: 1, 5079: 1, 6136: 2, 5049: 1, 6107: 1, ...})
```

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)  
print("average degree of ego graph:", degrees.mean())
```

```
average degree of ego graph: 2.4
```

There's a much easier way to compute this particular quantity, however. If we sum the degrees of all the nodes in a graph, we must get twice the number of edges in the graph. For n nodes and e edges, the average degree is therefore $2m/n$.

```
def average_degree(g):  
    return 2*g.number_of_edges() / g.number_of_nodes()  
  
print("average degree of Twitch network:", average_degree(twitch))
```

```
average degree of Twitch network: 9.914117316867808
```

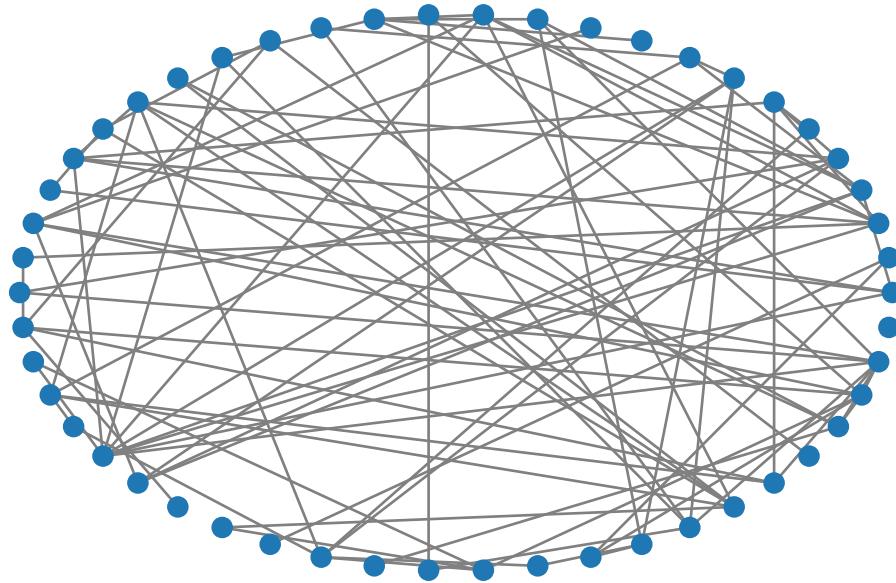
7.1.6 Random graphs

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.

An **Erdős-Rényi graph** (ER graph) includes each individual possible edge with a fixed probability p . That is, if you have a weighted coin that comes up heads ($100p\%$) of the time, then you toss the coin for each possible pair of vertices and include their edge if it is heads.

```
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



Since there are $\binom{n}{2}$ unique pairs among n nodes, the mean number of edges in an ER graph is

$$p \binom{n}{2} = \frac{pn(n-1)}{2}.$$

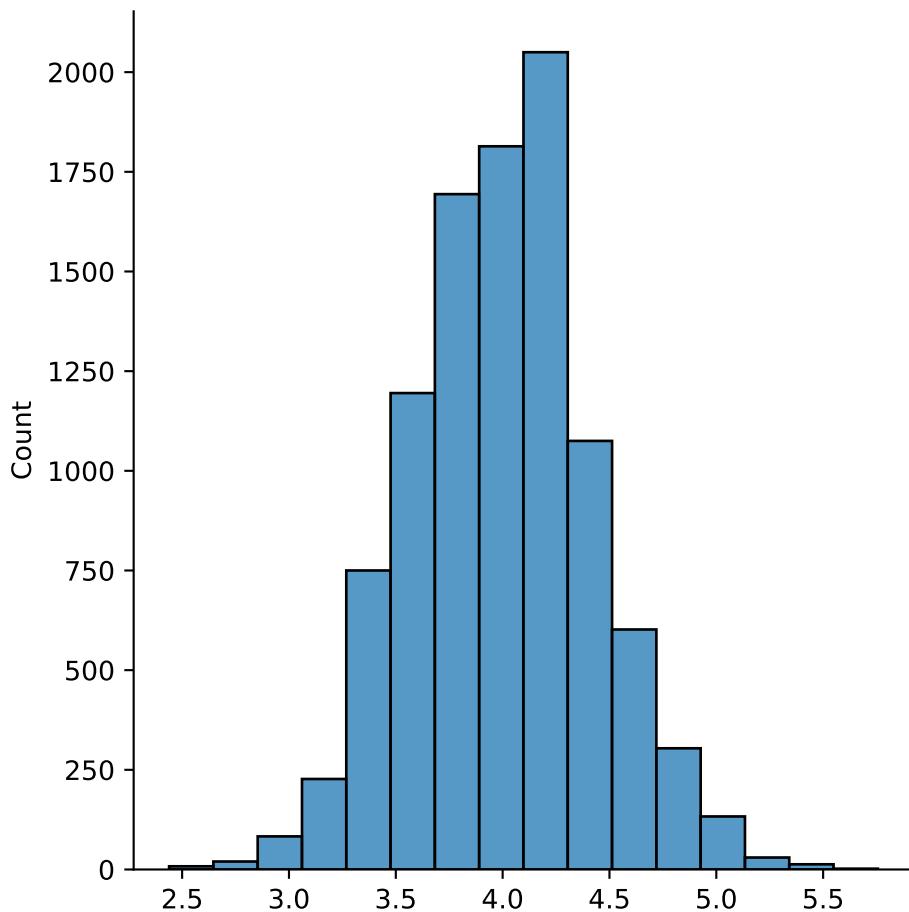
This fact is usually stated in terms of the average node degree, \bar{k} :

$$E[\bar{k}] = \frac{1}{n}pn(n-1) = p(n-1).$$

There are two senses of “average” going on here: in each graph instance, you find the average degree, then you take the average (expectation, $E[\cdot]$) over all random instances. Here is the distribution of \bar{k} over 10000 instances when its expected value is 4.0:

```
n,p = 41,0.1
kbar = []
for iter in range(10000):
    ER = nx.erdos_renyi_graph(n,p,seed=iter+1001)
    kbar.append(average_degree(ER))

sns.displot(x=kbar,bins=16);
```



7.2 Clustering

Note

The term *clustering* has a meaning for network analysis that has virtually nothing to do with *clustering* of numerical data.

In your 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 precisely, but one of the easiest is the **local clustering coefficient**, defined for a node i as

$$C(i) = \frac{2T(i)}{d_i(d_i - 1)}.$$

In this formula, 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 $C(i) = 0$.

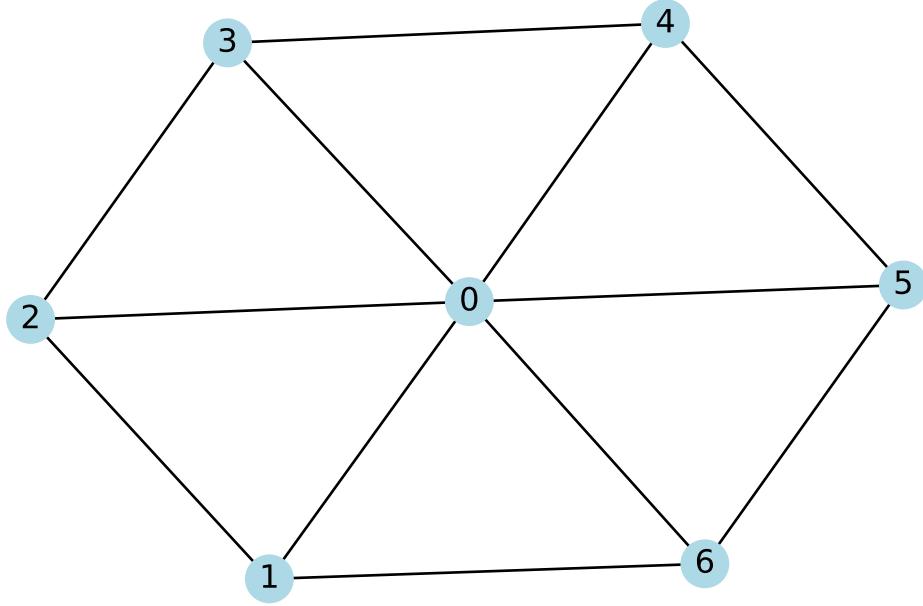
Equivalently, $T(i)$ is the number of triangles in the graph that pass through node i . Because the subgraph of the neighbors has

$$\binom{d_i}{2}$$

possible edges, the value of $C(i)$ is between 0 and 1.

Here is a wheel graph to help us explore a bit:

```
W = nx.wheel_graph(7)
nx.draw(W, with_labels=True, node_color="lightblue")
```



Example 7.1. Let's find the clustering coefficient for each node in the wheel graph drawn above.

Node 0 in this graph is adjacent to 6 other nodes, and there are 6 triangles passing through it. Thus, its clustering coefficient is

$$C(0) = \frac{6}{6 \cdot 5/2} = \frac{2}{5}.$$

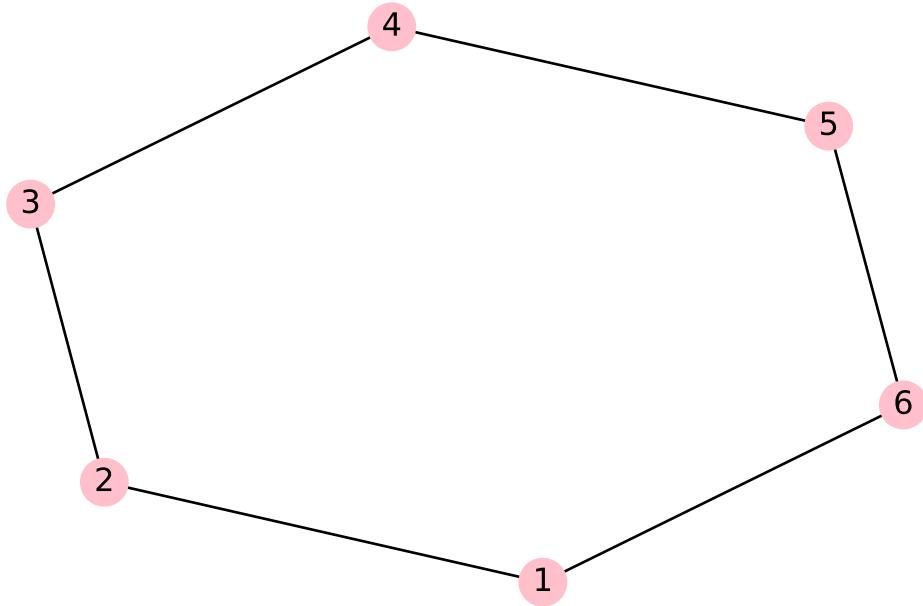
Every other node has 3 friends and 2 triangles, so they each have

$$C(i) = \frac{2}{3 \cdot 2/2} = \frac{2}{3}, \quad i \neq 0.$$

In NetworkX, we can manually count the number of edges among neighbors of node 0 by examining the 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, 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.

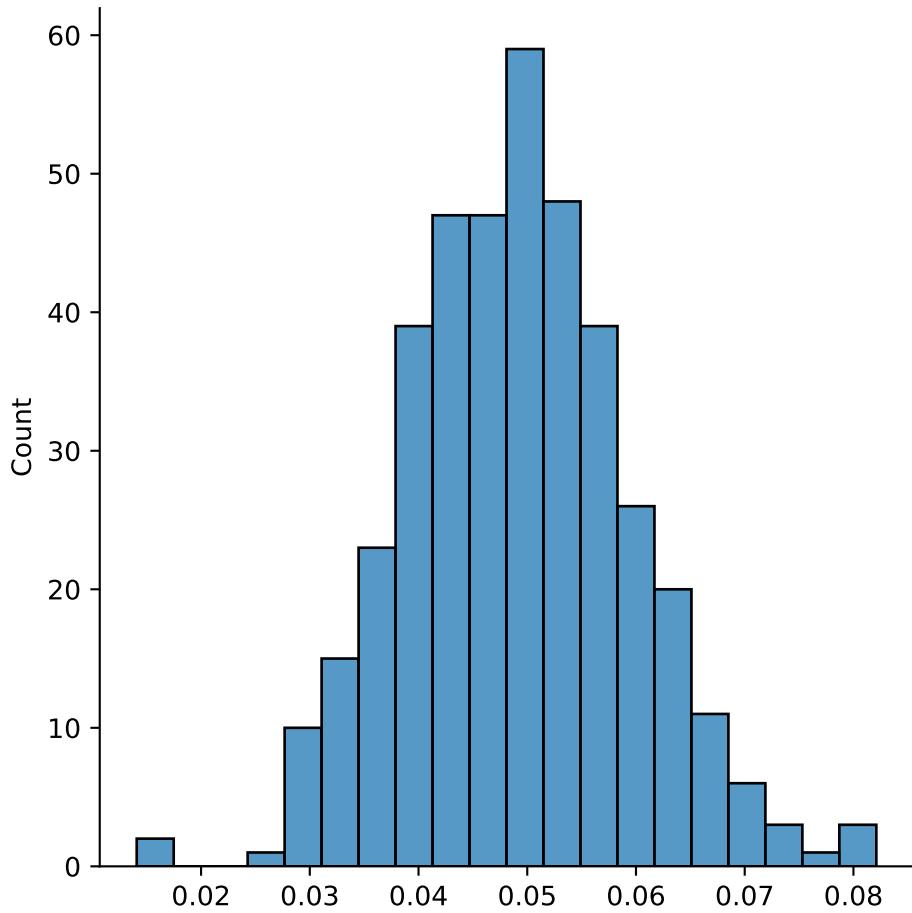
```
print("average clustering =", nx.average_clustering(W))
```

```
average clustering = 0.6285714285714284
```

Example 7.2. 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, which is not a coincidence.

Theorem 7.1. *The expected value of the average clustering 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, that is also the expected fraction of edges that appear within the neighborhood subgraph of any node.

Example 7.3. Let's examine clustering within the Twitch network.

```

twitch = nx.read_edgelist("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}")

```

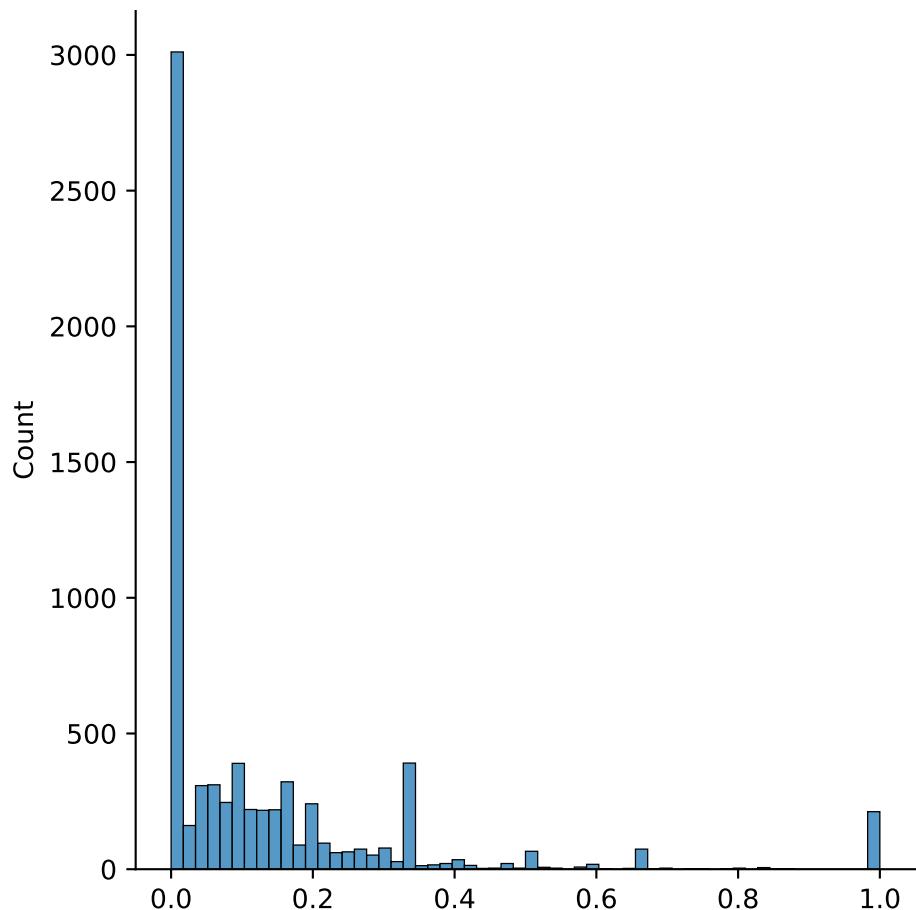
7126 nodes and 35324 edges
 average degree is 9.914

Computing the distances between all pairs of nodes in this graph would take a rather long time, so we will estimate the average distance by sampling.

```

cluster = pd.Series(nx.clustering(twitch), index=twitch.nodes)
sns.displot(data=cluster);

```



The average clustering coefficient is

```
print( "average Twitch clustering:", cluster.mean() )
```

```
average Twitch clustering: 0.1309282190147198
```

How does this value compare to an ER graph? If we set the number of nodes and average degree to be the same, then the expected average clustering for ER graphs is $p = \bar{k}/(n-1)$:

```
print( "average equivalent ER clustering:", kbar/(n-1) )
```

```
average equivalent ER clustering: 0.0013914550620165345
```

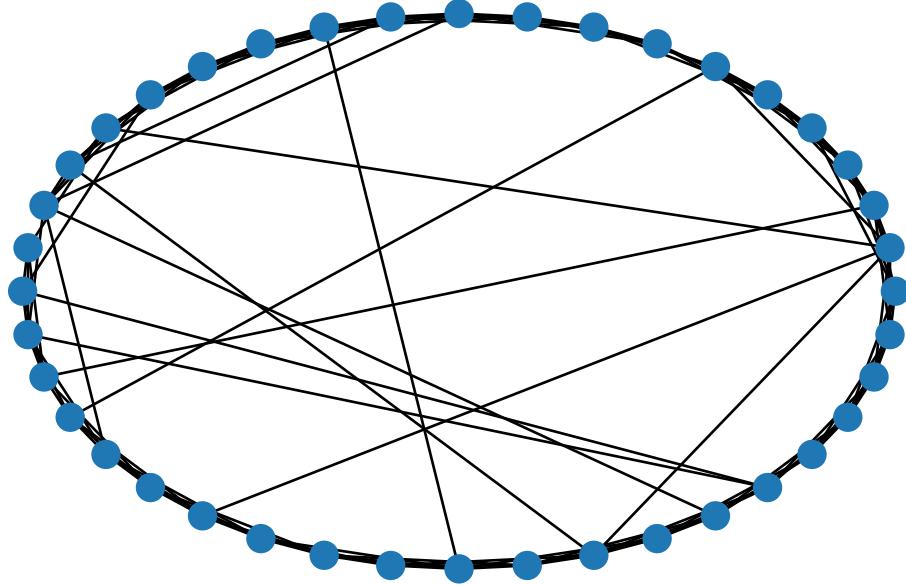
This is too small by a factor of 100! Clearly, the Twitch graph is not equivalent to a random graph in the sense of ER. From a sociological perspective, of course, this is a “no duh” conclusion.

7.2.1 Watts–Strogatz graphs

A **Watts–Strogatz graph** (WS graph) tries to model the small-world phenomenon. A WS graph has three parameters: n , an even integer k , and a probability q .

Imagine n nodes arranged in a circle. Connect each node with an edge to each of its $k/2$ left neighbors and $k/2$ right neighbors. Now we “rewire” some of the edges by visiting each node i in turn. For each edge from i to a neighbor, with probability q replace it with an edge between i and a node chosen at random from all the nodes i is not currently connected to. The idea is to start with tight-knit, overlapping communities, and randomly toss in some far-flung links.

```
WS = nx.watts_strogatz_graph(40, 6, 0.15, seed=1)
nx.draw_circular(WS, node_size=100)
```



By the nature of the construction, the initial state of the network (before the rewiring phase) is highly clustered. Thus, if q is close to zero, the final graph will retain much of this initial clustering.

```

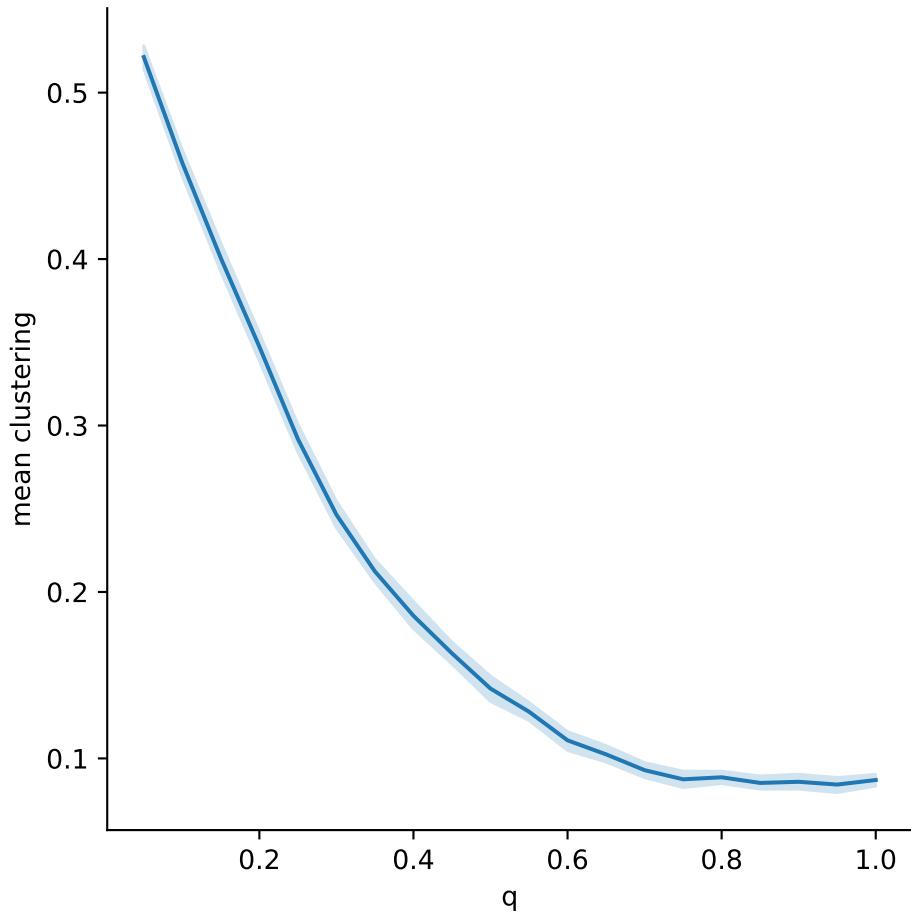
n, k = 60, 6
results = []
seed = 0
for q in np.arange(0.05, 1.05, 0.05):
    for iter in range(50):
        WS = nx.watts_strogatz_graph(n, k, q, seed=seed)
        results.append( (q, nx.average_clustering(WS)) )
        seed += 1

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 is nearly 10, which is the value we will use in the WS construction. To save computation time, we will use just one WS realization at each value of q .

```
seed = 99999
n, k = twitch.number_of_nodes(), 10
for q in np.arange(0.15, 0.61, 0.05):
    WS = nx.watts_strogatz_graph(n, k, q, seed=seed)
    print(f"q = {q:.2f}, avg WS clustering = {nx.average_clustering(WS):.4f}")
    seed += 1
```

q = 0.15, avg WS clustering = 0.4143

q = 0.20, avg WS clustering = 0.3418
q = 0.25, avg WS clustering = 0.2870

```

q = 0.30, avg WS clustering = 0.2326
q = 0.35, avg WS clustering = 0.1809

q = 0.40, avg WS clustering = 0.1470
q = 0.45, avg WS clustering = 0.1112
q = 0.50, avg WS clustering = 0.0851

q = 0.55, avg WS clustering = 0.0627
q = 0.60, avg WS clustering = 0.0424

```

The mean clustering resembles the value of 0.131 for the Twitch network at around $q = 0.42$, which we verify using more realizations:

```

seed = 999
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( "avg WS clustering at q = 0.42:", np.mean(cbar) )

```

```
avg WS clustering at q = 0.42: 0.13177740621327544
```

The WS construction gives a plausible way to reconstruct the clustering observed in the Twitch network. However, there are other graph properties left to examine.

7.3 Distance

The *small-world phenomenon* is, broadly speaking, the observation that any two people in a 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.

The **distance** between two nodes in a connected graph is the number of edges in the shortest path between them. 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.

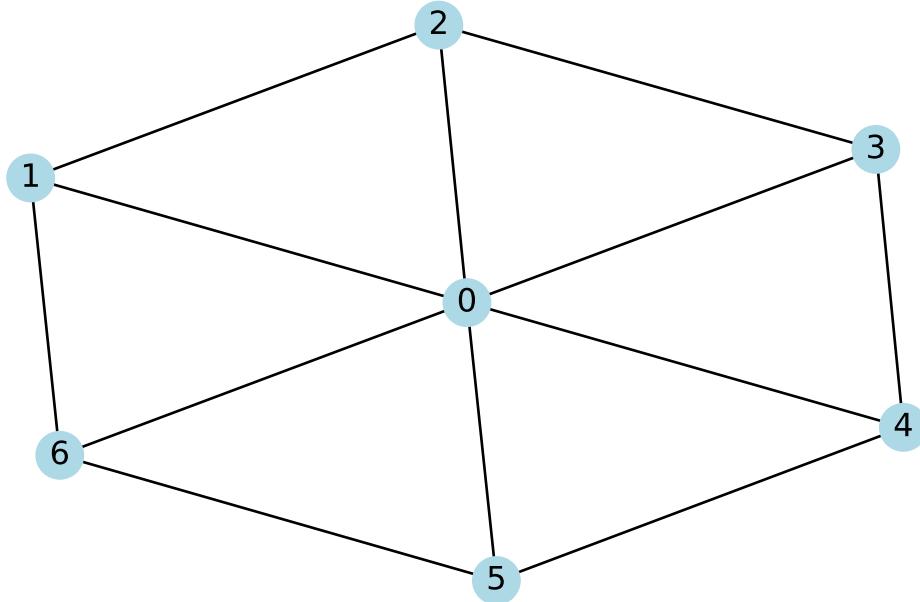
```
K5 = nx.complete_graph(5)
dist = pd.Series(nx.shortest_path_length(K5,0), index=K5.nodes)
print("Distance from node 0:", dist)
```

```
Distance from node 0: 0      0
1      1
2      1
3      1
4      1
dtype: int64
```

The maximum distance over all pairs of nodes in a graph is called its **diameter**. Since this value depends on an extreme outlier in the distribution of distances, we often prefer to use the **average distance** as a measure of how difficult it is to connect two randomly selected nodes.

For example, here is a 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 (if the first hop is to node 0), so the diameter of this graph is 2. The average distance is somewhat smaller. This graph is so small that we can easily find the entire matrix of pairwise distances. The matrix is symmetric, so it's only necessary to compute its upper triangle.

```

nodes = list(W.nodes)
n = len(nodes)
D = np.zeros( (n,n), dtype=int )
for i in range(n):
    for j in range(i+1,n):
        D[i,j] = nx.shortest_path_length(W, nodes[i], nodes[j])

print(D)

```

```

[[0 1 1 1 1 1]
 [0 0 1 2 2 2]
 [0 0 0 1 2 2]
 [0 0 0 0 1 2]
 [0 0 0 0 0 1]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]

```

To get the average distance, we can sum over all the entries and divide by $\binom{n}{2}$:

```
print( "average distance:", 2*D.sum() / (n*(n-1)) )
```

```
average distance: 1.4285714285714286
```

There is a convenience function for computing this average. (It becomes slow as n grows, though.)

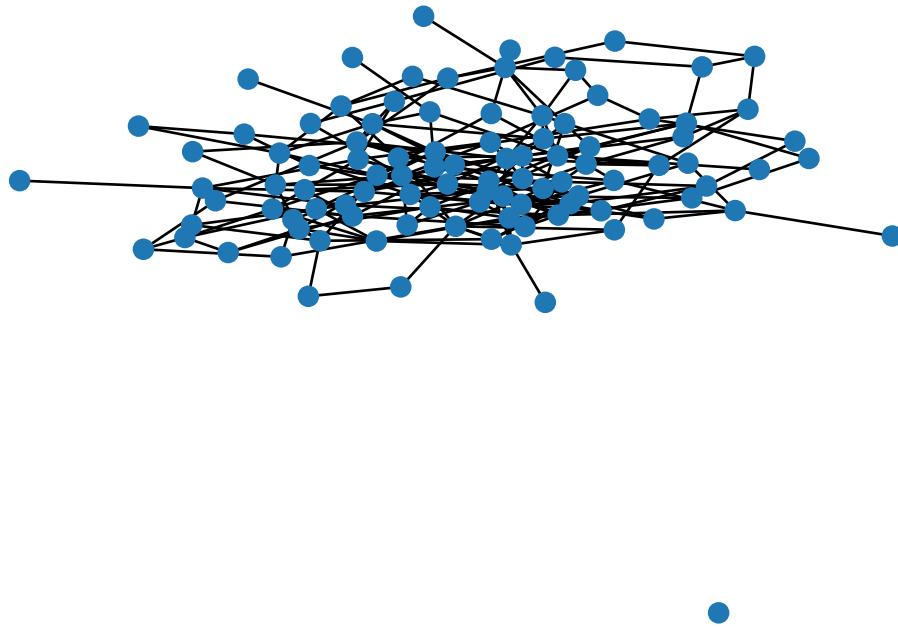
```
print( "average distance:", nx.average_shortest_path_length(W) )
```

```
average distance: 1.4285714285714286
```

7.3.1 ER graphs

If we want to compute distances within ER random graphs, we quickly run into a problem: an ER graph may not have a path between every pair of nodes:

```
n, p = 101, 1/25
ER = nx.erdos_renyi_graph(n, p, seed=0)
nx.draw(ER, node_size=50)
```



We say that such a graph is not **connected**. When no path exists between two nodes, the distance between them is either undefined or infinite. NetworkX will give an error if we try to compute the average distance in a disconnected graph:

```
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
```

Now the average path length is a valid computation.

```
nx.average_shortest_path_length(ER_sub)
```

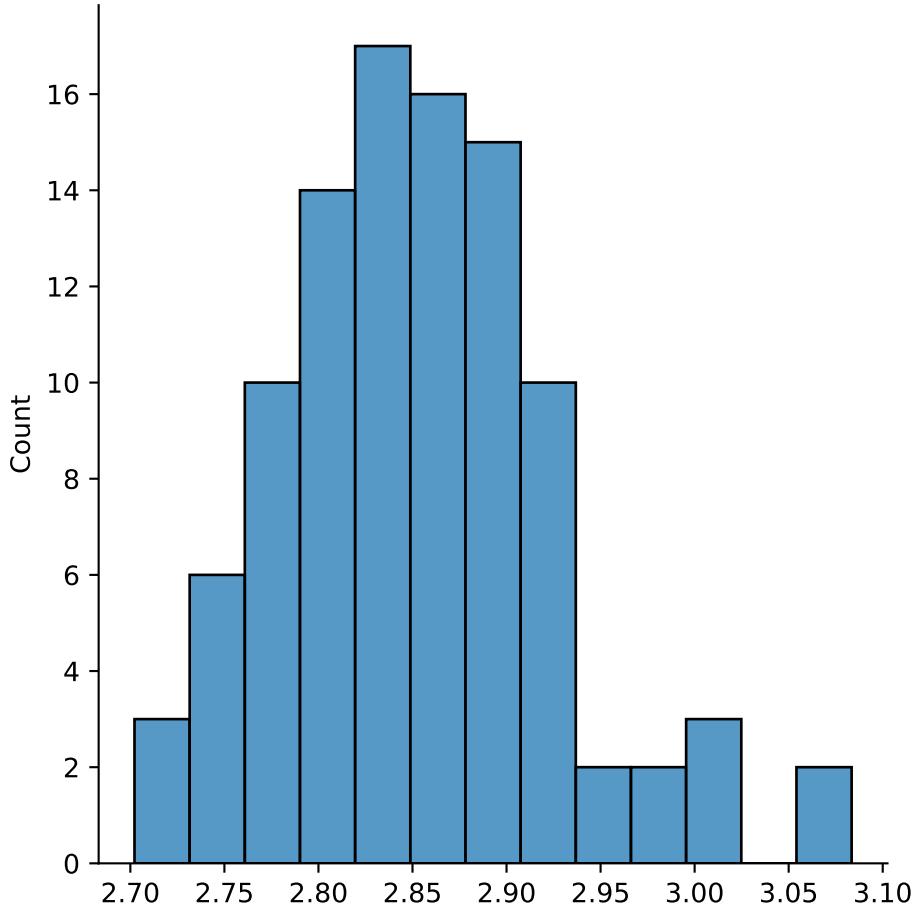
```
3.3082828282828283
```

Let's use this method to 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). In fact, theory states that as $n \rightarrow \infty$, the mean distance in ER graphs is expected to be approximately

$$\frac{\ln(n)}{\ln(\bar{k})}. \quad (7.1)$$

For $n = 121$ and $\bar{k} = 6$ as in the experiment above, this value is about 2.68.

7.3.2 Watts–Strogatz 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:

```

G = nx.watts_strogatz_graph(400, 6, 0) # q=0 ==> initial ring lattice
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

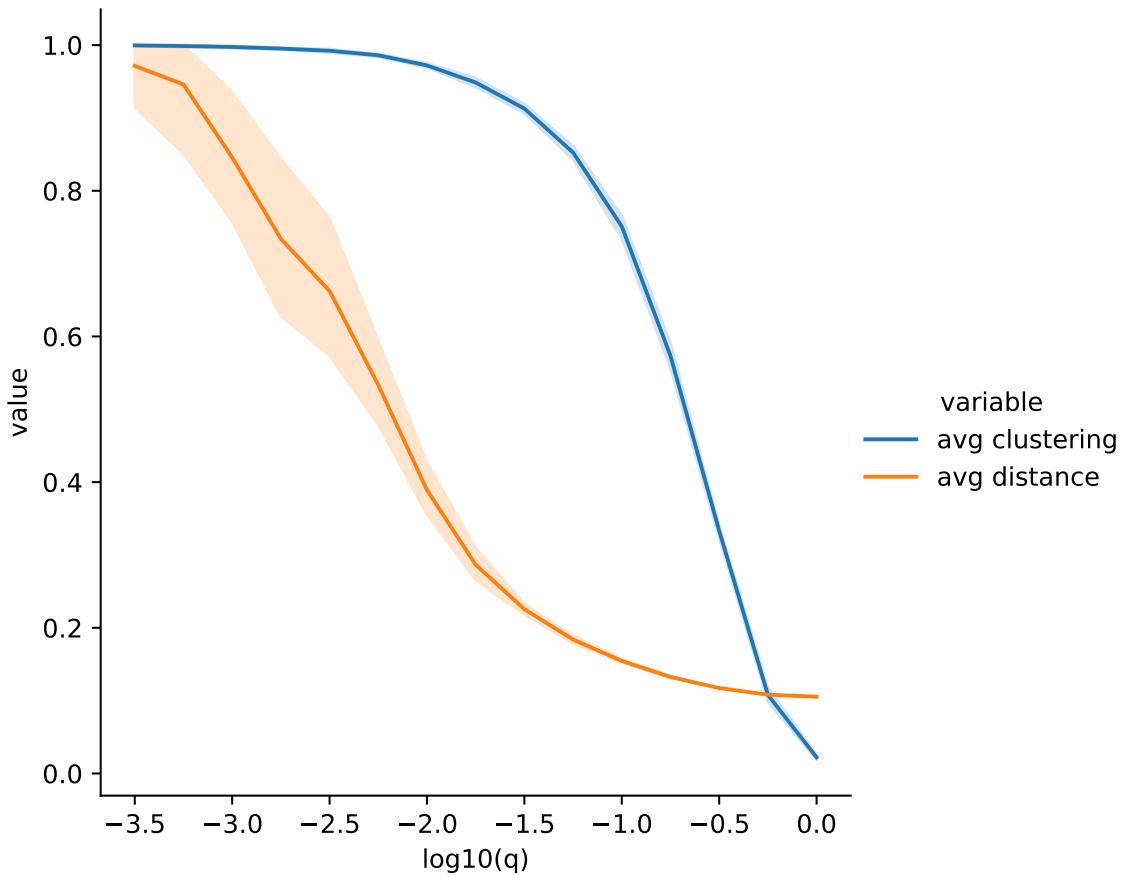
At the other extreme of $p = 1$, we get an ER random graph, which (at equivalent parameters) has small clustering and small average distance. The most interesting aspect of WS graphs is the transition between these extremes as p varies.

```

cbar, dbar, logq = [],[],[]
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=975+iter)
        cbar.append( nx.average_clustering(G) / C0 )
        dbar.append( nx.average_shortest_path_length(G) / L0 )
        logq.append(lq)

results = pd.DataFrame( {"log10(q)":logq, "avg clustering":cbar, "avg distance":dbar} )
sns.relplot(data=pd.melt(results, id_vars="log10(q)"),
             x="log10(q)", y="value",
             hue="variable", kind="line"
            );

```



The horizontal axis above is $\log_{10}(q)$, and the vertical axis shows the average clustering and shortest path length normalized by their values at $q = 0$. 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 a large clustering coefficient and small average distance.

7.3.3 Twitch network

Let's consider distances within the Twitch network.

```
twitch = nx.read_edgelist("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
```

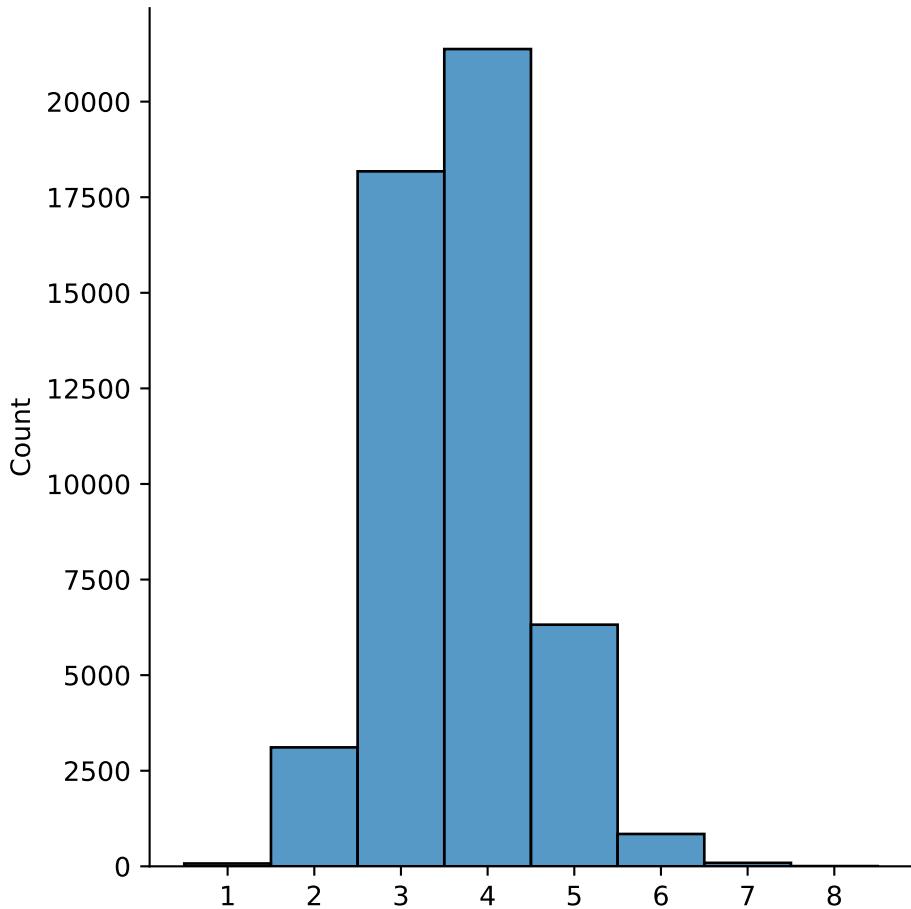
Computing the distances between all pairs of nodes in this graph would take a rather long time, so we will sample some pairs randomly.

```
rng = default_rng(1)

# 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 = 3.67376
```



Let's compare these results to ER graphs with the same size and average degree, i.e., with $p = \bar{k}/(n - 1)$. The theoretical estimate from above gives

```
print("Comparable ER graphs expected mean distance:", np.log(n) / np.log(kbar))
```

```
Comparable ER graphs expected mean distance: 3.8673326382368893
```

The Twitch network has a slightly smaller value than this, but the numbers are comparable. However, remember that the ER graphs have a negligible clustering coefficient.

Next we explore Watts–Strogatz graphs with the same n as the Twitch network and $k = 10$ to get a similar average degree.

```
results = []
seed = 44044
```

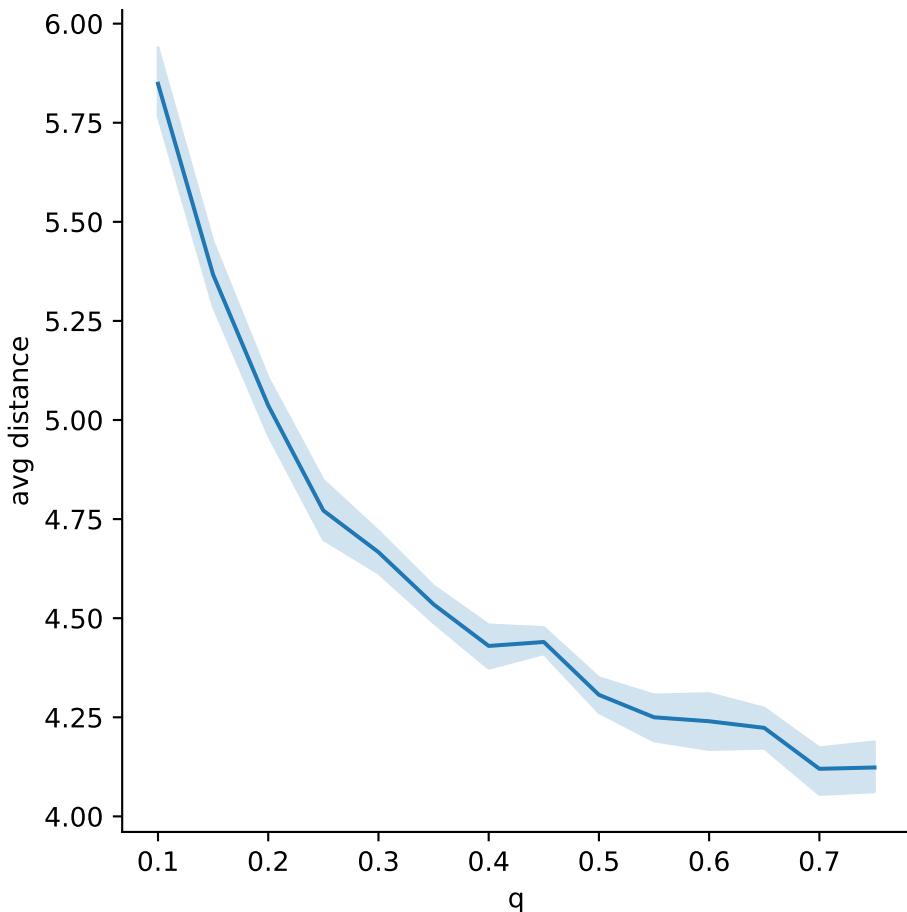
```

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", "avg distance"] )
print("Pairwise distances in WS graphs:")
sns.relplot(data=results, x="q", y="avg distance", kind="line");

```

Pairwise distances in WS graphs:



The decrease with q is less pronounced than it was for the smaller WS graphs above. In the

previous section, we found that $q = 0.42$ reproduces the same average clustering as in the Twitch network. That corresponds to a mean distance of about 4.5, which is a bit above the observed Twitch mean distance of 3.87, but not dramatically so. Thus, the Watts-Strogatz model could still be considered a plausible one for the Twitch network. In the next section, though, we will see that it misses badly in at least one important aspect.

7.4 Degree distributions

As we 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.

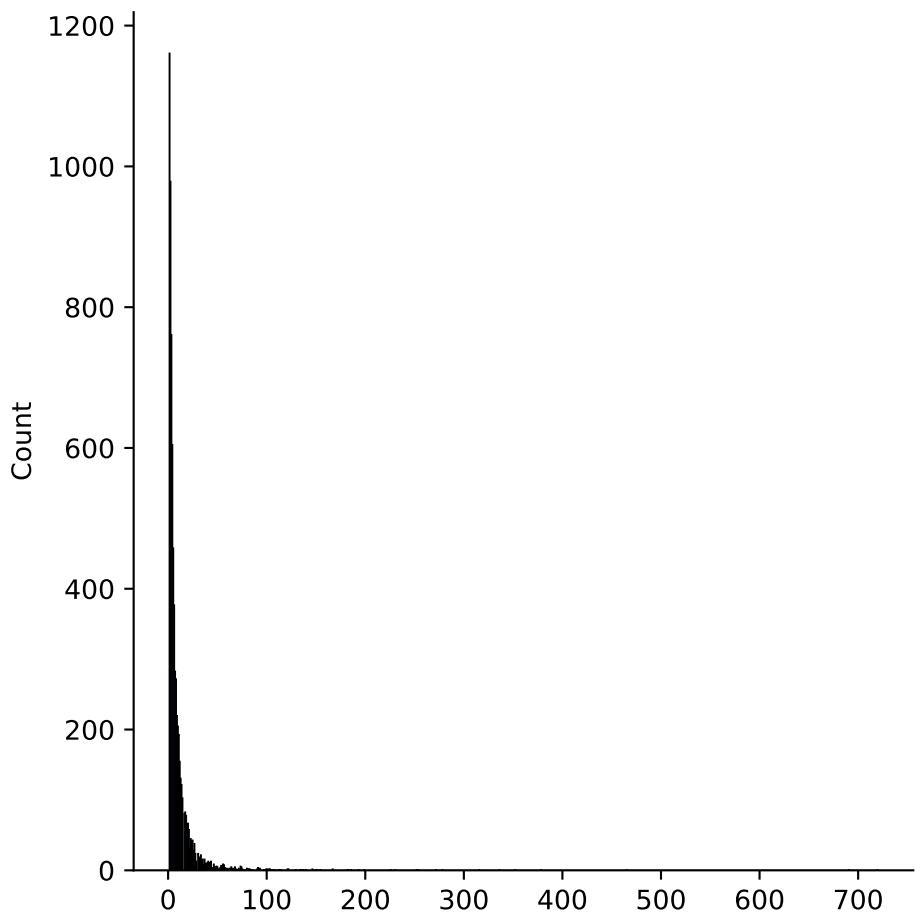
```
twitch = nx.read_edgelist("musae_edges.csv", delimiter=',', nodetype=int)
twitch_degrees = pd.Series( dict(twitch.degree), index=twitch.nodes )
twitch_degrees.describe()
```

	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 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.

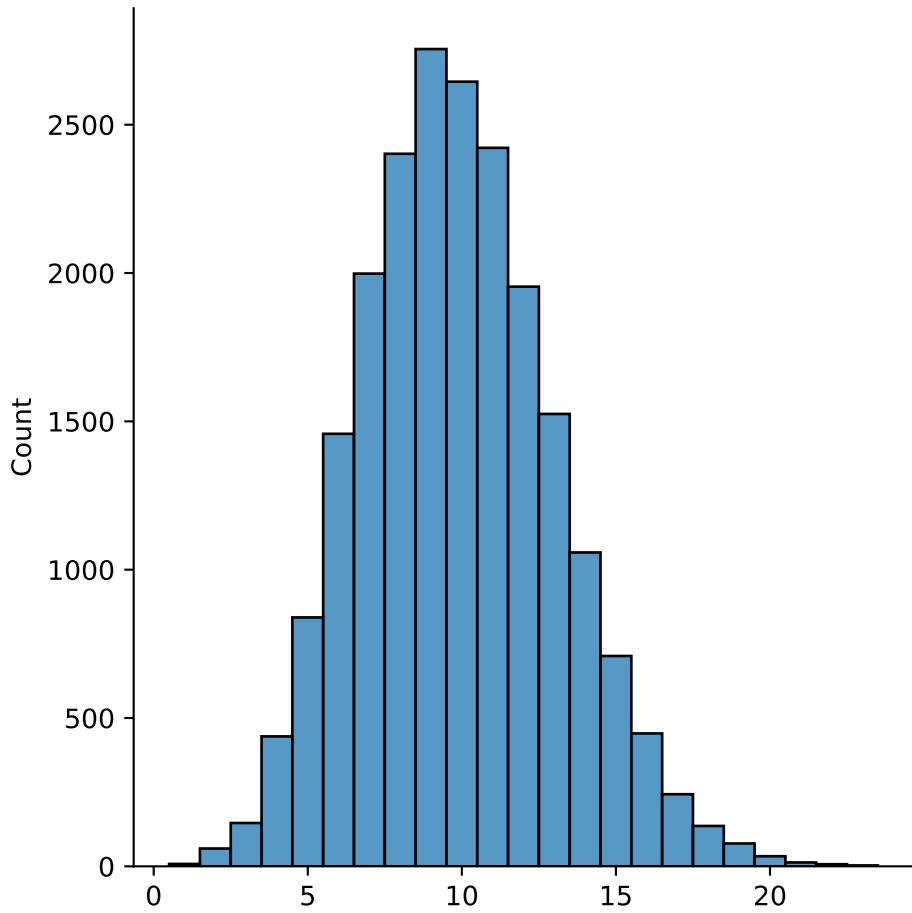
```

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:

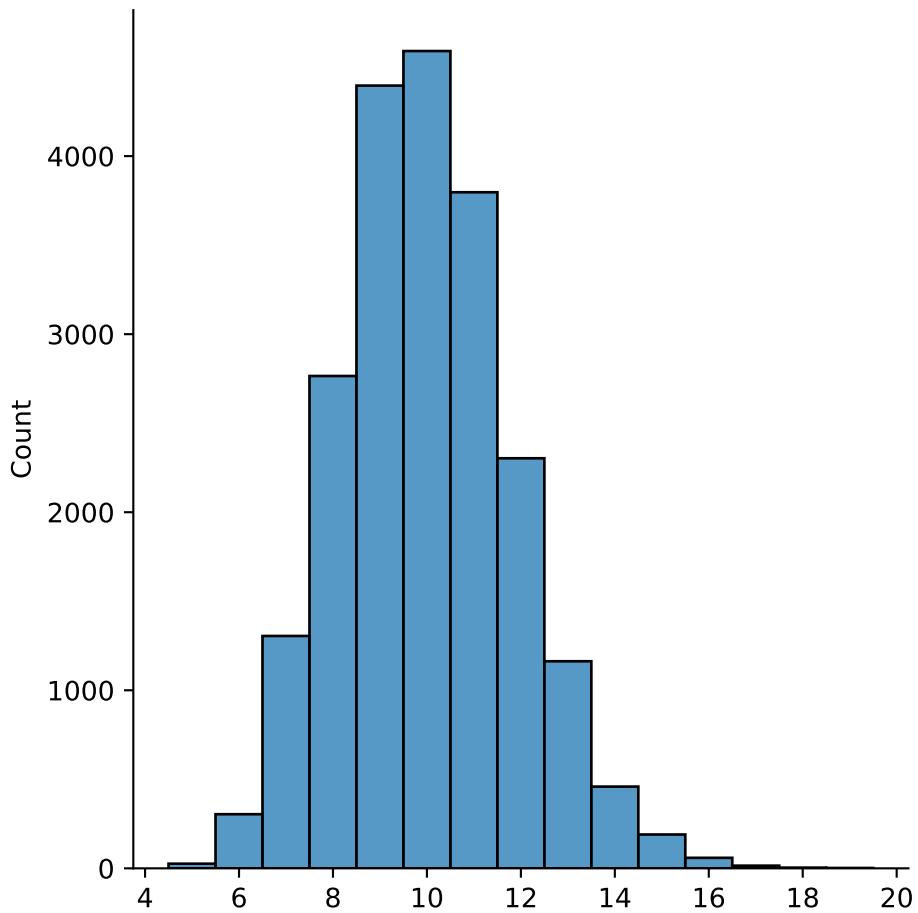
```

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);

```

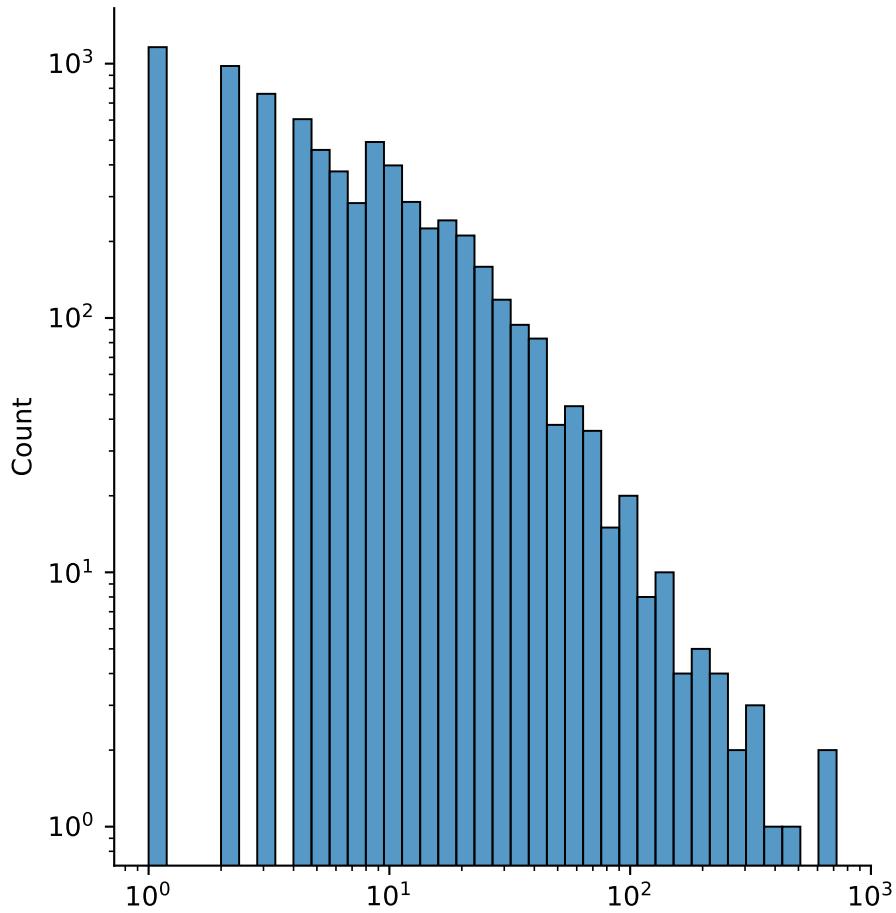
WS graphs degree distribution:



7.4.1 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**. Many social networks seem to follow a power-law distribution of node degrees, to some extent. (The precise extent is a subject of hot debate.)

Note that the decay of x^{-a} to zero as $x \rightarrow \infty$ is much slower than, say, the normal distribution's $e^{-x^2/2}$, or even just an exponential e^{-cx} . This last comparison is how a *heavy-tailed distribution* is usually defined.

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. It estimates that the degree counts vary as $Bx^{-2.1}$ over a wide range of degrees.

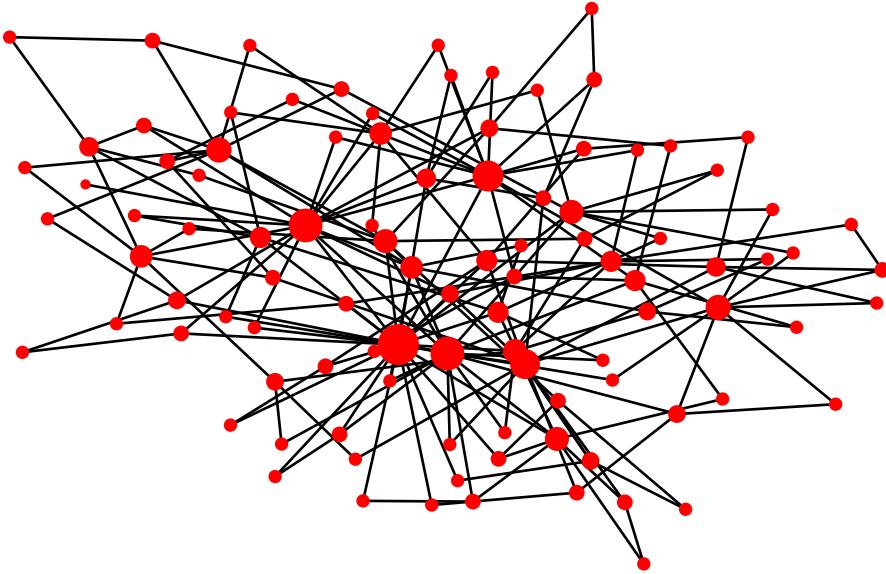
7.4.2 Barabási–Albert graphs

A random **Barabási–Albert** graph (BA graph) is constructed by starting with a small seed network and connecting one node at a time with m new edges to it. Edges are added randomly, but higher probability is given to connect to nodes that already have higher degree (i.e., are more “popular”), a concept known as *preferential attachment*. Because of this rule, there is a natural tendency to develop a few hubs of high degree.

```

BA = nx.barabasi_albert_graph(100, 2, seed=0)
BA_degrees = pd.Series( dict(BA.degree), index=BA.nodes )
nx.draw(BA, node_size=8*BA_degrees, node_color="red")

```



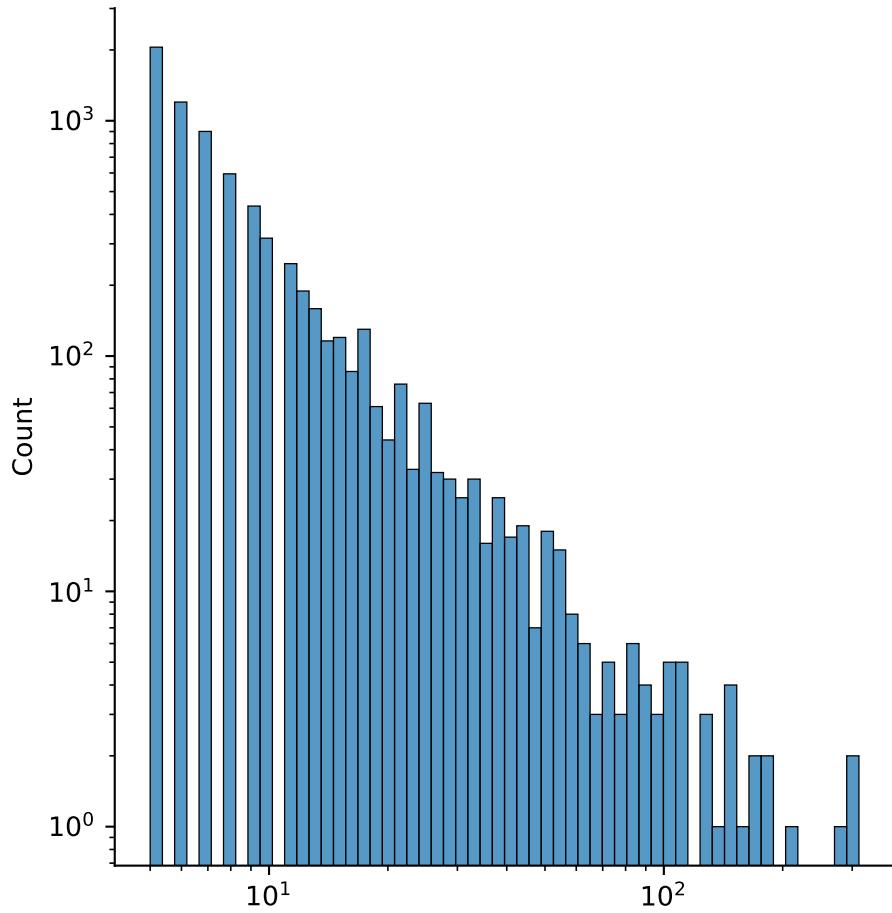
When we match these graphs to the size and average degree of the Twitch network, a power-law distribution emerges. Since we add m edges (almost) n times, the expected average degree is $2mn/n = 2m$. Therefore, in the BA construction we want to choose

$$m \approx \frac{\bar{k}}{2}.$$

```

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")

```



Theory predicts that the exponent of the power-law distribution in a BA graph is -3 .

```
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
```

Let's check distances and clustering, too. As a reminder, the mean distance in the Twitch network is approximately:

```

from numpy.random import default_rng
rng = default_rng(1)

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)

print("Mean distance in Twitch graph:",
      sum(pairdist(twitch) for _ in range(4000)) / 4000 )

```

Mean distance in Twitch graph: 3.657

Now we repeat that for some BA graphs.

```

dbar = []
seed = 911
for iter in range(10):
    BA = nx.barabasi_albert_graph(n, m, seed=seed)
    d = sum(pairdist(BA) for _ in range(200)) / 200
    dbar.append(d)
    seed += 1

print( "Mean distance in BA graphs:", np.mean(dbar) )

```

Mean distance in BA graphs: 3.5555

Not bad! Now, let's check the clustering. For Twitch, 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 BA model is our closest approach so far, but it fails to produce the close-knit neighbor subgraphs that we find in the Twitch network and the WS model.

7.5 Centrality

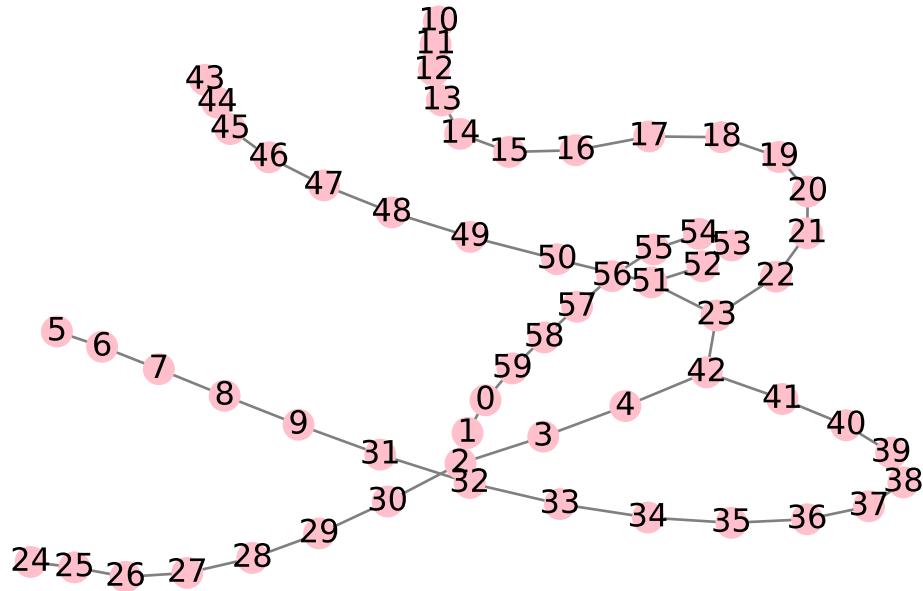
In some applications we might want to know which nodes of a network are the most important. For instance, we might want to find influential members of a social network, or nodes that are critical for efficient connections within the network. These traits go under the general name of **centrality**.

An easy candidate for measuring the centrality of a node is its degree. Usually this is normalized by the number of nodes in the graph and called **degree centrality**. While it can yield useful information in some networks, it is not always a reliable measuring stick. For example, consider the following Watts–Strogatz graph:

```

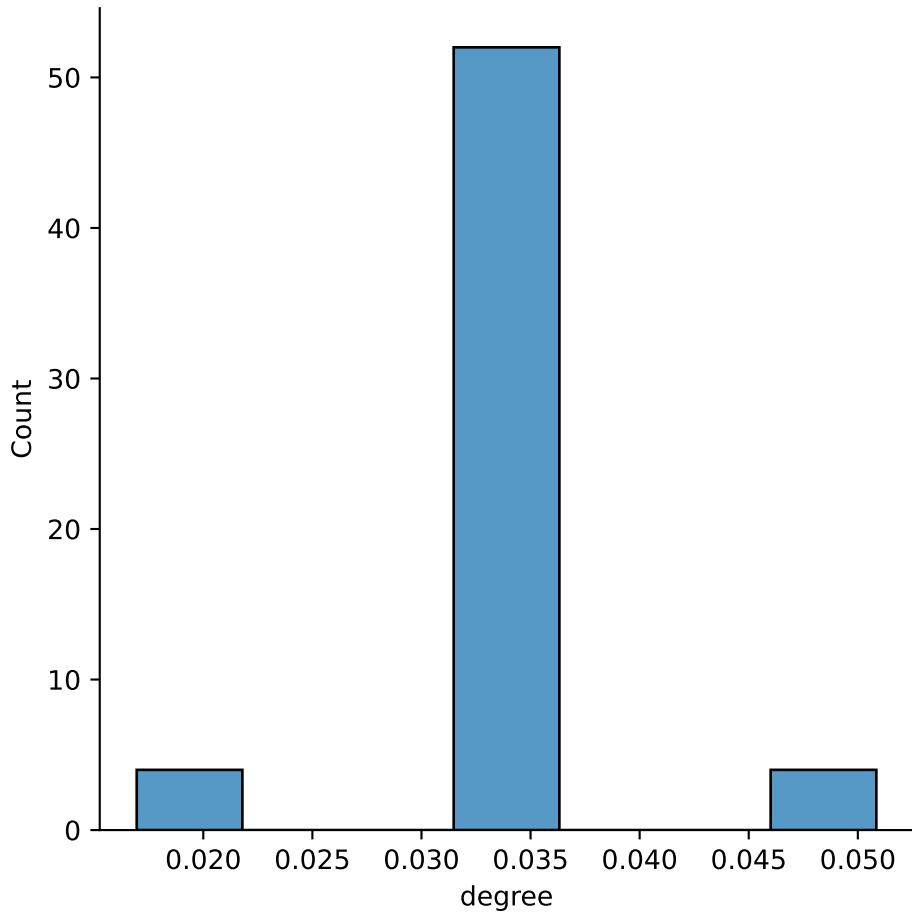
G = nx.watts_strogatz_graph(60, 2, .1, seed=6)
pos = nx.spring_layout(G, seed=1)
style = dict(pos=pos, edge_color="gray", node_color="pink", with_labels=True)
nx.draw(G, **style, node_size=120)

```



There is little variation in the degrees of the nodes. In fact, there are only 3 unique values of the degree centrality:

```
centrality = pd.DataFrame( {"degree":nx.degree_centrality(G)}, index=G.nodes )
sns.displot(data=centrality, x="degree");
```



From the drawing of the graph, however, it's clear that (for instance) nodes 3 and 6 do not have comparable roles, despite the fact that both have degree equal to 2.

7.5.1 Betweenness centrality

A different way to measure centrality is to use shortest paths between nodes. Let $\sigma(i, j)$ denote the number of shortest paths between nodes i and j . This means that we count the number of unique ways to get between these nodes using the minimum possible number of edges. Let $\sigma(i, j|k)$ be the number of such paths that pass through node k . Then, for a graph on n nodes, 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 $0 \leq c_B \leq 1$ for any node. The definition requires an expensive computation if the number of nodes is more than a few hundred, so the σ values are often estimated by sampling.

Example 7.4. We will find the betweenness centrality of the following *barbell graph*:

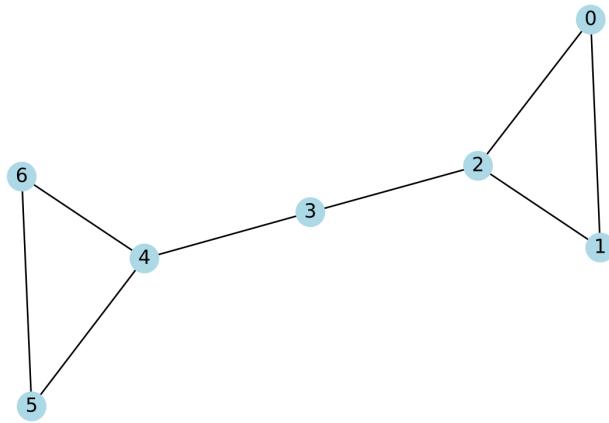


Figure 7.1: Barbell graph

Let's begin with node 3, in the middle. Any path, and therefore any shortest path, between nodes 0, 1, or 2 and nodes 4, 5, or 6 must pass through node 3, so these pairings each contribute 1 to the sum. The shortest paths for pairs of nodes within the end triangles clearly do not pass through node 3. Hence

$$c_B(3) = \frac{1}{15} \cdot (3 \cdot 3) = \frac{3}{5}.$$

Next, consider node 2. The shortest paths through this node are the ones that pair nodes 0 or 1 with nodes 3, 4, 5, or 6, so

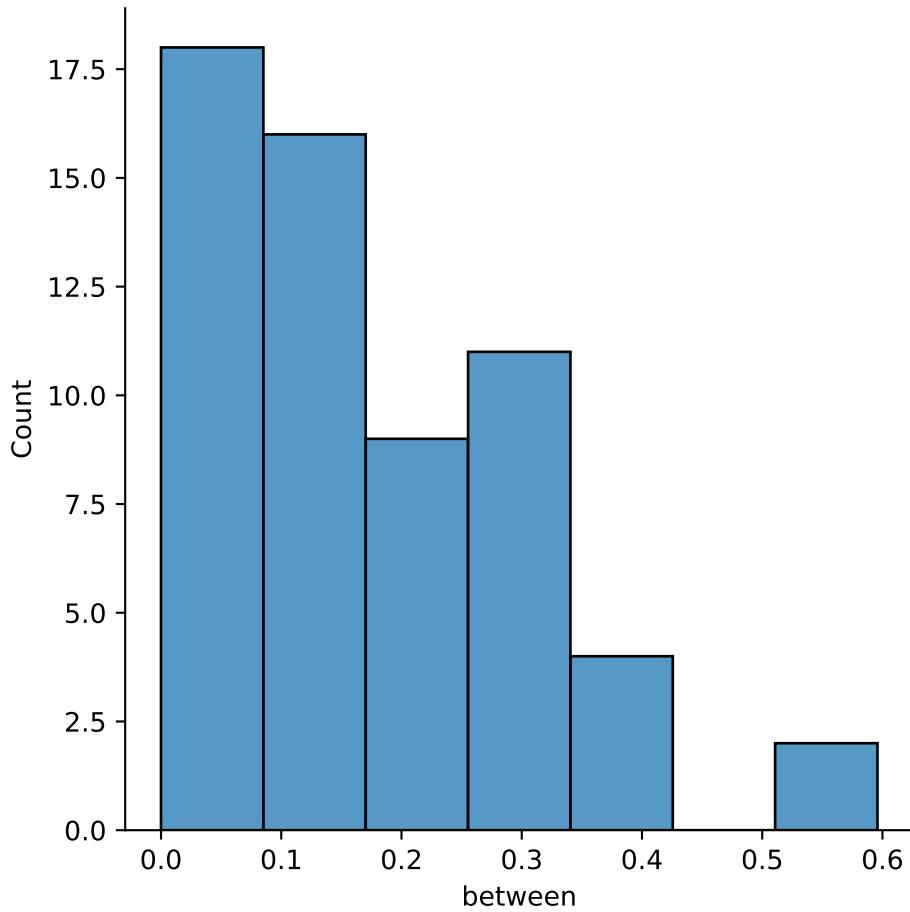
$$c_B(2) = \frac{1}{15} \cdot (2 \cdot 4) = \frac{8}{15}.$$

By symmetry, we get the same value for node 4.

All the other nodes play no role in any 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.

```
centrality["between"] = pd.Series(nx.betweenness_centrality(G), index=G.nodes)
sns.displot(data=centrality, x="between");
```



The distribution above shows that few nodes have a relatively high betweenness score in our graph.

7.5.2 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 λ . This is an **eigenvalue equation**, one of the fundamental problems in linear algebra.

:label: example-centrality-eigenvector 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 should hope that all three vertices are ranked equally. In fact, if we define $\mathbf{x} = \frac{1}{3}[1, 1, 1]$, then

$$\mathbf{Ax} = \left[\frac{2}{3}, \frac{2}{3}, \frac{2}{3} \right] = 2\mathbf{x},$$

so that $\lambda = 2$ is an eigenvalue to go with eigenvector \mathbf{x} . Note that any (nonzero) multiple of \mathbf{x} would work just as well:

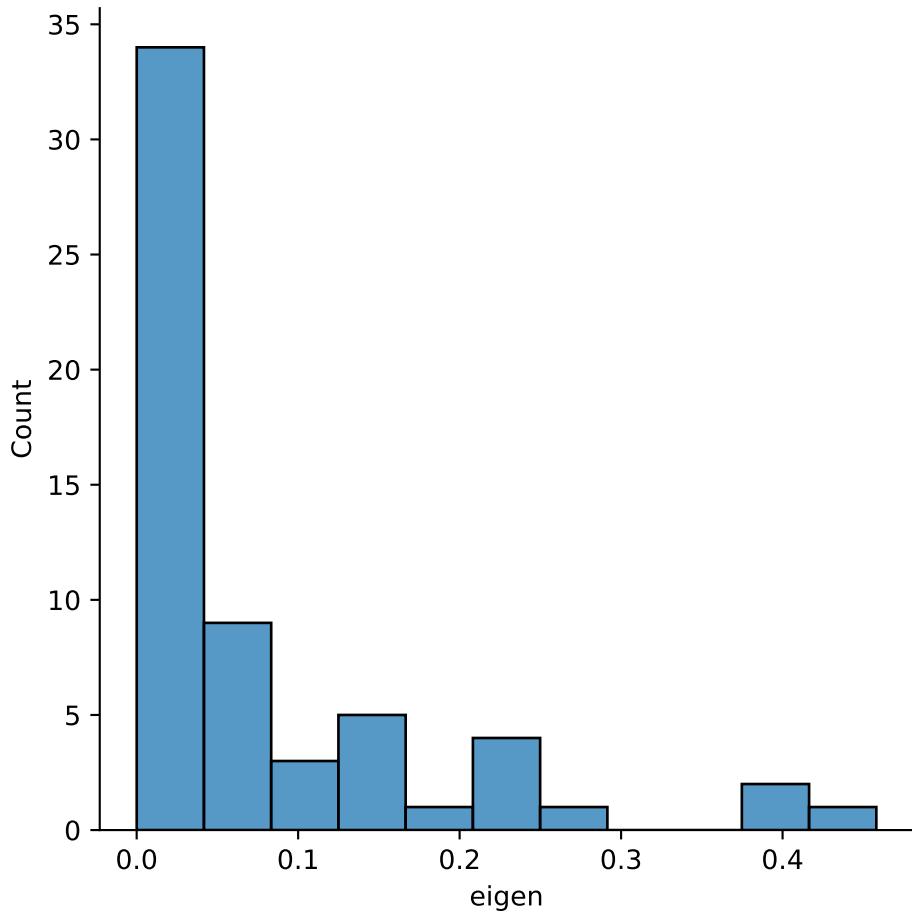
$$\mathbf{A}(c\mathbf{x}) = [\frac{2}{3}c, \frac{2}{3}c, \frac{2}{3}c] = 2(c\mathbf{x}),$$

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. This is called **eigenvector centrality**.

NetworkX has two functions for computing eigenvector centrality. Here we use the one that calls on numpy to solve the eigenvalue problem. As with betweenness centrality, the return value is a dictionary with nodes as the keys.

```
centrality["eigen"] = pd.Series(nx.eigenvector_centrality_numpy(G), index=G.nodes)
sns.displot(data=centrality,x="eigen");
```



You can see above that eigenvector centrality distinguishes a small number of nodes in our example.

7.5.3 Comparison

We can verify using correlation coefficients 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

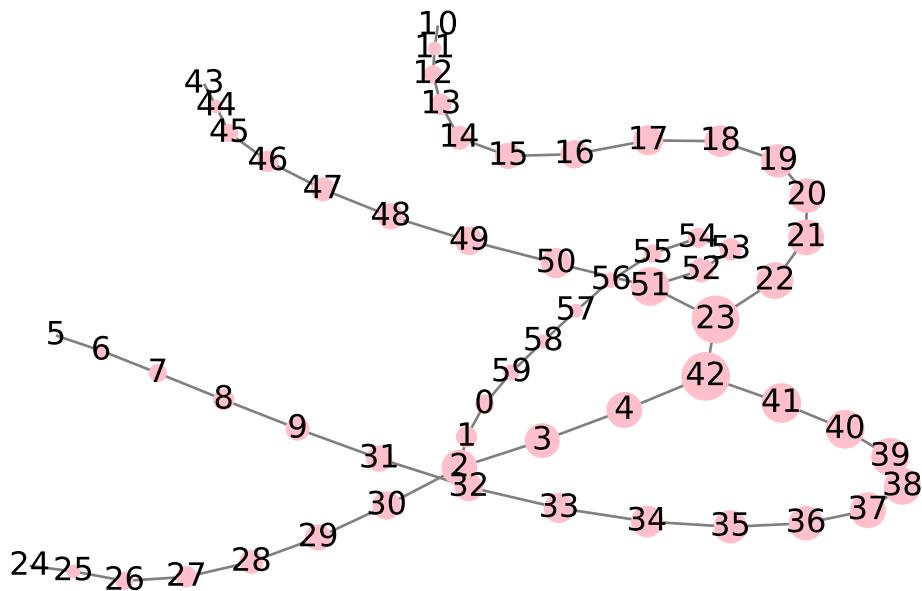
Here is how betweenness ranks the centrality of the nodes:

```
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, and a drawing of the graph supports the case that they are central:

```
nx.draw(G, node_size=500*centrality["between"], **style)
```



A weakness, though, is that there are many secondary nodes whose values taper off only slowly as we enter the remote branches.

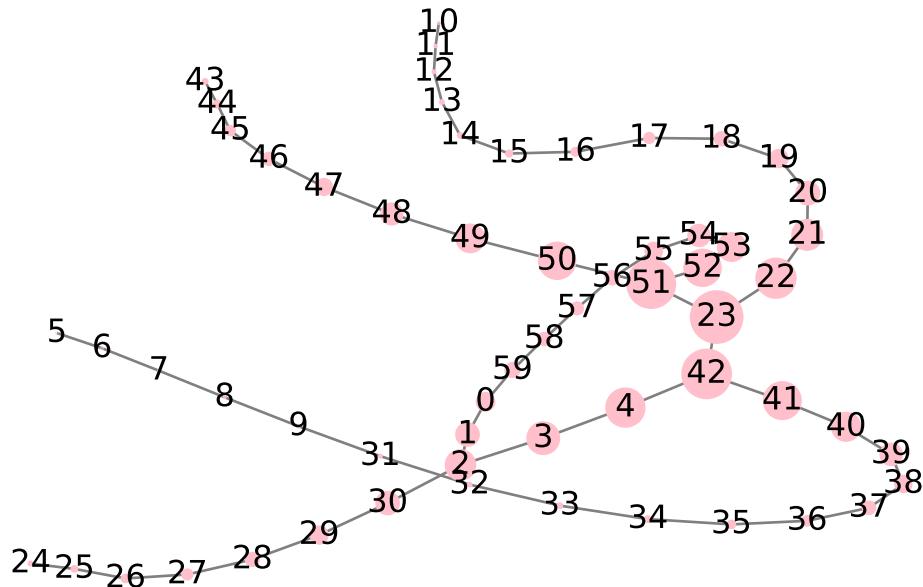
Here is a ranking according to eigenvector centrality:

```
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.

```
nx.draw(G, node_size=800*centrality["eigen"], **style)
```

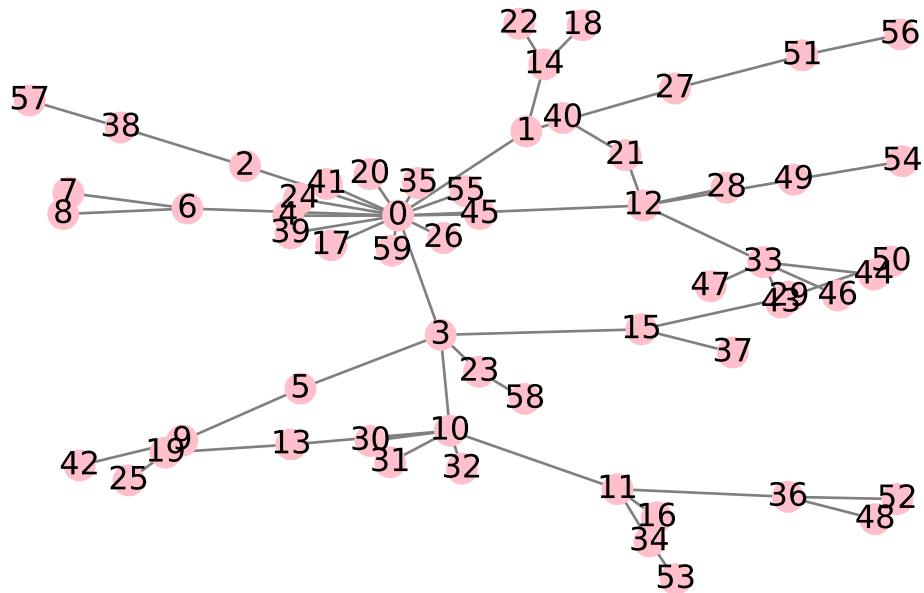


Eigenvector centrality identifies a more compact and distinct center. Of course, these observations are all made for a single network, so be careful not to over-generalize!

7.5.4 Power-law example

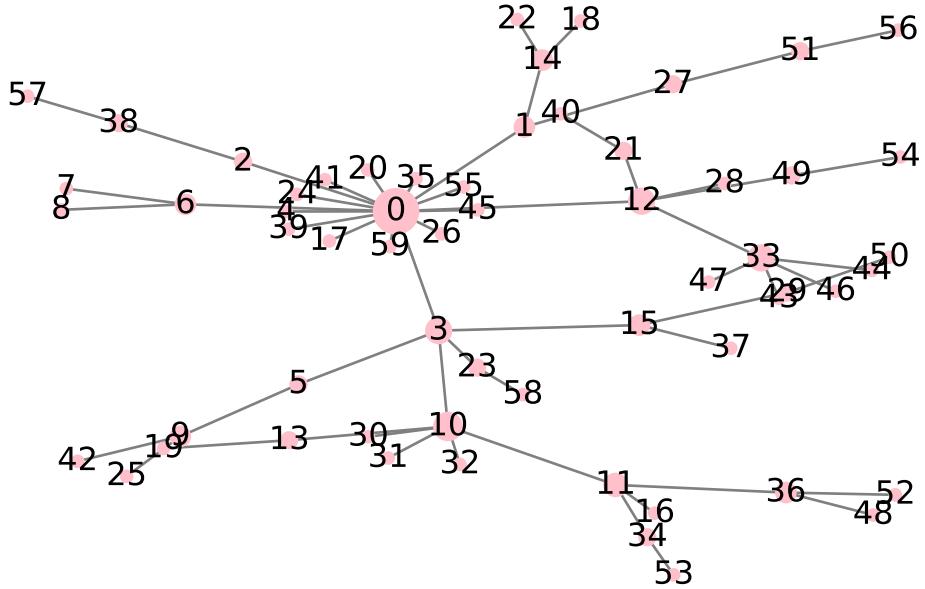
Let's take a look at centrality measures for a power-law graph of the same size. By construction, a BA graph has a hub-and-spoke structure.

```
G = nx.barabasi_albert_graph(60, 1, seed=2)
style["pos"] = nx.spring_layout(G, seed=3)
nx.draw(G, **style, node_size=120)
```



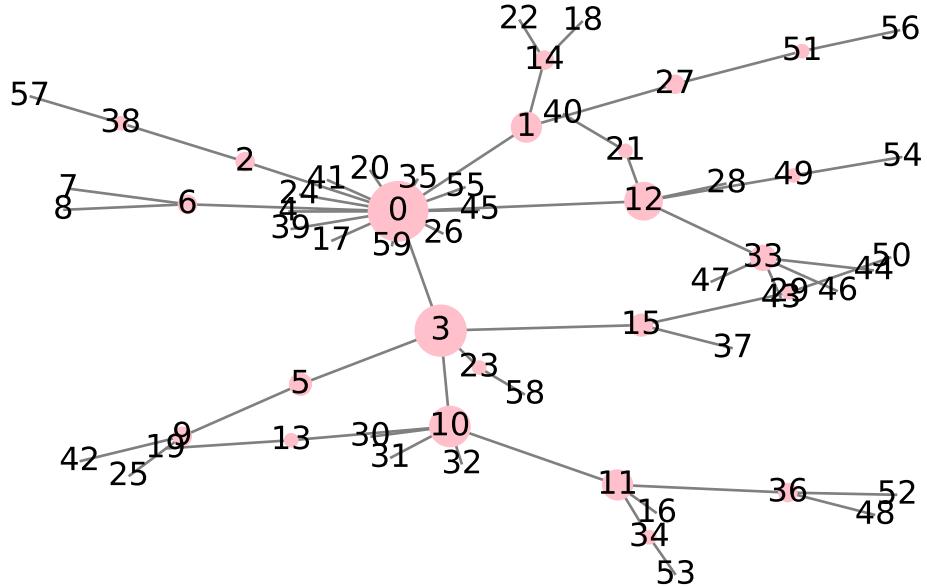
Degree centrality certainly notices the gregarious node 0:

```
centrality = pd.DataFrame( {"degree":nx.degree_centrality(G)}, index=G.nodes )
nx.draw(G, node_size=1000*centrality["degree"], **style)
```



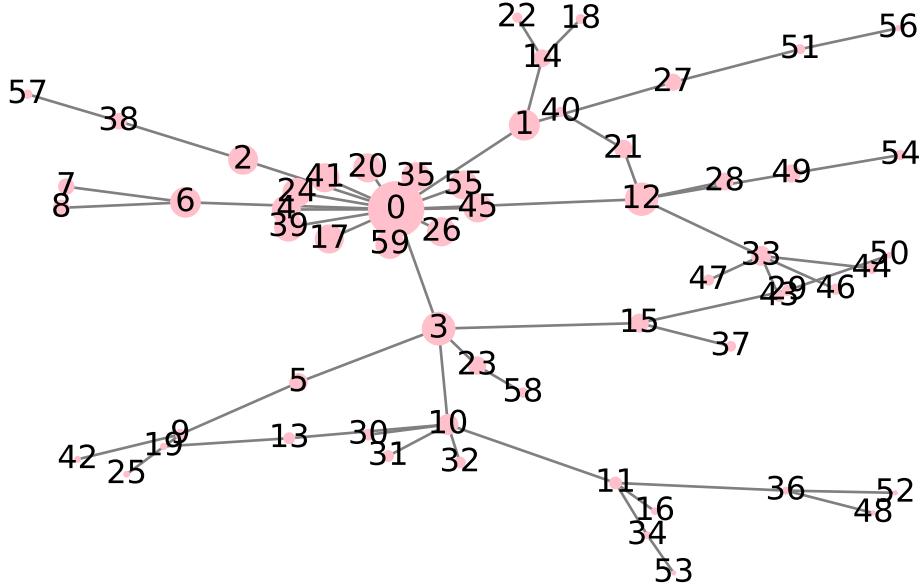
However, as you see above, the secondary hubs do not stand out much. Betweenness centrality highlights them quite nicely here:

```
centrality["between"] = pd.Series(nx.betweenness_centrality(G))
nx.draw(G, node_size=600*centrality["between"], **style)
```



On the other hand, eigenvector centrality puts a lot of emphasis on the friends of node 0, even the ones that are dead ends, at the expense of the secondary hubs:

```
centrality["eigen"] = pd.Series( nx.eigenvector_centrality_numpy(G) )
nx.draw(G, node_size=600*centrality["eigen"], **style)
```



This undesirable aspect of eigenvector centrality can be fixed through an extra normalization by the node degree, so that the hub node divides its “attention” into smaller parts. Such thinking leads to the *PageRank* algorithm, which is what put Google on the map for web searches.

7.5.5 Friendship paradox

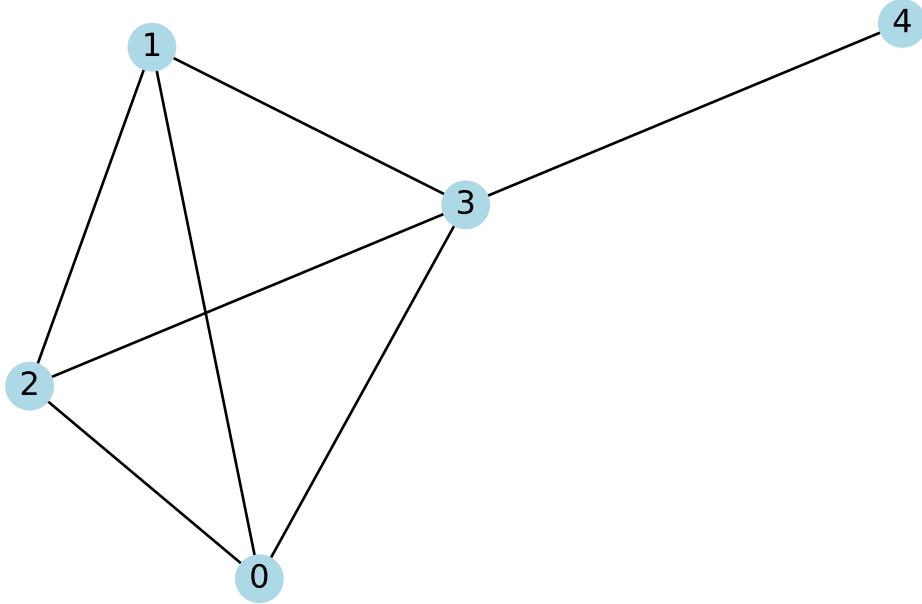
A surprising fact about social networks is that on average, your friends have more friends than you do, a fact that is called the **friendship paradox**. Let \mathbf{d} be an n -vector whose components are the degrees of the nodes in the network. On average, the number of “friends” (i.e., adjacent nodes) 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

$$\frac{\|\mathbf{d}\|_1}{n} \leq \frac{\mathbf{d}^T \mathbf{d}}{\|\mathbf{d}\|_1}. \quad (7.2)$$

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)
```

1.966666666666666 is less than 4.389830508474576

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

$$\frac{\|\mathbf{x}\|_1}{n} \leq \frac{\mathbf{x}^T \mathbf{d}}{\|\mathbf{d}\|_1}, \quad (7.3)$$

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)
```

0.07583936265862018 is less than 0.15936510918420194

In fact, the friendship paradox inequality for any vector \mathbf{x} is equivalent to \mathbf{x} having nonnegative correlation with the degree vector.

7.6 Communities

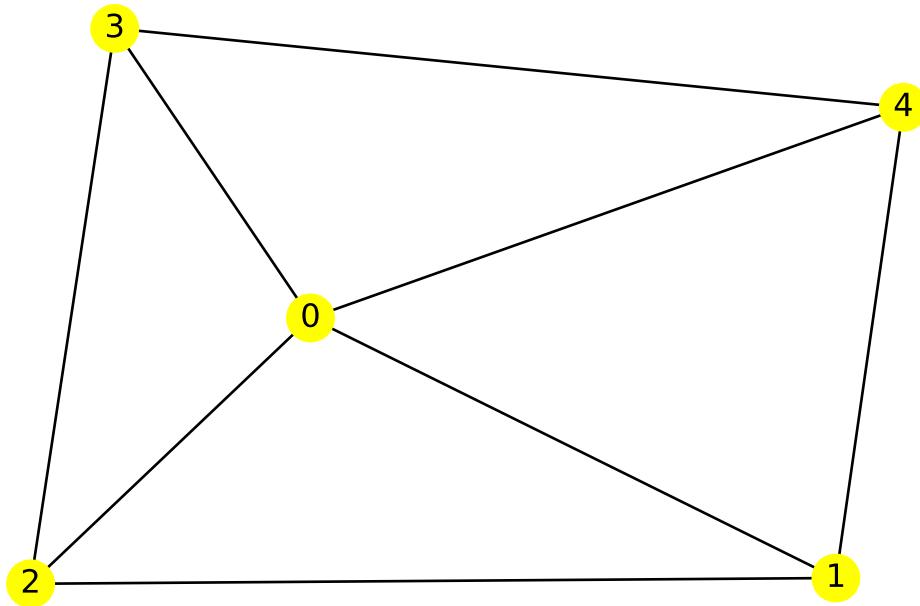
In applications, one may want to identify *communities* within a network. There are many ways to define this concept precisely. We will choose a **random-walk** model.

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

$$\underbrace{\frac{1}{3}}_{\text{to } 1} \cdot \underbrace{\frac{1}{3}}_{\text{to } 0} + \underbrace{\frac{1}{3}}_{\text{to } 3} \cdot \underbrace{\frac{1}{3}}_{\text{to } 0} = \frac{2}{9}.$$

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

$$\underbrace{\frac{1}{3}}_{\text{to } 0} \cdot \underbrace{\frac{1}{4}}_{\text{to } 2} + \underbrace{\frac{1}{3}}_{\text{to } 1} \cdot \underbrace{\frac{1}{3}}_{\text{to } 2} + \underbrace{\frac{1}{3}}_{\text{to } 3} \cdot \underbrace{\frac{1}{3}}_{\text{to } 2} = \frac{11}{36}.$$

This quantity is actually an inner product between the vector \mathbf{q} (probabilities of the prior location) and

$$\mathbf{w}_2 = \left[\frac{1}{4}, \frac{1}{3}, 0, \frac{1}{3}, 0 \right],$$

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,

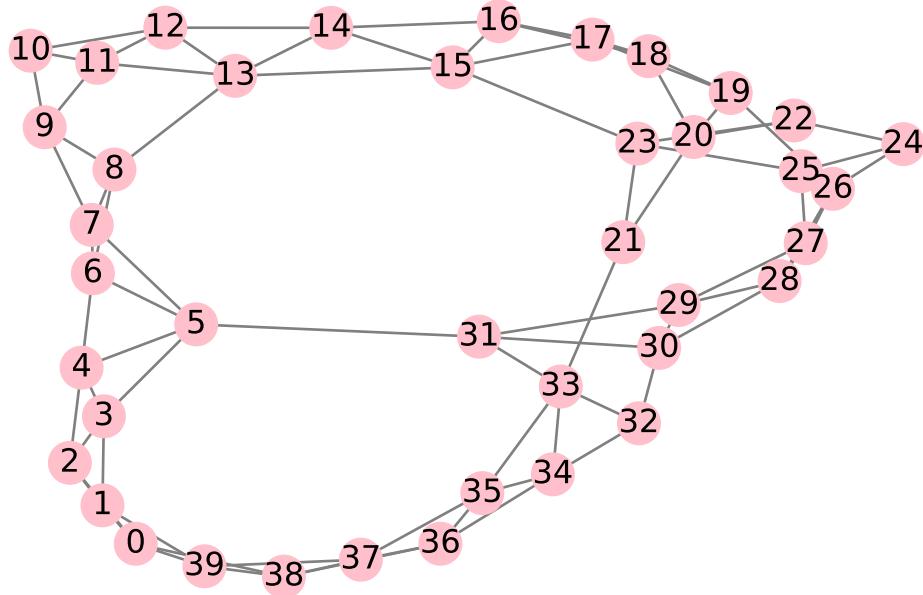
$$W_{ij} = \frac{1}{\deg(j)} A_{ij}.$$

7.6.1 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)
```

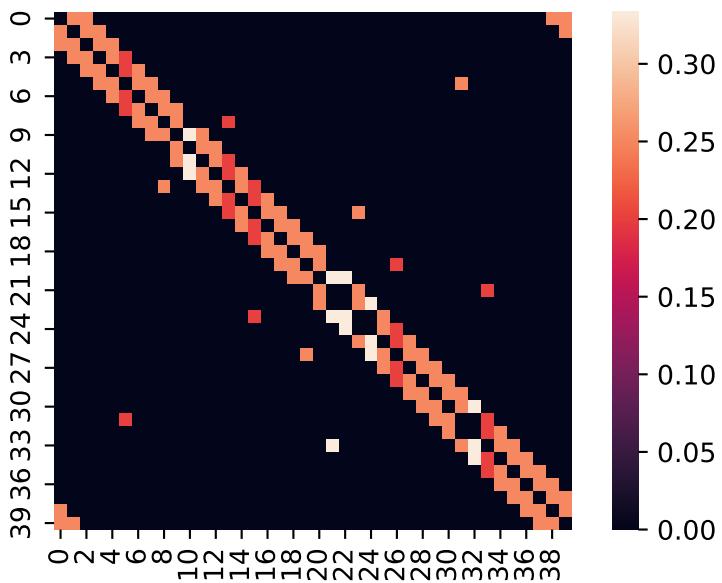


First, we construct the random-walk matrix \mathbf{W} .

```
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]

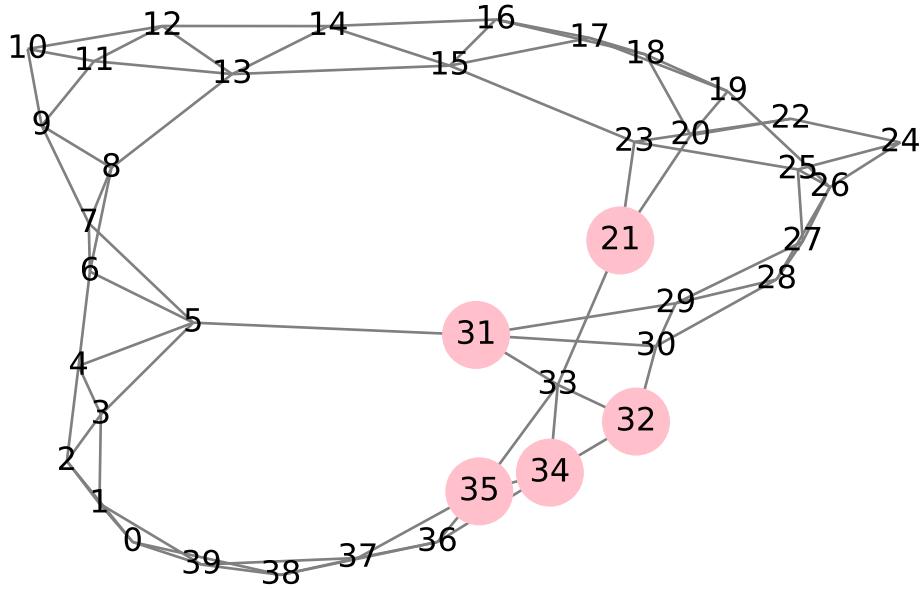
sns.heatmap(W.toarray()).set_aspect(1);
```



We set up a probability vector to start at node 0, and then use $\mathbf{W} \cdot \mathbf{p}$ 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)
```

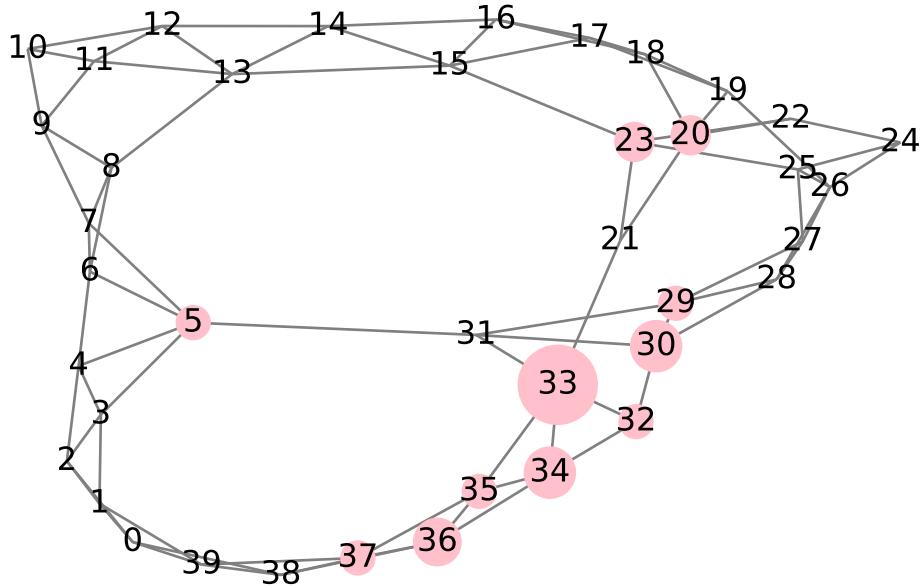
Total probability after 1 hop: 1.0



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



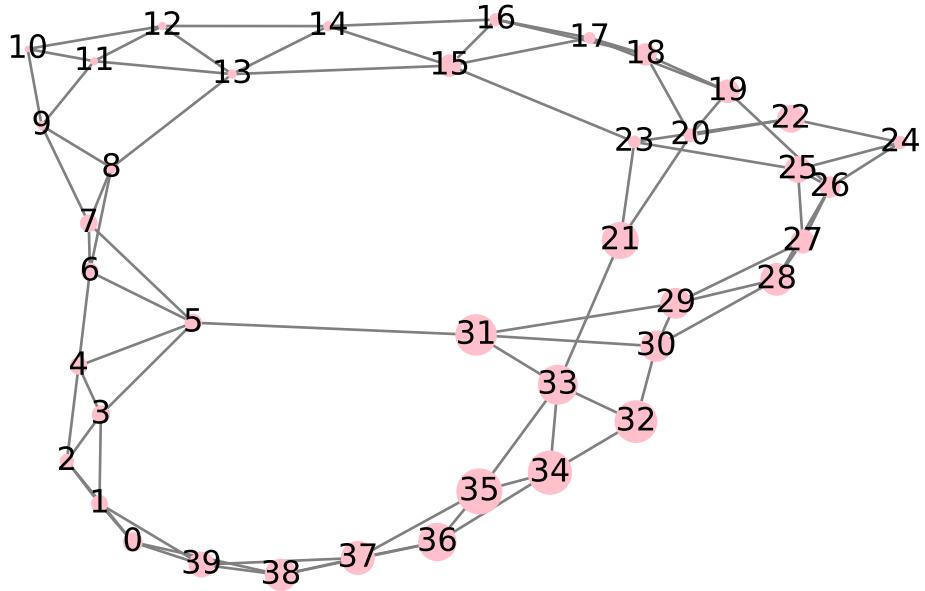
We'll take 3 more hops. That lets us penetrate a little into the distant nodes.

```

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

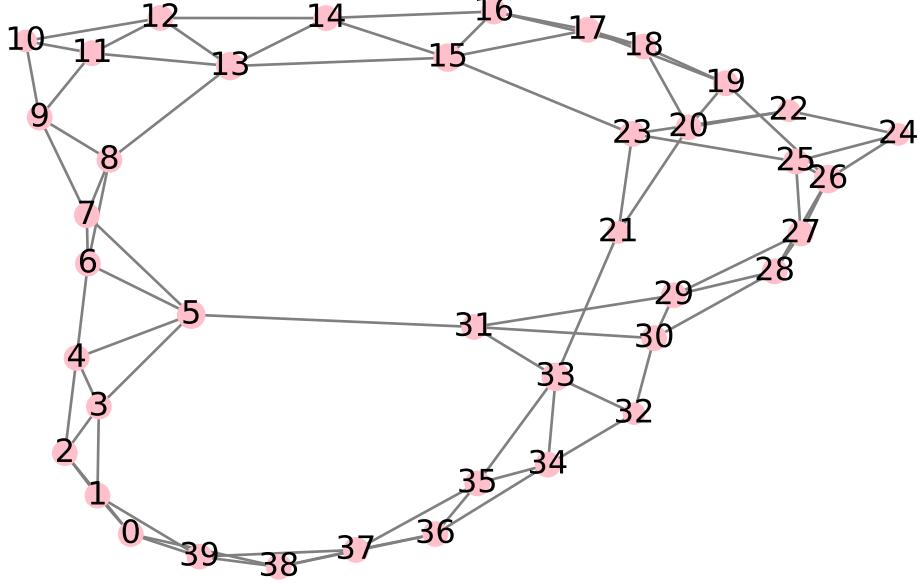


In the long run, the probabilities even out, as long as the graph is connected.

```

for k in range(200):
    p = W.dot(p)
nx.draw(WS, node_size=3000*p, **style)

```



7.6.2 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 λ^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.

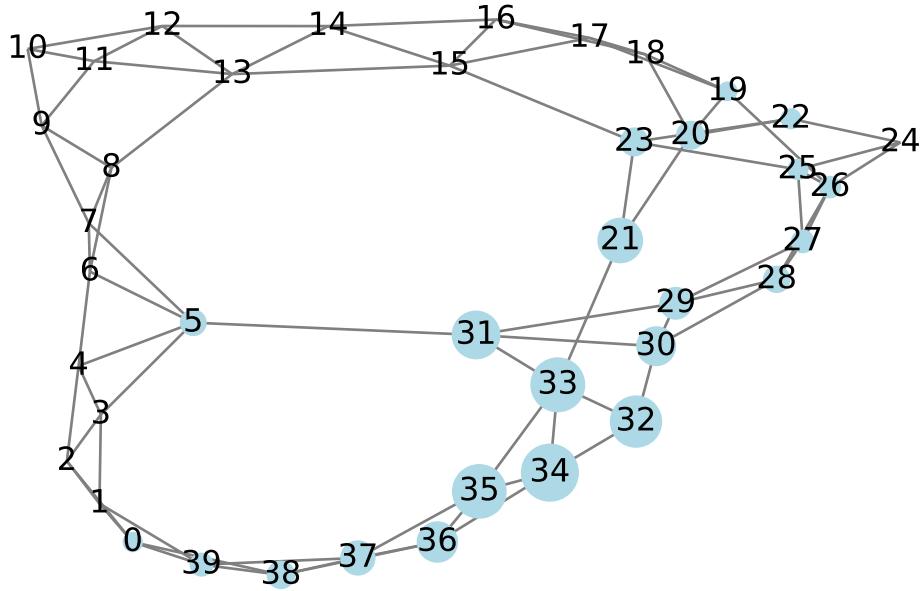
```
x = np.zeros(n)
mult = 1
for k in range(200):
    p = W.dot(p)
    mult *= lam
    x += mult*p

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 λ 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.

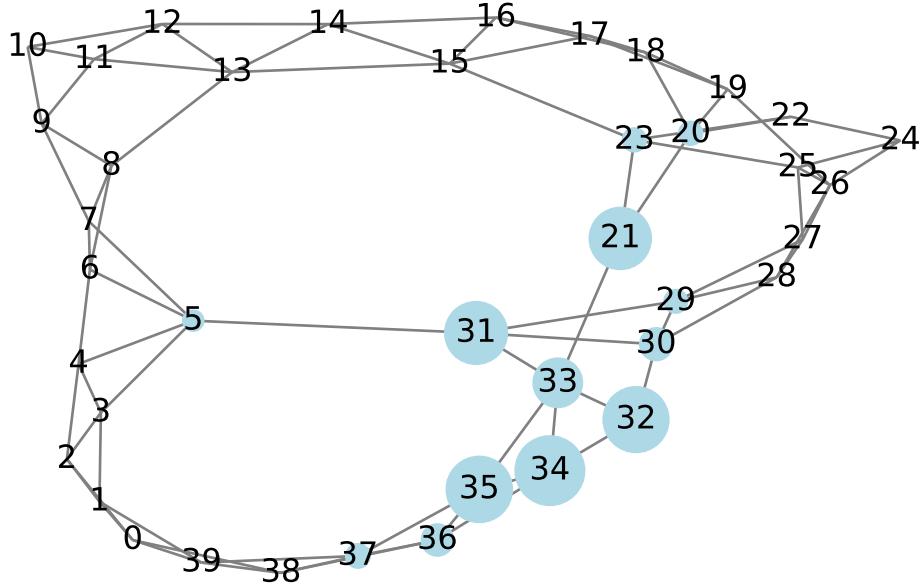
```

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.

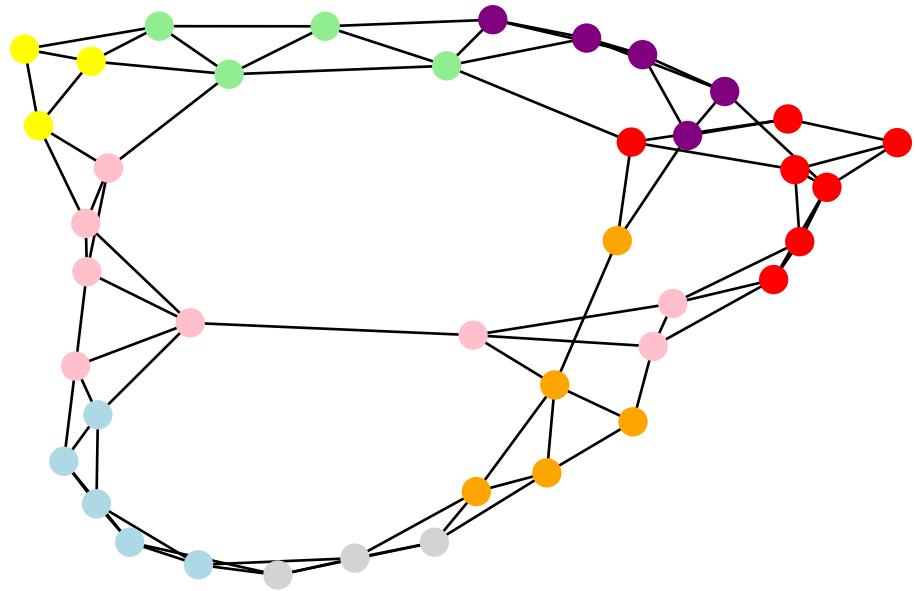
```
from networkx.algorithms.community import label_propagation_communities
comm = label_propagation_communities(WS)
[ print(c) for c in comm ];
```

```
{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","lightbrown"] )
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] )

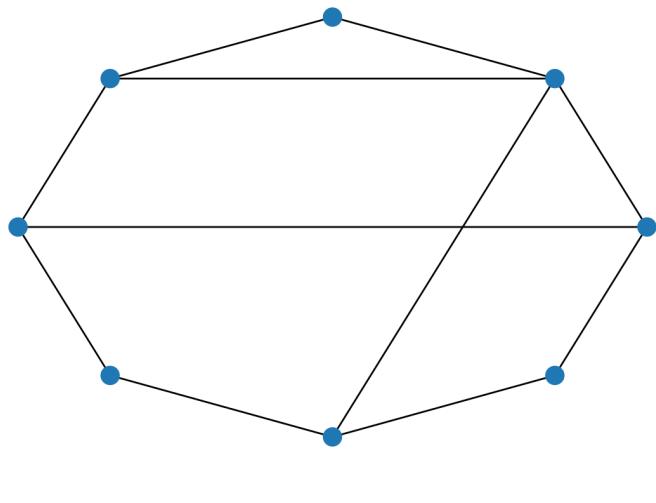
```



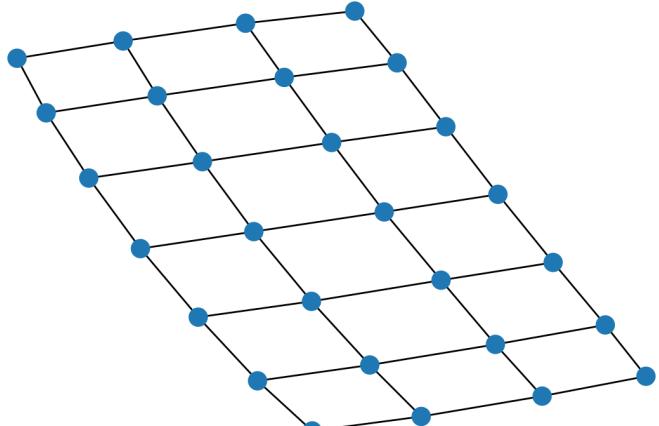
Exercises

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)

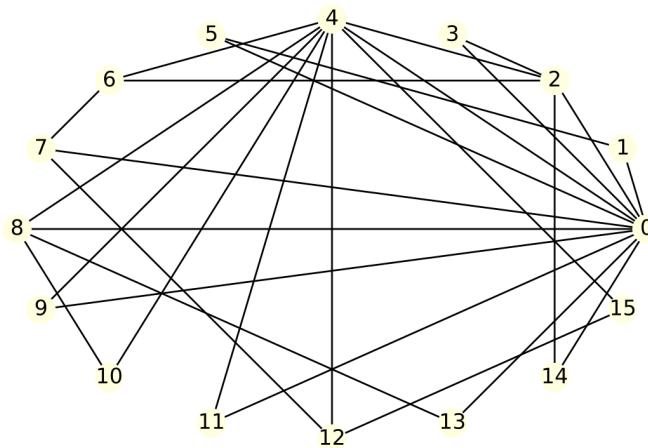


(c)

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.

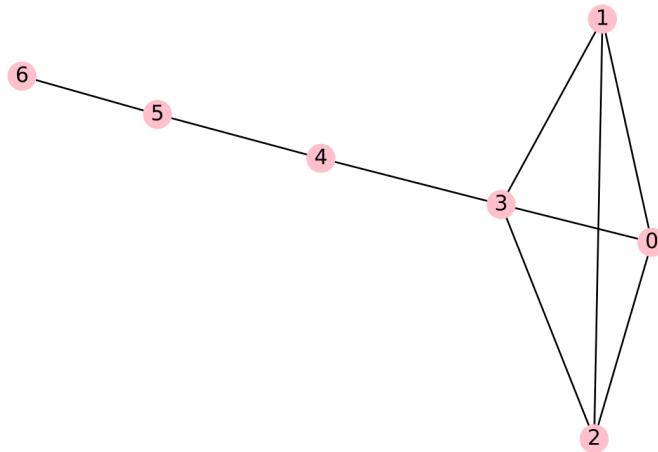
Exercise 7.6. Suppose that \mathbf{A} is the adjacency matrix of an undirected graph on n nodes. Let $\mathbf{1}$ be the n -vector whose components all equal 1, and let

$$\mathbf{d} = \mathbf{A}\mathbf{1}.$$

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 7.2. (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 θ . Simplify the inner product, square both sides, and show that it can be rearranged into Equation 7.2.)