

These observations are not coincidental. Mathematicians have proven the following:

- The principal components of a dataset are equal to the normalized eigenvectors of the dataset's covariance matrix.
- The variance along a principal direction is equal to the eigenvalue of the associated principal component.

Consequently, to uncover the first principal component, it is sufficient to do the following:

- 1 Compute the covariance matrix.
- 2 Find the eigenvector of the matrix with the largest eigenvalue. That eigenvector corresponds to the direction with the highest variance coverage.

We can extract the eigenvector with the largest eigenvalue using a straightforward algorithm called *power iteration*.

Key terminology

- *Covariance matrix*— $m @ m.T / m.shape[1]$, where m is a matrix with a mean of zero. The diagonal of the covariance matrix equals the variances along each axis of m .
- *Eigenvector*—A special type of vector associated with a matrix. If m is a matrix with eigenvector `eigen_vec`, then $m @ eigen_vec$ points in the same direction as `eigen_vec`. Also, if m is a covariance matrix, `eigen_vec` is a principal component.
- *Eigenvalue*—A numeric value associated with an eigenvector. If m is a matrix with eigenvector `eigen_vec`, then $m @ eigen_vec$ stretches the eigenvector by eigenvalue units. Therefore, the eigenvalue equals $\text{norm}(m @ eigen_vec) / \text{norm}(eigen_vec)$. The eigenvalue of a principal component equals the variance covered by that component.

14.4.1 Extracting eigenvectors using power iteration

Our goal is to obtain the eigenvectors of `cov_matrix`. The procedure for doing this is simple. We start by generating a random unit vector, `random_vector`.

Listing 14.45 Generating a random unit vector

```
np.random.seed(0)
random_vector = np.random.random(size=2)
random_vector /= norm(random_vector)
```

Next, we compute `cov_matrix @ random_vector`. This matrix-vector product both rotates and stretches our random vector. We normalize the new vector so that its magnitude is comparable to `random_vector` and then plot both the new vector and the random vector (figure 14.20). Our expectation is that the two vectors will point in different directions.

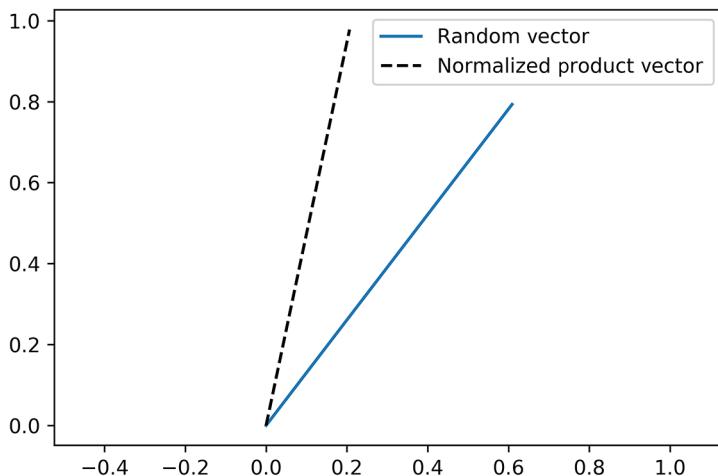


Figure 14.20 A plot of random_vector together with the normalized product of cov_matrix and random_vector. The two plotted vectors point in different directions.

Listing 14.46 Taking the product of cov_matrix and random_vector

```
product_vector = cov_matrix @ random_vector
product_vector /= norm(product_vector)

plt.plot([0, random_vector[0]], [0, random_vector[1]],
         label='Random Vector')
plt.plot([0, product_vector[0]], [0, product_vector[1]], linestyle='--',
         c='k', label='Normalized Product Vector')

plt.legend()
plt.axis('equal')
plt.show()
```

Our two vectors have nothing in common. Let's see what happens when we repeat the previous step by running `cov_matrix @ product_vector`. Next, we normalize and plot this additional vector along with the previously plotted `product_vector` (figure 14.21).

Listing 14.47 Taking the product of cov_matrix and product_vector

```
product_vector2 = cov_matrix @ product_vector
product_vector2 /= norm(product_vector2)

plt.plot([0, product_vector[0]], [0, product_vector[1]], linestyle='--',
         c='k', label='Normalized Product Vector')
plt.plot([0, product_vector2[0]], [0, product_vector2[1]], linestyle=':',
         c='r', label='Normalized Product Vector2')

plt.legend()
plt.axis('equal')
plt.show()
```

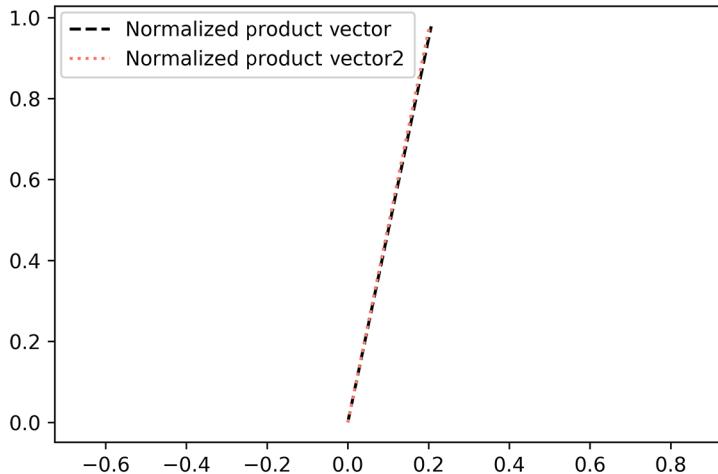


Figure 14.21 A plot of `product_vector` together with the normalized product of `cov_matrix` and `product_vector`. The two plotted vectors are identical. We've thus discovered an eigenvector of the covariance matrix.

Our `product_vector` vectors point in an identical direction! Therefore, `product_vector` is an eigenvector of `cov_matrix`. Basically, we've carried out a *power iteration*, which is a simple algorithm for eigenvector detection. We were lucky in our usage of the algorithm: a single matrix multiplication was enough to uncover the eigenvector. More commonly, a few additional iterations are required.

Power iteration works as follows:

- 1 Generate a random unit vector.
- 2 Multiply the vector by our matrix, and normalize the result. Our unit vector is rotated.
- 3 Iteratively repeat the previous step until the unit vector gets “stuck”—it won’t rotate anymore. By definition, it is now an eigenvector.

The power iteration is guaranteed to converge onto an eigenvector (if one exists). Generally, 10 iterations are more than sufficient to achieve convergence. The resulting eigenvector has the largest possible eigenvalue relative to the other eigenvectors of the matrix.

NOTE Some matrices have eigenvectors with negative eigenvalues. In such circumstances, the power iteration returns the eigenvector with the largest absolute eigenvalue.

Let's define a `power_iteration` function that takes a matrix as input. It returns an eigenvector and an eigenvalue as output. We test the function by running `power_iteration(cov_matrix)`.

Listing 14.48 Implementing the power iteration algorithm

```

np.random.seed(0)
def power_iteration(matrix):
    random_vector = np.random.random(size=matrix.shape[0])
    random_vector = random_vector / norm(random_vector)
    old_rotated_vector = random_vector
    for _ in range(10):
        rotated_vector = matrix @ old_rotated_vector
        rotated_vector = rotated_vector / norm(rotated_vector)
        old_rotated_vector = rotated_vector

    eigenvector = rotated_vector
    eigenvalue = norm(matrix @ eigenvector)
    return eigenvector, eigenvalue

eigenvector, eigenvalue = power_iteration(cov_matrix)
print(f"The extracted eigenvector is {eigenvector}")
print(f"Its eigenvalue is approximately {eigenvalue:.1f}")

The extracted eigenvector is [0.20223994 0.979336]
Its eigenvalue is approximately 541.8

```

The `power_iteration` function has extracted an eigenvector with an eigenvalue of approximately 541. This corresponds to the variance along the first principal axis. Hence, our eigenvector equals the first principal component.

NOTE You may have noticed that the extracted eigenvector in figure 14.21 stretches toward positive values in the plot. Meanwhile, the first principal component in figure 14.19 stretches toward negative values. As stated earlier, the principal component `pc` can be used interchangeably with `-pc` during PCA execution—projection onto principal directions will not be erroneously affected. The only noticeable effect will be a difference in reflection over the projected axes, as seen in figure 14.13.

Our function returns a single eigenvector with the largest eigenvalue. Consequently, `power_iteration(cov_matrix)` returns the principal component with the largest variance coverage. As stated earlier, the second principal component is also an eigenvector. Its eigenvalue corresponds to the variance along the second principal direction. Thus, that component is an eigenvector with the second largest eigenvalue. How do we find it? The solution requires just a few lines of code. That solution is not easy to understand without knowing higher mathematics, but we'll review its basic steps.

To extract the second eigenvector, we must eliminate all traces of the first eigenvector from `cov_matrix`. This process is known as *matrix deflation*. Once a matrix is deflated, its second-largest eigenvalue becomes its largest eigenvalue. To deflate `cov_matrix`, we must take the *outer product* of eigenvector with itself. That outer product is computed by taking the pairwise product of `eigenvector[i] * eigenvector[j]` for every possible value of `i` and `j`. The pairwise products are stored in a matrix `M`, where `M[i][j] =`

`eigenvector[i] * eigenvector[j]`. We can compute the outer product using two nested loops or using NumPy and running `np.outer(eigenvector, eigenvector)`.

NOTE Generally, the outer product is computed between two vectors `v1` and `v2`. That outer product returns a matrix `m` where `m[i][j]` equals `v1[i] * v2[j]`. During matrix deflation, both `v1` and `v2` are equal to `eigenvector`.

Listing 14.49 Computing the outer product of an eigenvector with itself

```
outer_product = np.outer(eigenvector, eigenvector)
for i in range(eigenvector.size):
    for j in range(eigenvector.size):
        assert outer_product[i][j] == eigenvector[i] * eigenvector[j]
```

Given the outer product, we can deflate `cov_matrix` by running `cov_matrix - eigenvalue * outer_product`. That basic operation produces a matrix whose primary eigenvector is equal to the second principal component.

Listing 14.50 Deflating the covariance matrix

```
deflated_matrix = cov_matrix - eigenvalue * outer_product
```

Running `product_iteration(deflated_matrix)` returns an eigenvector that we'll call `next_eigenvector`. Based on our discussion, we know that the following should be true:

- `next_eigenvector` equals the second principal component.
- Thus, `np.array([eigenvector, next_eigenvector])` equals a matrix of principal components that we call components.
- Executing `components @ centered_data` projects our dataset onto its principal directions.
- Plotting the projections should produce a horizontally positioned, cigar-shaped plot similar to the one in figure 14.8 or figure 14.13.

Next, we extract `next_eigenvector` and execute the aforementioned projections. We then plot the projections to confirm that our assumptions are true (figure 14.22).

Listing 14.51 Extracting the second principal component from the deflated matrix

```
np.random.seed(0)
next_eigenvector, _ = power_iteration(deflated_matrix)
components = np.array([eigenvector, next_eigenvector])
projections = components @ centered_data
plt.scatter(projections[0], projections[1])
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.axis('equal')
plt.show()
```

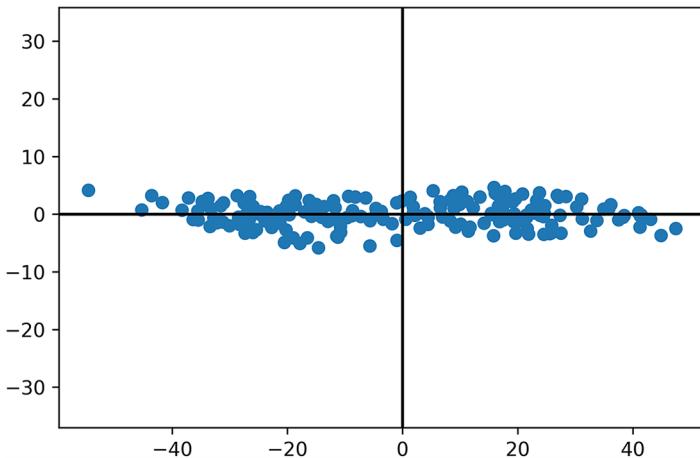


Figure 14.22 A plot of `centered_data` projected onto its principal components, in which the components were computed using power iteration. The plot is identical to figure 14.13, which was generated using scikit-learn’s PCA implementation.

NumPy matrix deflation computations

- `np.outer(eigenvector, eigenvector)`—Computes the outer product of an eigenvector with itself. Returns a matrix `m` where `m[i][j]` equals `eigenvector[i] * eigenvector[j]`.
- `matrix -= eigenvalue * np.outer(eigenvector, eigenvector)`—Deflates a matrix by removing all traces of the eigenvector with the largest eigenvalue from the matrix. Running `power_iteration(matrix)` returns an eigenvector with the next largest eigenvalue.

We’ve basically developed an algorithm for extracting the top K principal components of a matrix whose rows all average to zero. Given any such `centered_matrix`, the algorithm is executed as follows:

- 1 Compute the covariance matrix of `centered_matrix` by running `centered_matrix @ centered_matrix.T`.
- 2 Run `power_iteration` on the covariance matrix. The function returns an eigenvector of the covariance matrix (`eigenvector`), corresponding to the largest possible eigenvalue (`eigenvalue`). This eigenvector equals the first principal component.
- 3 Deflate the matrix by subtracting `eigenvalue * np.outer(eigenvector, eigenvector)`. Running `power_iteration` on the deflated matrix extracts the next principal component.

- 4 Repeat the previous step $K - 2$ more times to extract the top K principal components.

Let's implement the algorithm by defining `find_top_principal_components`. The function extracts the top K principal components from a `centered_matrix` input.

Listing 14.52 Extracting the top K principal components

Makes a copy of the matrix so we can deflate that copy without modifying the original

Extracts the top K principal components from a matrix whose rows all average to zero. The value of K is preset to 2.

```
def find_top_principal_components(centered_matrix, k=2):
    cov_matrix = centered_matrix @ centered_matrix.T
    cov_matrix /= centered_matrix[1].size
    return find_top_eigenvectors(cov_matrix, k=k)
```



```
def find_top_eigenvectors(matrix, k=2):
    matrix = matrix.copy()
    eigenvectors = []
    for _ in range(k):
        eigenvector, eigenvalue = power_iteration(matrix)
        eigenvectors.append(eigenvector)
        matrix -= eigenvalue * np.outer(eigenvector, eigenvector)

    return np.array(eigenvectors)
```

Principal components are simply the top eigenvectors of the covariance matrix (where eigenvector rank is determined by the eigenvalues). To emphasize this point, we define a separate function for extracting the top K eigenvectors of any matrix.

We defined a function for extracting the top K principal components of a dataset. The components allow us to project the dataset onto its top K principal directions. These directions maximize data dispersion along K axes. The remaining data axes can thus be disregarded, shrinking the coordinate column size to K . Consequently, we can reduce any dataset to K dimensions.

Basically, we're now able to run PCA from scratch without relying on scikit-learn. We can use PCA to reduce an N -dimensional dataset to K dimensions (where N is the column count of an input data matrix). To run the algorithm, we must execute the following steps:

- 1 Compute the mean along each of the axes in the dataset.
- 2 Subtract the mean from every axis, thus centering the dataset at the origin.
- 3 Extract the top K principal components of the centered dataset using the `find_top_principal_components` function.
- 4 Take the matrix product between the principal components and the centered dataset.

Let's implement these steps in a single function named `reduce_dimensions`. Why not name the function `pca`? Well, the first two steps of PCA require us to centralize our data. However, we'll soon learn that dimension reduction can be achieved without centralization. Thus, we pass an optional `centralize_data` parameter into our function.

We preset the parameter to True, guaranteeing that the function executes PCA under default conditions.

Listing 14.53 Defining a `reduce_dimensions` function

The function takes as input a data matrix whose columns correspond to axes.

This is consistent with the input orientation of scikit-learn's `fit_transform` method. The function then reduces the matrix from N columns to k columns.

```
def reduce_dimensions(data, k=2, centralize_data=True):    ↪
    data = data.T.copy()                                ↪ Data is transposed so that it remains
    if centralize_data:                                ↪ consistent with the expected input
        for i in range(data.shape[0]):                  ↪ into find_principal_components.
            data[i] -= data[i].mean()

    principal_components = find_top_principal_components(data)
    return (principal_components @ data).T
```

Optionally centralizes the data by subtracting its mean so the new mean equals 0

Let's test `reduce_dimensions` by applying it to our previously analyzed flower measurements data. We reduce that data to 2D using our custom PCA implementation and then visualize the results (figure 14.23). Our plot should be consistent with the plot in figure 14.18.

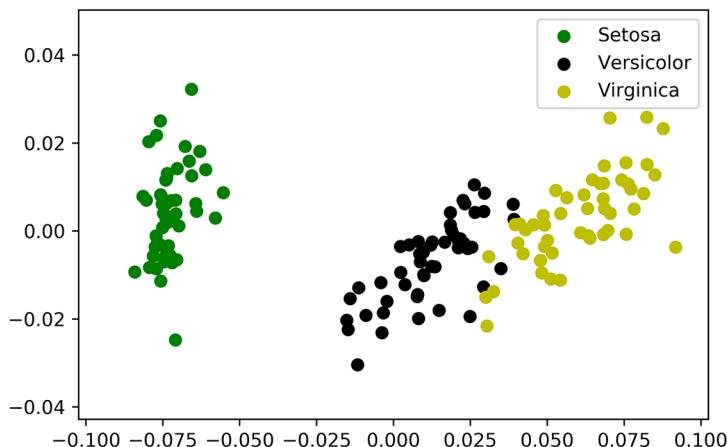


Figure 14.23 4D normalized flower measurements reduced to two dimensions using a custom PCA implementation. The plot is identical to the PCA output generated by scikit-learn.

Listing 14.54 Reducing flower data to 2D using a custom PCA implementation

```
np.random.seed(0)
dim_reduced_data = reduce_dimensions(flower_measurements)
visualize_flower_data(dim_reduced_data)
```

Our plot perfectly resembles scikit-learn's PCA output. We have reengineered scikit-learn's implementation, but with one big difference: in our function, centralization is optional. This will prove useful! As we've discussed, we can't reliably perform centralization on normalized data. Also, our flower dataset has been normalized to eliminate unit differences. Consequently, we cannot reliably run PCA on `flower_measurements`. One alternative is to bypass centralization by passing `centralize_data=False` into `reduce_dimensions`. This, of course, violates many assumptions of the PCA algorithm. However, the output could still be useful. What will happen if we reduce the dimensions of `flower_measurements` without centralization? Let's find out by setting `centralize_data` to `False` and plotting the results (figure 14.24).

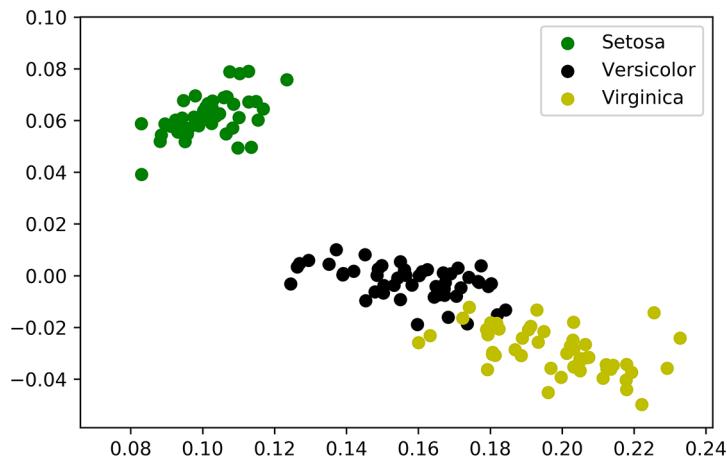


Figure 14.24 4D normalized flower measurements reduced to two dimensions without centralization. Each plotted flower point is colored based on its species. The three species continue to fall into three clusters. However, the plot no longer resembles our PCA output.

Listing 14.55 Running `reduce_dimensions` without centralization

```
np.random.seed(3)      ←
dim_reduced_data = reduce_dimensions(flower_measurements,
                                      centralize_data=False)
visualize_flower_data(dim_reduced_data)
```

As previously stated, the randomness of eigenvector extraction can influence 2D plot orientation. Here, we seed the algorithm to ensure that the orientation aligns with another plot, presented later (figure 14.25).

In the output, the three species of flowers continue to separate into three clusters. Furthermore, *Setosa* remains spatially distinct from other species. However, there are changes in the plot. *Setosa* forms a tighter cluster than previously observed in the PCA results. This begs the question, is our latest plot as comprehensive as our PCA output? In other words, does it continue to represent 97% of the total data variance? We can

check by measuring the variance of `dim_reduced_data` and dividing it by the total variance of `flower_measurements`.

Listing 14.56 Checking the variance of data reduced without centralization

```
variances = [sum(data[:, i].var() for i in range(data.shape[1]))  
            for data in [dim_reduced_data, flower_measurements]]  
dim_reduced_var, total_var = variances  
percent_coverege = 100 * dim_reduced_var / total_var  
print(f"Our plot covers {percent_coverege:.2f}% of the total variance")  
  
Our plot covers 97.29% of the total variance
```

Our 2D variance coverage is maintained. Even though its value fluctuates slightly, the coverage remains at approximately 97%. We thus can reduce dimensionality without relying on centralization. However, centralization remains a defining feature of PCA, so our modified technique needs a different name. Officially, as mentioned earlier, the technique is called *singular value decomposition* (SVD).

WARNING Unlike PCA, SVD is not guaranteed to maximize the variance for each axis in the reduced output. However, in most real-world circumstances, SVD is able to dimensionally reduce data to a very practical degree.

The mathematical properties of SVD are complicated and lie beyond the scope of this book. Nonetheless, computer scientists can use these properties to execute SVD very efficiently, and these optimizations have been incorporated into scikit-learn. In the next subsection, we utilize scikit-learn's optimized SVD implementation.

14.5 Efficient dimension reduction using SVD and scikit-learn

Scikit-learn contains a dimension-reduction class called `TruncatedSVD` that is designed for optimal execution of SVD. Let's import `TruncatedSVD` from `sklearn.decomposition`.

Listing 14.57 Importing TruncatedSVD from scikit-learn

```
from sklearn.decomposition import TruncatedSVD
```

Applying `TruncatedSVD` to our `flower_measurements` data is easy. First, we need to run `TruncatedSVD(n_components=2)` to create an `svd_object` object capable of reducing data to two dimensions. Then, we can execute SVD by running `svd_object.fit_predict(flower_measurements)`. That method call returns a two-dimensional `svd_transformed_data` matrix. Next, we apply `TruncatedSVD` and plot our results (figure 14.25). The plot should resemble our custom SVD output, shown in figure 14.24.

NOTE Unlike the `PCA` class, scikit-learn's `TruncatedSVD` implementation requires an `n_components` input. The default value for that parameter is pre-set to 2.

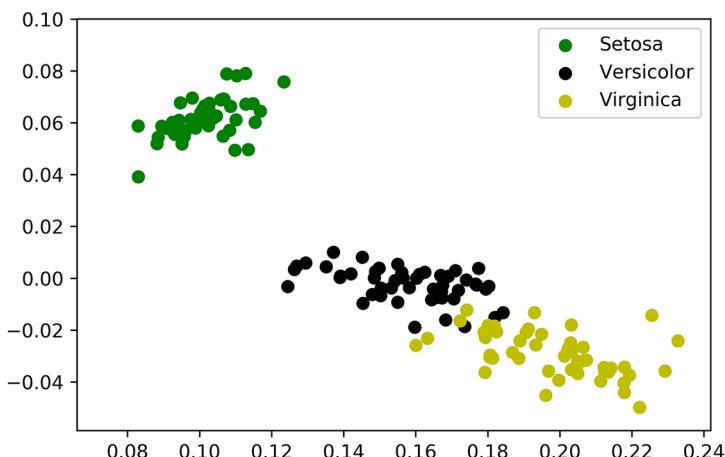


Figure 14.25 4D normalized flower measurements reduced to two dimensions using scikit-learn’s SVD implementation. The output is identical to figure 14.24, which was generated using our custom SVD implementation.

Listing 14.58 Running SVD using scikit-learn

```
svd_object = TruncatedSVD(n_components=2)
svd_transformed_data = svd_object.fit_transform(flower_measurements)
visualize_flower_data(svd_transformed_data)
```

Not surprisingly, scikit-learn’s results are identical to our custom SVD implementation. Scikit-learn’s algorithm is faster and more memory efficient, but its output does not diverge from ours.

NOTE The outputs will not diverge when we reduce the number of dimensions. However, as the dimension count goes up, our implementation will become less precise because minor errors will creep into our eigenvector calculations. These minor errors will be magnified with each computed eigenvector. Scikit-learn, on the other hand, uses mathematical tricks to limit these errors.

We can further verify the overlap between outputs by comparing variance coverage. Our `svd_object` has an `explained_variance_ratio_` attribute that holds an array of the fractional variances covered by each reduced dimension. Summing over `100 * explained_variance_ratio_` should return the percent of total variance covered in the 2D plot. Based on our analysis, we expect that output to approximate 97.29%. Let’s confirm.

Listing 14.59 Extracting variance from scikit-learn's SVD output

The attribute is a NumPy array containing fractional coverage for each axis. Multiplying by 100 converts these fractions into percentages.

```
percent_variance_coverages = 100 * svd_object.explained_variance_ratio_ ←
x_axis_coverage, y_axis_coverage = percent_variance_coverages ←
total_2d_coverage = x_axis_coverage + y_axis_coverage
print(f"Our Scikit-Learn SVD output covers {total_2d_coverage:.2f}% of "
      "the total variance")
```

Our Scikit-Learn SVD output covers 97.29% of the total variance

Each *i*th element of the array corresponds to the variance coverage of the *i*th axis.

Common scikit-learn SVD methods

- `svd_object = TruncatedSVD(n_components=K)`—Creates an SVD object capable of reducing input data to K dimensions.
- `svd_transformed_data = svd_object.fit_transform(data)`—Executes SVD on inputted data using an initialized TruncatedSVD object. The `fit_transform` method assumes that the columns of the data matrix correspond to spatial axes. The dimensionally reduced results are stored in the `svd_transformed_data` matrix.
- `svd_object.explained_variance_ratio_`—Returns the fractional variance coverage associated with each dimensionally reduced axis of a fitted TruncatedSVD object.

Scikit-learn's optimized SVD implementation can decrease data from tens of thousands of dimensions to only a few hundred or a few dozen. The shrunken data can more efficiently be stored, transferred, and processed by predictive algorithms. Many real-world data tasks require SVD for data shrinkage before analysis. Applications range from image compression, to audio-noise removal, to natural language processing. NLP, in particular, is dependent on the algorithm due to the bloated nature of text data. As we discussed in the previous section, real-world documents form very large matrices whose column counts are way too high. We cannot multiply such matrices efficiently and therefore cannot compute text similarities. Fortunately, SVD makes these document matrices much more manageable. SVD allows us to shrink text-matrix column counts while retaining most of the variance, so we can compute large-scale text similarities in a timely manner. These text similarities can then be used to cluster the input documents.

In the subsequent section, we finally analyze large document datasets. We learn how to clean and cluster these datasets while also visualizing the clustering output. Our use of SVD will prove absolutely fundamental to this analysis.

Summary

- Reducing dataset dimensionality can simplify certain tasks, like clustering.
- We can reduce a 2D dataset to one dimension by rotating the data about the origin until the data points lie close to the x-axis. Doing so maximizes data spread along the x-axis, thus allowing us to delete the y-axis. However, the rotation requires us to first centralize the dataset so its mean coordinates lie at the origin.
- Rotating the data toward the x-axis is analogous to rotating the x-axis toward the *first principal direction*. The first principal direction is the linear direction in which data variance is maximized. The *second principal direction* is perpendicular to the first. In a 2D dataset, that direction represents the remaining variance not covered by the first principal direction.
- Dimensional reduction can be carried out using *principal component analysis* (PCA). PCA uncovers a dataset's principal directions and represents them as using unit vectors called *principal components*. The product of the centralized data matrix and the principal components swaps the data's standard axes with the principal directions. This axis swap is called a *projection*. When we project our data onto the principal directions, we maximize data dispersion along some axes and minimize it along others. Axes with minimized dispersion can be deleted.
- We can extract principal components by computing a *covariance matrix*: the matrix product of a centered dataset with itself, divided by the dataset size. The diagonal of that matrix represents the axes' variance values.
- Principal components are the *eigenvectors* of the covariance variance. Thus, by definition, the normalized product of the covariance matrix and each principal component is equal to that principal component.
- We can extract the top eigenvector of a matrix using the *power iteration* algorithm. Power iteration consists of repeated multiplication and normalization of a vector by a matrix. Applying power iteration to a covariance matrix returns the first principal component.
- By applying *matrix deflation*, we can eliminate all traces of an eigenvector. Deflating the covariance matrix and reapplying the power iteration returns the second principal component. Repeating that process iteratively returns all principal components.
- PCA is sensitive to units of measurement. Normalizing our input data reduces that sensitivity. However, as a consequence, the normalized data will be proximate to its mean. This is a problem because PCA requires us to subtract the mean from each axis value to centralize the data. Subtracting proximate values leads to floating-points errors.
- We can avoid these floating-point errors by refusing to centralize our data before running dimension reduction. The resulting output captures data variance to a sufficiently meaningful degree. This modified technique is called *singular value decomposition* (SVD).

15

NLP analysis of large text datasets

This section covers

- Vectorizing texts using scikit-learn
- Dimensionally reducing vectorized text data
- Clustering large text datasets
- Visualizing text clusters
- Concurrently displaying multiple visualizations

Our previous discussions of natural language processing (NLP) techniques focused on toy examples and small datasets. In this section, we execute NLP on large collections of real-world texts. This type of analysis is seemingly straightforward, given the techniques presented thus far. For example, suppose we're doing market research across multiple online discussion forums. Each forum is composed of hundreds of users who discuss a specific topic, such as politics, fashion, technology, or cars. We want to automatically extract all the discussion topics based on the contents of the user conversations. These extracted topics will be used to plan a marketing campaign, which will target users based on their online interests.

How do we cluster user discussions into topics? One approach would be to do the following:

- 1 Convert all discussion texts into a matrix of word counts using techniques discussed in section 13.
- 2 Dimensionally reduce the word count matrix using singular value decomposition (SVD). This will allow us to efficiently complete all pairs of text similarities with matrix multiplication.
- 3 Utilize the matrix of text similarities to cluster the discussions into topics.
- 4 Explore the topic clusters to identify useful topics for our marketing campaign.

Of course, in real life, this simple analysis is not as straightforward as it seems. Unanswered questions still remain. How do we efficiently explore the topic clusters without reading all the clustered texts one at a time? Also, which of the clustering algorithms introduced in section 10 should we utilize to cluster the discussions?

Even at the level of pairwise text comparison, we face certain questions. How do we deal with common uninformative words such as *the*, *it*, and *they*? Should we penalize them? Ignore them? Filter them out entirely? What about other common, corpus-specific words such as the names of the websites that host the discussion forums?

All of these questions have answers that can best be understood by actually exploring an online forum dataset containing thousands of texts. Scikit-learn includes one such real-world dataset among its example data collections. In this section, we load, explore, and cluster this large dataset of online forums. Python’s external data science libraries, such as scikit-learn and NumPy, will prove invaluable in this real-world analysis.

15.1 Loading online forum discussions using scikit-learn

Scikit-learn provides us with data from Usenet, which is a well-established online collection of discussion forums. These Usenet forums are called *newsgroups*. Each individual newsgroup focuses on some topic of discussion, which is briefly outlined in the newsgroup name. Users in a newsgroup converse by posting messages. These user posts are not limited in length, and hence some posts can be quite long. Both the diversity and the varying lengths of the posts will give us a chance to expand our NLP skills. For training purposes, the scikit-learn library provides access to over 10,000 posted messages. We can load these newsgroup posts by importing `fetch_20newsgroups` from `sklearn.datasets`. Calling `fetch_20newsgroups()` returns a `newsgroups` object that contains the textual data. Furthermore, optionally passing `remove=('headers', 'footers')` into the function call removes redundant information from the text. (That deleted metadata does not correspond to meaningful post content.) Listing 15.1 loads the newsgroup data while filtering redundant information.

WARNING The newsgroups dataset is quite large. For this reason, it is not pre-packaged with scikit-learn. Running `fetch_20newsgroups` forces scikit-learn to download and store the dataset on a local machine, so an internet connection is required when the dataset is first fetched. All subsequent calls to `fetch_20newsgroups` will load the dataset locally without requiring an internet connection.

Listing 15.1 Fetching the newsgroup dataset

```
from sklearn.datasets import fetch_20newsgroups
newsgroups = fetch_20newsgroups(remove=('headers', 'footers'))
```

The newsgroups object contains posts from 20 different newsgroups. As mentioned, each newsgroup's discussion theme is outlined in its name. We can view these newsgroup names by printing newsgroups.target_names.

Listing 15.2 Printing the names of all 20 newsgroups

```
print(newsgroups.target_names)

['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x',
'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball',
'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space',
'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast',
'talk.politics.misc', 'talk.religion.misc']
```

The newsgroup categories vary greatly, from space exploration (*sci.space*) to cars (*rec.auto*) to electronics (*sci.electronics*). Some of the categories are very broad. For instance, politics (*talk.politics.misc*) can cover a wide range of political themes. Other categories are very narrow in scope: for example, *comp.sys.mac.hardware* focuses on Mac hardware, while *comp.sys.ibm.pc.hardware* focuses on PC hardware. Categorically, these two newsgroups are exceedingly similar: the only differentiator is whether the computer hardware belongs to a Mac or a PC. Sometimes categorical differences are subtle; boundaries between text topics are fluid and not necessarily etched in stone. We need to keep this in mind later in the section when we cluster the newsgroup posts.

Now, let's turn our attention to the actual newsgroup texts, which are stored as a list in the newsgroups.data attribute. For example, newsgroups.data[0] contains the text of the first stored newsgroup post. Let's output that post.

Listing 15.3 Printing the first newsgroup post

```
print(newsgroups.data[0])
```

I was wondering if anyone out there could enlighten me on this car I saw the other day. It was a 2-door sports car, looked to be from the late 60s/early 70s. It was called a Bricklin. The doors were really small. In addition, the front bumper was separate from the rest of the body. This is all I know. If anyone can tell me a model name, engine specs, years of production, where this car is made, history, or whatever info you have on this funky looking car, please e-mail.

The post is about a car. It probably was posted to the car discussion newsgroup, *rec.autos*. We can confirm by printing newsgroups.target_names[newsgroups.target[0]].

NOTE `newsgroups.target[i]` returns the index of the newsgroup name associated with the *i*th document.

Listing 15.4 Printing the newsgroup name at index 0

```
origin = newsgroups.target_names[newsgroups.target[0]]
print(f"The post at index 0 first appeared in the '{origin}' group")

The post at index 0 first appeared in the 'rec.autos' group
```

As we predicted, our car-related post appeared in the car discussion group. The presence of a few keywords such as *car*, *bumper*, and *engine*, was sufficient to make this distinction. Of course, this is just one post out of many. Categorizing the remaining posts may not be so easy.

Let's dive deeper into our newsgroup dataset by printing out the dataset size.

Listing 15.5 Counting the number of newsgroup posts

```
dataset_size = len(newsgroups.data)
print(f"Our dataset contains {dataset_size} newsgroup posts")

Our dataset contains 11314 newsgroup posts
```

Our dataset contains over 11,000 posts. Our goal is to cluster these posts by topic, but carrying out text clustering on this scale requires computational efficiency. We need to efficiently compute newsgroup post similarities by representing our text data as a matrix. To do so, we need to transform each newsgroup post into a term-frequency (TF) vector. As discussed in section 13, a TF vector's indices map to word counts in a document. Previously, we computed these vectorized word counts using custom functions. Now we will compute them using scikit-learn.

15.2 Vectorizing documents using scikit-learn

Scikit-learn provides a built-in class for transforming input texts into TF vectors: `CountVectorizer`. Initializing `CountVectorizer` creates a vectorizer object capable of vectorizing our texts. Next, we import `CountVectorizer` from `sklearn.feature_extraction.text` and initialize the class.

Listing 15.6 Initializing a CountVectorizer object

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
```

We are now ready to vectorize the texts stored in the `newsgroups.data` list. All we need to do is run `vectorizer.fit_transform(newsgroups.data)`. The method call returns the TF matrix corresponding to the vectorized newsgroup posts. As a reminder,

a TF matrix stores the counts of words (columns) across all texts (rows). Let's vectorize the posts and then print the resulting TF matrix.

Listing 15.7 Computing a TF matrix with scikit-learn

```
tf_matrix = vectorizer.fit_transform(newsgroups.data)
print(tf_matrix)

(0, 108644)      4
(0, 110106)      1
(0, 57577)       2
(0, 24398)       2
(0, 79534)       1
(0, 100942)      1
(0, 37154)       1
(0, 45141)       1
(0, 70570)       1
(0, 78701)       2
(0, 101084)      4
(0, 32499)       4
(0, 92157)       1
(0, 100827)      6
(0, 79461)       1
(0, 39275)       1
(0, 60326)       2
(0, 42332)       1
(0, 96432)       1
(0, 67137)       1
(0, 101732)      1
(0, 27703)       1
(0, 49871)       2
(0, 65338)       1
(0, 14106)       1
:
:
(11313, 55901)   1
(11313, 93448)   1
(11313, 97535)   1
(11313, 93393)   1
(11313, 109366)  1
(11313, 102215)  1
(11313, 29148)   1
(11313, 26901)   1
(11313, 94401)   1
(11313, 89686)   1
(11313, 80827)   1
(11313, 72219)   1
(11313, 32984)   1
(11313, 82912)   1
(11313, 99934)   1
(11313, 96505)   1
(11313, 72102)   1
(11313, 32981)   1
(11313, 82692)   1
(11313, 101854)  1
```

```
(11313, 66399)    1
(11313, 63405)    1
(11313, 61366)    1
(11313, 7462)      1
(11313, 109600)   1
```

Our printed `tf_matrix` does not appear to be a NumPy array. What sort of data structure is it? We can check by printing `type(tf_matrix)`.

Listing 15.8 Checking the data type of `tf_matrix`

```
print(type(tf_matrix))

<class 'scipy.sparse.csr.csr_matrix'>
```

The matrix is a SciPy object called `csr_matrix`. *CSR* stands for *compressed sparse row*, which is a storage format for compressing matrices that are composed mostly of zeros. These mostly empty matrices are referred to as *sparse matrices*. They can be made smaller by storing only the nonzero elements. This compression leads to more efficient memory usage and faster computation. Large-scale text-based matrices are usually very sparse, since a single document normally contains just a small percentage of the total vocabulary. Thus, scikit-learn automatically converts the vectorized text to the CSR format. The conversion is carried out using a `csr_matrix` class imported from SciPy.

This interplay between various external data science libraries is useful but also a bit confusing. In particular, the differences between a NumPy array and a SciPy CSR matrix can be tricky for a newcomer to grasp. This is because arrays and CSR matrices share some, but not all, attributes. Also, arrays and CSR matrices are compatible with some, but not all, NumPy functions. To minimize confusion, we will convert `tf_matrix` into a 2D NumPy array. Most of our subsequent analyses will be carried out on that NumPy array. However, periodically, we will compare array usage with CSR matrix usage. Doing so will allow us to more fully comprehend the similarities and differences between the two matrix representations. Listing 15.9 converts `tf_matrix` to NumPy by running `tf_matrix.toarray()` and then prints the converted result.

WARNING This conversion is very memory intensive, requiring almost 10 GB of memory. If your local machine has limited memory available, we suggest you execute this code in the cloud using Google Colaboratory (Colab), a free, cloud-based Jupyter Notebook environment with 12 GB freely available memory. Google provides a comprehensive introduction to using Colab that covers everything that you need to get started: <https://colab.research.google.com/notebooks/welcome.ipynb>.

Listing 15.9 Converting a CSR matrix to a NumPy array

```
tf_np_matrix = tf_matrix.toarray()
print(tf_np_matrix)
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

The printed matrix is a 2D NumPy array. All previewed matrix elements are zeros, confirming that the matrix is rather sparse. Each matrix element corresponds to the count of a word in the post. The matrix rows represent the post, while the columns represent individual words. Thus, the total column count equals our dataset's vocabulary size. We access that count using the `shape` attribute. This attribute is shared by both the CSR matrix and the NumPy array. Let's use `tf_np_matrix.shape` to output the vocabulary size.

Listing 15.10 Checking the vocabulary size

```
assert tf_np_matrix.shape == tf_matrix.shape
num_posts, vocabulary_size = tf_np_matrix.shape
print(f"Our collection of {num_posts} newsgroup posts contain a total of "
      f"{vocabulary_size} unique words")
```

```
Our collection of 11314 newsgroup posts contain a total of 114751 unique
words
```

Our data contains 114751 unique words. However, most posts contain only a few dozen of these words. We can measure the unique word count of a post at index `i` by counting the number of nonzero elements in row `tf_np_matrix[i]`. The easiest way to count these nonzero elements is with NumPy. The library allows us to obtain all nonzero indices of the vector at `tf_np_matrix[i]`. We simply need to input the vector into the `np.flatnonzero` function. Next, we count and output the nonzero indices of the car post in `newsgroups.data[0]`.

Listing 15.11 Counting the unique words in the car post

This is equivalent to running `np.nonzero(tf_vector)[0]`. The `np.nonzero` function generalizes the computation of nonzero indices across an x -dimensional array. It returns a tuple of length x , where each i th tuple element represents the nonzero indices along the i th dimension. Hence, given a 1D `tf_vector` array, the `np.nonzero` function returns a tuple of the form (`non_zero_indices`).

```
import numpy as np
tf_vector = tf_np_matrix[0]
non_zero_indices = np.flatnonzero(tf_vector) ←
num_unique_words = non_zero_indices.size
print(f"The newsgroup in row 0 contains {num_unique_words} unique words.")
print("The actual word counts map to the following column indices:\n")
print(non_zero_indices)
```

The newsgroup in row 0 contains 64 unique words.
The actual word-counts map to the following column indices:

```
[ 14106 15549 22088 23323 24398 27703 29357 30093 30629 32194
 32305 32499 37154 39275 42332 42333 43643 45089 45141 49871
 49881 50165 54442 55453 57577 58321 58842 60116 60326 64083
 65338 67137 67140 68931 69080 70570 72915 75280 78264 78701
 79055 79461 79534 82759 84398 87690 89161 92157 93304 95225
 96145 96432 100406 100827 100942 101084 101732 108644 109086 109254
109294 110106 112936 113262]
```

The first newsgroup post contains 64 unique words. What are these words? To find out, we need a mapping between TF vector indices and word values. That mapping can be generated by calling `vectorizer.get_feature_names()`, which returns a list of words that we'll call `words`. Each index i corresponds to the i th word in the list. Thus, running `[words[i] for i in non_zero_indices]` will return all unique words in our post.

NOTE We can also obtain these words by calling `vectorizer.inverse_transform(tf_vector)`. The `inverse_transform` method returns all words associated with an inputted TF vector.

Listing 15.12 Printing the unique words in the car post

```
words = vectorizer.get_feature_names()
unique_words = [words[i] for i in non_zero_indices]
print(unique_words)

['60s', '70s', 'addition', 'all', 'anyone', 'be', 'body', 'bricklin',
'bumper', 'called', 'can', 'car', 'could', 'day', 'door', 'doors',
'early', 'engine', 'enlighten', 'from', 'front', 'funky', 'have',
'history', 'if', 'in', 'info', 'is', 'it', 'know', 'late', 'looked',
'looking', 'made', 'mail', 'me', 'model', 'name', 'of', 'on', 'or',
'other', 'out', 'please', 'production', 'really', 'rest', 'saw',
'separate', 'small', 'specs', 'sports', 'tellme', 'the', 'there',
'this', 'to', 'was', 'were', 'whatever', 'where', 'wondering', 'years',
'you']
```

We've printed all the words in `newsgroups.data[0]`. Of course, not all of these words have equal mention counts—some occur more frequently than others. Perhaps these frequent words are more relevant to the topic of cars. Listing 15.13 prints the 10 most frequent words in the post, along with their associated counts. We represent this output as a Pandas table for visualization purposes.

Extracting nonzero elements of 1D NumPy arrays

- `non_zero_indices = np.flatnonzero(np_vector)`—Returns the nonzero indices in a 1D NumPy array
- `non_zero_vector = np_vector[non_zero_indices]`—Selects the nonzero elements of a 1D NumPy array (assuming `non_zero_indices` corresponds to nonzero indices of that array)

Listing 15.13 Printing the most frequent words in the car post

```
import pandas as pd
data = {'Word': unique_words,
        'Count': tf_vector[non_zero_indices]}

df = pd.DataFrame(data).sort_values('Count', ascending=False) ←
print(df[:10].to_string(index=False))

Word  Count
the      6
this     4
was      4
car      4
if       2
is       2
it       2
from     2
on       2
anyone   2
```

Sorts the Pandas table based on counts, from highest to lowest

Four of the 64 words in the post are mentioned at least four times. One of these words is *car*, which is not surprising given that the post appeared in a car discussion group. The other three words, however, have nothing to do with cars: *the*, *this*, and *was* are among the most common words in the English language. They don't provide a differentiating signal between the car post and a post with a different theme—instead, the common words are a source of noise and increase the likelihood that two unrelated documents will cluster together. NLP practitioners refer to such noisy words as *stop words* because they are blocked from appearing in the vectorized results. Stop words are generally deleted from the text before vectorization. That is why the `CountVectorizer` class has a built-in stop-word deletion option. Running `CountVectorizer(stop_words='english')` initializes a vectorizer that is primed for stop-word deletion. The vectorizer ignores all of the most common English words in the text.

Next, we reinitialize a stop-word-aware vectorizer. Then we rerun `fit_transform` to recompute the TF matrix. The number of word columns in that matrix will be less than our previously computed vocabulary size of 114,751. We also regenerate our words list: this time, common stop words such as *the*, *this*, *of*, and *it* will be missing.

Listing 15.14 Removing stop words during vectorization

```
vectorizer = CountVectorizer(stop_words='english')
tf_matrix = vectorizer.fit_transform(newsgroups.data)
assert tf_matrix.shape[1] < 114751 ← Checks to ensure that our vocabulary size has gotten smaller

words = vectorizer.get_feature_names()
for common_word in ['the', 'this', 'was', 'if', 'it', 'on']:
    assert common_word not in words ← Common stop words have been filtered out.
```

All stop words have been deleted from the recomputed `tf_matrix`. Now we can regenerate the 10 most frequent words in `newsgroups.data[0]`. Note that in the process, we recompute `tf_np_matrix`, `tf_vector`, `unique_words`, `non_zero_indices`, and `df`.

WARNING This regeneration is memory intensive, requiring 2.5 GB of memory.

Listing 15.15 Reprinting the top words after stop-word deletion

```
tf_np_matrix = tf_matrix.toarray()
tf_vector = tf_np_matrix[0]
non_zero_indices = np.flatnonzero(tf_vector)
unique_words = [words[index] for index in non_zero_indices]
data = {'Word': unique_words,
        'Count': tf_vector[non_zero_indices]}

df = pd.DataFrame(data).sort_values('Count', ascending=False)
print(f"After stop-word deletion, {df.shape[0]} unique words remain.")
print("The 10 most frequent words are:\n")
print(df[:10].to_string(index=False))
```

After stop-word deletion, 34 unique words remain.
The 10 most frequent words are:

Word	Count
car	4
60s	1
saw	1
looking	1
mail	1
model	1
production	1
really	1
rest	1
separate	1

After stop-word filtering, 34 words remain. Among them, *car* is the only word that is mentioned more than once. The other 33 words share a mention count of 1 and are treated equally by the vectorizer. However, it's worth noting that not all words are equal in their relevancy. Some words are more relevant to a car discussion than others: for instance, the word *model* refers to a car model (although of course it could also refer to a supermodel or a machine learning model). Meanwhile, the word *really* is more general; it doesn't refer to anything car related. The word is so irrelevant and common that it could almost be a stop word. In fact, some NLP practitioners keep *really* on their stop-word list—but others don't. Unfortunately, there is no consensus about which words are always useless and which aren't. However, all practitioners agree that a word becomes less useful if it's mentioned in too many texts. Thus, *really* is less relevant than *model* because the former is mentioned in more posts. Therefore, when ranking words by relevance, we should use both post frequency and count. If two words share an equal count, we should rank them by post frequency, instead.

Let's rerank our 34 words based on both post frequency and count. Then we'll explore how these rankings can be used to improve text vectorization.

Common scikit-learn CountVectorizer methods

- `vectorizer = CountVectorizer()`—Initializes a CountVectorizer object capable of vectorizing input texts based on their TF counts.
- `vectorizer = CountVectorizer(stopwords='english')`—Initializes an object capable of vectorizing input texts while filtering for common English words like *this* and *the*.
- `tf_matrix = vectorizer.fit_transform(texts)`—Executes TF vectorization on a list of input texts using the initialized vectorizer object, and returns a CSR matrix of term frequency values. Each matrix row *i* corresponds to `texts[i]`. Each matrix column *j* corresponds to the term frequency of word *j*.
- `vocabulary_list = vectorizer.get_feature_names()`—Returns the vocabulary list associated with the columns of a computed TF matrix. Each column *j* of the matrix corresponds to `vocabulary_list[j]`.

15.3 Ranking words by both post frequency and count

Each of the 34 words in `df.Word` appears in a certain fraction of newsgroup posts. In NLP, this fraction is referred to as the *document frequency* of a word. We hypothesize that the document frequencies can improve our word rankings. As scientists, we will now attempt to validate this hypothesis by exploring how the document frequencies relate to word importance. Initially, we'll limit our exploration to a single document. Later, we will generalize our insights to the other documents in the dataset.

NOTE Such open-ended explorations are common to data science. We start by exploring a small slice of data. By probing that small sample, we can hone our intuition about grander patterns in the dataset. Then we can test that intuition on a larger scale.

We now begin our exploration. Our immediate goal is to compute 34 document frequencies, to try to improve our word relevancy rankings. We can compute these frequencies using a series of NumPy matrix manipulations. First, we want to select the columns of `tf_np_matrix` that correspond to the 34 nonzero indices in the `non_zero_indices` array. We can obtain this submatrix by running `tf_np_matrix[:,non_zero_indices]`.

Listing 15.16 Filtering matrix columns with non_zero_indices

```
sub_matrix = tf_np_matrix[:,non_zero_indices]
print("Our sub-matrix corresponds to the 34 words within post 0. ")
    "The first row of the sub-matrix is:"
print(sub_matrix[0])
```

Accesses just those columns of the matrix that hold nonzero values in the first matrix row

Our sub-matrix corresponds to the 34 words within post 0. The first row of the sub-matrix is:

```
[1 1 1 1 1 1 1 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

The first row of sub_matrix corresponds to the 34 word counts in df. Together, all the matrix rows correspond to counts across all posts. However, we are not currently interested in exact word counts: we just want to know whether each word is present or absent from each post. So, we need to convert our counts into binary values. Basically, we require a binary matrix where element (i, j) equals 1 if word j is present in post i, and 0 otherwise. We can binarize the submatrix by importing binarize from sklearn.preprocessing. Then, running binarize(sub_matrix) will produce the necessary results.

Listing 15.17 Converting word counts to binary values

```
from sklearn.preprocessing import binarize
binary_matrix = binarize(sub_matrix)
print(binary_matrix)
```

[[1 1 1 ... 1 1 1]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 1 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]

The binarize function replaces all nonzero elements with ones in any x-dimensional array.

Now we need to add together the rows of our binary submatrix. Doing so will produce a vector of integer counts. Each i th vector element will equal the number of unique posts in which word i is present. To sum the rows of a 2D array, we simply need to pass axis=0 into the sum method of the array. Running binary_matrix.sum(axis=0) returns a vector of unique post counts.

NOTE A 2D NumPy array contains two axes: axis 0 corresponds to horizontal rows, and axis 1 corresponds to vertical columns. Thus, running binary_matrix.sum(axis=0) returns a vector of summed rows. Meanwhile, running binary_matrix.sum(axis=1) returns a vector of summed columns.

Listing 15.18 Summing matrix rows to obtain post counts

```
unique_post_mentions = binary_matrix.sum(axis=0)
print("This vector counts the unique posts in which each word is "
      "mentioned:\n {unique_post_mentions}")
```

This vector counts the unique posts in which each word is mentioned:
[18 21 202 314 4 26 802 536 842 154 67 348 184 25
 7 368 469 3093 238 268 780 901 292 95 1493 407 354 158
 574 95 98 2 295 1174]

Generally, running multi_dim_array.sum(axis=i) returns a vector of summed values across the i th axis of a multidimensional array.

We should note that the previous three procedures can be combined into a single line of code by running `binarize(tf_np_matrix[:,non_zero_indices]).sum(axis=0)`. Furthermore, substituting NumPy's `tf_np_matrix` with SciPy's `tf_matrix` will still produce the same post mention counts.

Listing 15.19 Computing post mention counts in a single line of code

```
np_post_mentions = binarize(tf_np_matrix[:,non_zero_indices]).sum(axis=0)
csr_post_mentions = binarize(tf_matrix[:,non_zero_indices]).sum(axis=0)
print(f'NumPy matrix-generated counts:\n{n {np_post_mentions}\n')
print(f'CSR matrix-generated counts:\n{n {csr_post_mentions}\n')

NumPy matrix-generated counts:
[[ 18   21  202  314    4   26   802   536   842   154   67   348   184   25
   7  368   469 3093  238   268   780   901   292   95 1493  407   354   158
 574   95   98     2  295 1174]]
```

CSR matrix-generated counts:

```
[[ 18   21  202  314    4   26   802   536   842   154   67   348   184   25
   7  368   469 3093  238   268   780   901   292   95 1493  407   354   158
 574   95   98     2  295 1174]]
```

The numbers in `np_post_mentions` and `csr_post_mentions` appear identical. However, `csr_post_mentions` contains an extra set of brackets because the aggregated sum of CSR matrix rows doesn't return a NumPy array; instead, it returns a special matrix object. In that object, the 1D vector is represented as a matrix with one row and `n` columns. To convert the matrix into a 1D NumPy array, we must run `np.asarray(csr_post_mentions)[0]`.

Methods for aggregating matrix rows

- `vector_of_sums = np_matrix.sum(axis=0)`—Sums the rows of a NumPy matrix. If `np_matrix` is a TF matrix, then `vector_of_sums[i]` equals the total mention count of word `i` in the dataset.
- `vector_of_sums = binarize(np_matrix).sum(axis=0)`—Converts a NumPy matrix into a binary matrix and then sums its rows. If `np_matrix` is a TF matrix, then `vector_of_sums[i]` equals the total count of texts in which word `i` is mentioned.
- `matrix_1D = binarize(csr_matrix).sum(axis=0)`—Converts a CSR matrix to binary and then sums its rows. The returned result is a special one-dimensional matrix object—it is not a NumPy vector. `matrix_1D` can be converted into a NumPy vector by running `np.asarray(matrix_1D) [0]`.

Based on the printed vector of post mention counts, we know that some words appear in thousands of posts. Other words appear in fewer than a dozen posts. Let's transform these counts into document frequencies and align the frequencies with `df.Word`. Then we'll output all the words that are mentioned in at least 10% of newsgroup posts. These words are likely to appear across the board in a variety of posts; thus, we hypothesize that the printed words will not be specific to a particular topic. If the hypothesis is correct, these words will not be very relevant.

Listing 15.20 Printing the words with the highest document frequency

```
document_frequencies = unique_post_mentions / dataset_size
data = {'Word': unique_words,
        'Count': tf_vector[non_zero_indices],
        'Document Frequency': document_frequencies}

df = pd.DataFrame(data)
df_common_words = df[df['Document Frequency'] >= .1]      ←
print(df_common_words.to_string(index=False))

    Word  Count  Document Frequency
    know     1       0.273378
  really     1       0.131960
    years     1       0.103765
```

We only choose words with a document frequency greater than 1/10.

As a reminder, the document frequency refers to all our posts. Meanwhile, the count refers to just the post at index 0.

Three of the 34 words have a document frequency greater than 0.1. As expected, these words are very general and not car specific. We thus can utilize document frequencies for ranking purposes. Let's rank our words by relevance in the following manner. First, we sort the words by count, from greatest to smallest. Then, all words with equal count are sorted by document frequency, from smallest to greatest. In Pandas, we can execute this dual-column sorting by running `df.sort_values(['Count', 'Document Frequency'], ascending=[False, True])`.

Listing 15.21 Ranking words by both count and document frequency

```
df_sorted = df.sort_values(['Count', 'Document Frequency'],
                           ascending=[False, True])
print(df_sorted[:10].to_string(index=False))

    Word  Count  Document Frequency
      car     4       0.047375
  tellme     1       0.000177
bricklin     1       0.000354
    funky     1       0.000619
     60s     1       0.001591
     70s     1       0.001856
enlighten     1       0.002210
    bumper     1       0.002298
     doors     1       0.005922
production     1       0.008397
```

Our sorting was successful. New car-related words, such as *bumper*, are now present in our list of top-ranked words. However, the actual sorting procedure was rather convoluted: it required us to sort two columns separately. Perhaps we can simplify the process by combining the word counts and document frequencies into a single score. How can we do this? One approach is to divide each word count by its associated document frequency. The resulting value will increase if either of the following is true:

- The word count goes up.
- The document frequency goes down.

Let's combine the word counts and the document frequencies into a single score. We start by computing $1 / \text{document_frequencies}$. Doing so produces an array of *inverse document frequencies* (IDFs). Next, we multiply `df.Count` by the IDF array to compute the combined score. We then add both the IDF values and our combined scores to our Pandas table. Finally, we sort on the combined score and output the top results.

Listing 15.22 Combining counts and frequencies into a single score

```
inverse_document_frequencies = 1 / document_frequencies
df['IDF'] = inverse_document_frequencies
df['Combined'] = df.Count * inverse_document_frequencies
df_sorted = df.sort_values('Combined', ascending=False)
print(df_sorted[:10].to_string(index=False))
```

Word	Count	Document Frequency	IDF	Combined
tellme	1	0.000177	5657.000000	5657.000000
bricklin	1	0.000354	2828.500000	2828.500000
funky	1	0.000619	1616.285714	1616.285714
60s	1	0.001591	628.555556	628.555556
70s	1	0.001856	538.761905	538.761905
enlighten	1	0.002210	452.560000	452.560000
bumper	1	0.002298	435.153846	435.153846
doors	1	0.005922	168.865672	168.865672
specs	1	0.008397	119.094737	119.094737
production	1	0.008397	119.094737	119.094737

Our new ranking failed! The word *car* no longer appears at the top of the list. What happened? Well, let's take a look at our table. There is a problem with the IDF values: some of them are huge! The printed IDF values range from approximately 100 to over 5,000. Meanwhile, our word-count range is very small: from 1 to 4. Thus, when we multiply word counts by IDF values, the IDF dominates, and the counts have no impact on the final results. We need to somehow make our IDF values smaller. What should we do?

Data scientists are commonly confronted with numeric values that are too large. One way to shrink the values is to apply a logarithmic function. For instance, running `np.log10(1000000)` returns 6. Essentially, a value of 1,000,000 is replaced by the count of zeros in that value.

Listing 15.23 Shrinking a large value using its logarithm

```
assert np.log10(1000000) == 6
```

Let's recompute our ranking score by running `df.Count * np.log10(df.IDF)`. The product of the counts and the shrunken IDF values should lead to a more reasonable ranking metric.

Listing 15.24 Adjusting the combined score using logarithms

```
df['Combined'] = df.Count * np.log10(df.IDF)
df_sorted = df.sort_values('Combined', ascending=False)
print(df_sorted[:10].to_string(index=False))
```

Word	Count	Document	Frequency	IDF	Combined
car	4		0.047375	21.108209	5.297806
tellme	1		0.000177	5657.000000	3.752586
bricklin	1		0.000354	2828.500000	3.451556
funky	1		0.000619	1616.285714	3.208518
60s	1		0.001591	628.555556	2.798344
70s	1		0.001856	538.761905	2.731397
enlighten	1		0.002210	452.560000	2.655676
bumper	1		0.002298	435.153846	2.638643
doors	1		0.005922	168.865672	2.227541
specs	1		0.008397	119.094737	2.075893

Our adjusted ranking score has yielded good results. The word *car* is once again present at the top of the ranked list. Also, *bumper* still appears among the top 10 ranked words. Meanwhile, *really* is missing from the list.

Our effective score is called the *term frequency-inverse document frequency* (TFIDF). The TFIDF can be computed by taking the product of the TF (word count) and the log of the IDF.

NOTE Mathematically, `np.log(1 / x)` is equal to `-np.log(x)`. Therefore, we can compute the TFIDF directly from the document frequencies. We simply need to run `df.Count * -np.log10(document_frequencies)`. Also be aware that other, less common formulations of the TFIDF exist in the literature. For instance, when dealing with large documents, some NLP practitioners compute the TFIDF as `np.log(df.Count + 1) * -np.log10(document_frequencies)`. This limits the influence of any very common word in a document.

The TFIDF is a simple but powerful metric for ranking words in a document. Of course, the metric is only relevant if that document is part of a larger document group. Otherwise, the computed TFIDF values all equal zero. The metric also loses its effectiveness when applied to small collections of similar texts. Nonetheless, for most real-world text datasets, the TFIDF produces good ranking results. And it has additional uses: it can be utilized to vectorize words in a document. The numeric content of `df.Combined` is essentially a vector produced by modifying the TF vector stored in `df.Count`. In this same manner, we can transform any TF vector into a TFIDF vector. We just need to multiply the TF vector by the log of inverse document frequencies.

Is there a benefit to transforming TF vectors into more complicated TFIDF vectors? Yes! In larger text datasets, TFIDF vectors provide a greater signal of textual similarity and divergence. For example, two texts that are both discussing cars are more likely to cluster together if their irrelevant vector elements are penalized. Thus, penalizing common words using the IDF improves the clustering of large text collections.

NOTE This isn't necessarily true of smaller datasets, where the number of documents is low and the document frequency is high. Consequently, the IDF may be too small to meaningfully improve the clustering results.

We therefore stand to gain by transforming our TF matrix into a TFIDF matrix. We can easily execute this transformation using custom code. However, it's more convenient to compute the TFIDF matrix with scikit-learn's built-in `TfidfVectorizer` class.

15.3.1 Computing TFIDF vectors with scikit-learn

That `TfidfVectorizer` class is nearly identical to `CountVectorizer`, except that it takes IDF into account during the vectorization process. Next, we import `TfidfVectorizer` from `sklearn.feature_extraction.text` and initialize the class by running `TfidfVectorizer(stop_words='english')`. The constructed `tfidf_vectorizer` object is parameterized to ignore all stop words. Subsequently, executing `tfidf_vectorizer.fit_transform(newsgroups.data)` returns a matrix of vectorized TFIDF values. The matrix shape is identical to `tf_matrix.shape`.

Listing 15.25 Computing a TFIDF matrix with scikit-learn

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf_vectorizer.fit_transform(newsgroups.data)
assert tfidf_matrix.shape == tf_matrix.shape
```

Our `tfidf_vectorizer` has learned the same vocabulary as the simpler TF vectorizer. In fact, the indices of words in `tfidf_matrix` are identical to those of `tf_matrix`. We can confirm this by calling `tfidf_vectorizer.get_feature_names()`. The method call returns an ordered list of words identical to our previously computed words list.

Listing 15.26 Confirming the preservation of vectorized word indices

```
assert tfidf_vectorizer.get_feature_names() == words
```

Since word order is preserved, we should expect the nonzero indices of `tfidf_matrix[0]` to equal our previously computed `non_zero_indices` array. We'll confirm after converting `tfidf_matrix` from a CSR data structure to a NumPy array.

Listing 15.27 Confirming the preservation of nonzero indices

```
tfidf_np_matrix = tfidf_matrix.toarray()
tfidf_vector = tfidf_np_matrix[0]
tfidf_non_zero_indices = np.flatnonzero(tfidf_vector)
assert np.array_equal(tfidf_non_zero_indices,
                     non_zero_indices)
```

The nonzero indices of `tf_vector` and `tfidf_vector` are identical. We thus can add the TFIDF vector as a column in our existing `df` table. Adding a TFIDF column will allow us to compare scikit-learn's output with our manually computed score.

Listing 15.28 Adding a TFIDF vector to the existing Pandas table

```
df['TFIDF'] = tfidf_vector[non_zero_indices]
```

Sorting by `df.TFIDF` should produce a relevance ranking that is consistent with our previous observations. Let's verify that both `df.TFIDF` and `df.Combined` produce the same word rankings after sorting.

Listing 15.29 Sorting words by `df.TFIDF`

```
df_sorted_old = df.sort_values('Combined', ascending=False)
df_sorted_new = df.sort_values('TFIDF', ascending=False)
assert np.array_equal(df_sorted_old['Word'].values,
                      df_sorted_new['Word'].values)
print(df_sorted_new[:10].to_string(index=False))
```

Word	Count	Document Frequency	IDF	Combined	TFIDF
car	4	0.047375	21.108209	5.297806	0.459552
tellme	1	0.000177	5657.000000	3.752586	0.262118
bricklin	1	0.000354	2828.500000	3.451556	0.247619
funky	1	0.000619	1616.285714	3.208518	0.234280
60s	1	0.001591	628.555556	2.798344	0.209729
70s	1	0.001856	538.761905	2.731397	0.205568
enlighten	1	0.002210	452.560000	2.655676	0.200827
bumper	1	0.002298	435.153846	2.638643	0.199756
doors	1	0.005922	168.865672	2.227541	0.173540
specs	1	0.008397	119.094737	2.075893	0.163752

Our word rankings have remained unchanged. However, the values of the `TFIDF` and `Combined` columns are not identical. Our top 10 manually computed `Combined` values are all greater than 1, but all of scikit-learn's `TFIDF` values are less than 1. Why is this the case?

As it turns out, scikit-learn automatically normalizes its `TFIDF` vector results. The magnitude of `df.TFIDF` has been modified to equal 1. We can confirm by calling `norm(df.TFIDF.values)`.

NOTE To turn off normalization, we must pass `norm=None` into the vectorizer's initialization function. Running `TfidfVectorizer(norm=None, stop_words='english')` returns a vectorizer in which normalization has been deactivated.

Listing 15.30 Confirming that our `TFIDF` vector is normalized

```
from numpy.linalg import norm
assert norm(df.TFIDF.values) == 1
```

Why would scikit-learn automatically normalize the vectors? For our own benefit! As discussed in section 13, it's easier to compute text vector similarity when all vector magnitudes equal 1. Consequently, our normalized `TFIDF` matrix is primed for similarity analysis.

Common scikit-learn TfidfVectorizer methods

- `tfidf_vectorizer = TfidfVectorizer(stopwords='english')`—Initializes a TfidfVectorizer object capable of vectorizing input texts based on their TFIDF values. The object is preset to filter common English stop words.
- `tfidf_matrix = tfidf_vectorizer.fit_transform(texts)`—Executes TFIDF vectorization on a list of input texts using the initialized vectorizer object and returns a CSR matrix of normalized TFIDF values. Each row of the matrix is automatically normalized, for easier similarity computation.
- `vocabulary_list = tfidf_vectorizer.get_feature_names()`—Returns the vocabulary list associated with the columns of a computed TFIDF matrix. Each column j of the matrix corresponds to `vocabulary_list[j]`.

15.4 Computing similarities across large document datasets

Let's answer a simple question: which of our newsgroup posts is most similar to `newsgroups.posts[0]`? We can get the answer by computing all the cosine similarities between `tfidf_np_matrix` and `tf_np_matrix[0]`. As discussed in section 13, these similarities can be obtained by taking the product of `tfidf_np_matrix` and `tfidf_matrix[0]`. The simple multiplication between the matrix and the vector is sufficient because all rows in the matrix have a magnitude of 1.

Listing 15.31 Computing similarities to a single newsgroup post

```
cosine_similarities = tfidf_np_matrix @ tfidf_np_matrix[0]
print(cosine_similarities)

[1.          0.00834093  0.04448717 ... 0.           0.00270615  0.01968562]
```

The matrix-vector product takes a few seconds to complete. Its output is a vector of cosine similarities: each i th index of the vector corresponds to the cosine similarity between `newsgroups.data[0]` and `newsgroups.data[i]`. From the printout, we can see that `cosine_similarities[0]` is equal to 1.0. This is not surprising since `newsgroups_data[0]` will have a perfect similarity with itself. What is the next-highest similarity in the vector? We can find out by calling `np.argsort(cosine_similarities)[-2]`. The `argsort` call sorts the array indices by their ascending values. So, the second-to-last index will correspond to the post with the second-highest similarity.

NOTE We are assuming that no other post exists with a perfect similarity of 1. Also, note that we can achieve the same result by calling `np.argmax(cosine_similarities[1:]) + 1`, although this approach only works for posts at index 0.

We now extract that index and print its corresponding similarity. We also print the corresponding text to confirm its overlap with the car post stored in `newsgroups.data[0]`.

Listing 15.32 Finding the most similar newsgroup post

```
most_similar_index = np.argsort(cosine_similarities)[-2]
similarity = cosine_similarities[most_similar_index]
most_similar_post = newsgroups.data[most_similar_index]
print(f"The following post has a cosine similarity of {similarity:.2f} "
      "with newsgroups.data[0]:\n")
print(most_similar_post)

The following post has a cosine similarity of 0.64 with newsgroups.data[0]:  

In article <1993Apr20.174246.14375@wam.umd.edu> lerxst@wam.umd.edu
(where's my
thing) writes:
>
> I was wondering if anyone out there could enlighten me on this car I saw
> the other day. It was a 2-door sports car, looked to be from the late
> 60s/ early 70s. It was called a Bricklin. The doors were really small. In
addition,
> the front bumper was separate from the rest of the body. This is
> all I know. If anyone can tellme a model name, engine specs, years
> of production, where this car is made, history, or whatever info you
> have on this funky looking car, please e-mail.

Bricklins were manufactured in the 70s with engines from Ford. They are
rather odd looking with the encased front bumper. There aren't a lot of
them around, but Hemmings (Motor News) ususally has ten or so listed.
Basically, they are a performance Ford with new styling slapped on top.

> ----- brought to you by your neighborhood Lerxst ----

Rush fan?
```

The printed text is a reply to the car post at index 0. The reply includes the original post, which is a question about a certain car brand. We see a detailed answer to the question near the very bottom of the reply. Due to textual overlap, both the original post and the reply are very similar to each other. Their cosine similarity is 0.64, which does not seem like a large number. However, in extensive text collections, a cosine similarity greater than 0.6 is a good indicator of overlapping content.

NOTE As discussed in section 13, the cosine similarity can easily be converted into the Tanimoto similarity, which has a deeper theoretical basis for text overlap. We can convert `cosine_similarities` into Tanimoto similarities by running `cosine_similarities / (2 - cosine_similarities)`. However, that conversion will not change our final results. Choosing the top index of the Tanimoto array will still return the same posted reply. Thus, for simplicity's sake, we focus on the cosine similarity during our next few text-comparison examples.

Thus far, we've only analyzed the car post at index 0. Let's extend our analysis to another post. We'll pick a newsgroup post at random, choose its most similar neighbor,

and then output both posts, along with their cosine similarity. To make this exercise more interesting, we'll first compute a matrix of all-by-all cosine similarities. We'll then use the matrix to select our random pair of similar posts.

NOTE Why are we computing a matrix of all-by-all similarities? Mostly, it's to practice what we've learned in the previous section. However, having access to that matrix does confer certain benefits. Suppose we wish to increase our network of neighboring posts from 2 to 10. We also wish to include the neighbor of every neighbor (similar to our derivation of DBSCAN in section 10). Under such circumstances, it is much more efficient to compute all text similarities in advance.

How do we compute the matrix of all-by-all cosine similarities? The naive approach is to multiply `tfidf_np_matrix` with its transpose. However, for reasons discussed in section 13, this matrix multiplication is not computationally efficient. Our TFIDF matrix has over 100,000 columns. We need to reduce the matrix size before executing the multiplication. In the previous section, we learned how to reduce the column count using scikit-learn's `TruncatedSVD` class. The class is able to shrink a matrix to a specified number of columns. The reduced column count is determined by the `n_components` parameter. According to scikit-learn's documentation, an `n_components` value of 100 is recommended for processing text data.

NOTE Scikit-learn's documentation occasionally provides useful parameters for common algorithm applications. For instance, take a look at the `TruncatedSVD` documentation at <http://mng.bz/PXP9>. According to that page, "Truncated SVD works on term count/tf-idf matrices as returned by the vectorizers in `sklearn.feature_extraction.text`. In that context, it is known as latent semantic analysis (LSA)." Further down, the documentation describes the `n_components` parameter like this: "Desired dimensionality of output data. Must be strictly less than the number of features. For LSA, a value of 100 is recommended."

Most NLP practitioners agree that passing `n_components=100` reduces a TFIDF matrix to an efficient size while maintaining useful column information. Next, we'll follow this recommendation by running `TruncatedSVD(n_components=100).fit_transform(tfidf_matrix)`. The method call will return a 100-column `shrunk_matrix` result that will be a 2D NumPy array even if we pass the SciPy-based `tfidf_matrix` as our input.

Listing 15.33 Dimensionally reducing `tfidf_matrix` using SVD

```
np.random.seed(0)    ←
from sklearn.decomposition import TruncatedSVD

shrunk_matrix = TruncatedSVD(n_components=100).fit_transform(tfidf_matrix)
print(f"We've dimensionally reduced a {tfidf_matrix.shape[1]}-column "
      f"{type(tfidf_matrix)} matrix.")
```

The final SVD output depends on the orientation of the computed eigenvectors. As we saw in the previous section, that orientation is determined randomly. Thus, we run `np.random.seed(0)` to ensure consistent results.

```
print(f"Our output is a {shrunk_matrix.shape[1]}-column "
      f"{{type(shrunk_matrix)}} matrix.")
```

We've dimensionally reduced a 114441-column
<class 'scipy.sparse.csr.csr_matrix'> matrix.
Our output is a 100-column <class 'numpy.ndarray'> matrix.

Our shrunk matrix contains just 100 columns. We can now efficiently compute the cosine similarities by running `shrunk_matrix @ shrunk_matrix.T`. However, first we need to confirm that the matrix rows remain normalized. Let's check the magnitude of `shrunk_matrix[0]`.

Listing 15.34 Checking the magnitude of `shrunk_matrix[0]`

```
magnitude = norm(shrunk_matrix[0])
print(f"The magnitude of the first row is {magnitude:.2f}")
```

The magnitude of the first row is 0.49

The magnitude of the row is less than 1. Scikit-learn's SVD output has not been automatically normalized. We need to manually normalize the matrix before computing the similarities. Scikit-learn's built-in `normalize` function will assist us in that process. We import `normalize` from `sklearn.preprocessing` and then run `normalize(shrunk_matrix)`. The magnitude of the rows in the resulting normalized matrix will subsequently equal 1.

Listing 15.35 Normalizing the SVD output

```
from sklearn.preprocessing import normalize
shrunk_norm_matrix = normalize(shrunk_matrix)
magnitude = norm(shrunk_norm_matrix[0])
print(f"The magnitude of the first row is {magnitude:.2f}")
```

The magnitude of the first row is 1.00

The shrunken matrix has been normalized. Now, running `shrunk_norm_matrix @ shrunk_norm_matrix.T` should produce a matrix of all-by-all cosine similarities.

Listing 15.36 Computing all-by-all cosine similarities

```
cosine_similarity_matrix = shrunk_norm_matrix @ shrunk_norm_matrix.T
```

We have our similarity matrix. Let's use it to choose a random pair of very similar texts. We start by randomly selecting a post at some `index1`. We next select an index of `cosine_similarities[index1]` that has the second-highest cosine similarity. Then, we print both the indices and their similarity before displaying the texts.

Listing 15.37 Choosing a random pair of similar posts

```
np.random.seed(1)
index1 = np.random.randint(dataset_size)
```

```
index2 = np.argsort(cosine_similarity_matrix[index1]) [-2]
similarity = cosine_similarity_matrix[index1][index2]
print(f"The posts at indices {index1} and {index2} share a cosine "
      f"similarity of {similarity:.2f}")
```

The posts at indices 235 and 7805 share a cosine similarity of 0.91

Listing 15.38 Printing a randomly chosen post

```
print(newsgroups.data[index2].replace('\n\n', '\n'))
```

Hello,
 Who can tell me Where can I find the PD or ShareWare
 Which can CAPTURE windows 3.1's output of printer manager?
 I want to capture the output of HP Laser Jet III.
 Though the PostScript can setup to print to file, but HP can't.
 I try DOS's redirect program, but they can't work in Windows 3.1
 Thankx for any help....
--
 Internet Address: u7911093@cc.nctu.edu.tw
 English Name: Erik Wang
 Chinese Name: Wang Jyh-Shyang

This post contains blank lines. We filter out these lines to conserve space.

Once again, the printed post is a question. It's safe to assume that the post at index1 is an answer to that question.

Listing 15.39 Printing the most similar post response

```
print(newsgroups.data[index1].replace('\n\n', '\n'))
```

u7911093@cc.nctu.edu.tw ("By SWH) writes:
>Who can tell me which program (PD or ShareWare) can redirect windows 3.1's
>output of printer manager to file?
> I want to capture HP Laser Jet III's print output.
> Though PostScript can setup print to file, but HP can't.
> I use DOS's redirect program, but they can't work in windows.
> Thankx for any help...
>--
> Internet Address: u7911093@cc.nctu.edu.tw
> English Name: Erik Wang
> Chinese Name: Wang Jyh-Shyang
> National Chiao-Tung University, Taiwan, R.O.C.
Try setting up another HPIII printer but when choosing what port to connect it to choose FILE instead of like :LPT1. This will prompt you for a file name everytime you print with that "HPIII on FILE" printer. Good Luck.

Thus far, we have examined two pairs of similar posts. Each post pair was composed of a question and a reply, where the question was included in the reply. Such boring pairs of overlapping texts are trivial to extract. Let's challenge ourselves to find something more interesting. We'll search for clusters of similar texts where posts in a cluster share some text without perfectly overlapping.

15.5 Clustering texts by topic

In section 10, we introduced two clustering algorithms: K-means and DBSCAN. K-means can only cluster on Euclidean distance. Conversely, DBSCAN can cluster based on any distance metric. One possible metric is cosine distance, which equals 1 minus cosine similarity.

NOTE Why use cosine distance instead of cosine similarity? Well, all clustering algorithms assume that two identical data points share a distance of 0. Meanwhile, the cosine similarity equals 0 if two data points have nothing in common. It also equals 1 when two data points are perfectly identical. We can fix this discrepancy by running `1 - cosine_similarity_matrix`, thus converting our result to cosine distance. After the conversion, two identical texts will share a cosine distance of 0.

Cosine distance is commonly used in conjunction with DBSCAN. That is why scikit-learn's DBSCAN implementation permits us to specify cosine distance directly during object initialization. We simply need to pass `metric='cosine'` into the class constructor. Doing so will initialize a `cluster_model` object that's set to cluster based on cosine distance.

NOTE Scikit-learn's DBSCAN implementation computes cosine distance by first recomputing `cosine_similarity_matrix`. Alternatively, we can avoid the recomputation by passing `metric='precomputed'` into the constructor. Doing so initializes a `cluster_model` object that's set to cluster on a matrix of precomputed distances. Next, running `cluster_model.fit_transform(1 - cosine_similarity_matrix)` should theoretically return the clustering results. However, practically speaking, negative values in the distance matrix (which can arise from floating-point errors) could cause issues during clustering. All negative values in the distance matrix must be replaced with zero before clustering. This operation would need to be run manually in NumPy by executing `x[x < 0] = 0`, where `x = 1 - cosine_similarity_matrix`.

Let's cluster `shrunk_matrix` with DBSCAN based on cosine distance. During clustering, we will make the following reasonable assumptions:

- Two newsgroup posts fall in a cluster if they share a cosine similarity of at least 0.6 (which corresponds to a cosine distance of no greater than 0.4).
- A cluster contains at least 50 newsgroup posts.

Based on these assumptions, the algorithm's `eps` and `min_samples` parameters should equal 0.4 and 50, respectively. Thus, we initialize DBSCAN by running `DBSCAN(eps=0.4, min_samples=50, metric='cosine')`. Then we use the initialized `cluster_model` object to cluster `shrunk_matrix`.

Listing 15.40 Clustering newsgroup posts with DBSCAN

```
from sklearn.cluster import DBSCAN
cluster_model = DBSCAN(eps=0.4, min_samples=50, metric='cosine')
clusters = cluster_model.fit_predict(shrunk_matrix)
```

We've generated an array of clusters. Let's quickly estimate the clustering quality. We already know that the newsgroups dataset covers 20 newsgroup categories. Some of the category names are very similar to each other; other topics are incredibly broad. Thus, it's reasonable to assume that the dataset covers 10 to 25 truly diverging topics. Consequently, we can expect our clusters array to contain somewhere between 10 and 25 clusters—otherwise, there's something wrong with our input clustering parameters. We now count the number of clusters.

Listing 15.41 Counting the number of DBSCAN clusters

```
cluster_count = clusters.max() + 1
print(f"We've generated {cluster_count} DBSCAN clusters")

We've generated 3 DBSCAN clusters
```

We've generated just three clusters, which is way lower than our expected cluster count. Clearly, our DBSCAN parameters were wrong. Is there some algorithmic method to adjust these parameters accordingly? Or are there perhaps well-known DBSCAN settings in the literature that yield acceptable text clusters? Sadly, no. As it happens, DBSCAN clustering of text is highly sensitive to the inputted document data. DBSCAN parameters for clustering specific types of texts (such as newsgroup posts) are unlikely to transfer well to other document categories (such as news articles or emails). Consequently, unlike with SVD, the DBSCAN algorithm lacks consistent NLP parameters. This does not mean DBSCAN cannot be applied to our text data, but the appropriate `eps` and `min_samples` inputs must be determined by trial and error. Unfortunately, DBSCAN lacks a well-established algorithm for optimizing these two crucial parameters.

K-means, on the other hand, takes as input a single K parameter. We can estimate K using the elbow plot technique, which we introduced in section 10. However, the K-means algorithm can only cluster based on Euclidean distance: it cannot process cosine distance. Is this a problem? Not necessarily. As it happens, we're in luck! All rows in `shrunk_norm_matrix` are normalized unit vectors. In section 13, we showed how the Euclidean distance of two normalized vectors v_1 and v_2 equals $(2 - 2 * v_1 @ v_2)^{0.5}$. Furthermore, the cosine distance between the vectors equals $1 - v_1 @ v_2$. With basic algebra, we can easily show that the Euclidean distance of two normalized vectors is proportional to the square root of the cosine distance. The two distance metrics are very closely related! This relationship provides us with mathematical justification for clustering `shrunk_norm_matrix` using K-means.

WARNING If two vectors are normalized, their Euclidean distance is an adequate substitute for cosine similarity. However, this is not the case for non-normalized vectors. Thus, we should never apply K-means to text-derived matrices if these matrices have not been normalized.

Research has shown that K-means clustering provides a reasonable segmentation of text data. This may seem confusing, since in previous sections, DBSCAN gave us superior results. Regrettably, in data science, the right choice of algorithm varies by domain. Rarely can one algorithm solve every type of problem. By analogy, not every task requires a hammer—sometimes we need a screwdriver or a wrench. Data scientists must remain flexible when choosing the appropriate tool for a given task.

NOTE Sometimes we may not know which algorithm to use on a given problem. When we are stuck, it helps to read known solutions online. The scikit-learn website in particular provides insightful solutions to common problems. For instance, the scikit-learn site offers example code for clustering text at <http://mng.bz/wQ9q>. Notably, the documented code illustrates how K-means can cluster text vectors (after SVD processing). The documentation also specifies that the vectors must be normalized “for better results.”

Let’s utilize K-means to cluster `shrunk_norm_matrix` into K different groups. We first need to assign a value for K . Supposedly, our texts belong to 20 different newsgroup categories. But as mentioned earlier, the actual cluster count may not equal 20. We want to estimate the true value of K by generating an elbow plot. To this end, we’ll execute K-means across K values of 1 through 60 and then plot the inertia results.

However, we face a problem. Our dataset is large, containing over 10,000 points. The scikit-learn `KMeans` implementation will take a second or two to cluster the data. That lag time is acceptable for a single clustering run, but it’s not acceptable for 60 different runs, where the execution time may add up to multiple minutes. How can we speed up the K-means running time? Well, one approach is to sample randomly from our hefty dataset. We can choose 1,000 random newsgroup posts during the K-means centroid calculation, then select another random 1,000 posts, and then update the cluster centers based on post content. In this manner, we can iteratively estimate the centers through sampling. At no point will we need to analyze the full dataset all at once. This modified version of the K-means algorithm is known as *mini-batch K-means*. Scikit-learn offers a mini-batch implementation with its `MiniBatchKMeans` class. `MiniBatchKMeans` is nearly identical in its methods to the standard `KMeans` class. Next, we import both implementations and compare their running times.

NOTE We should emphasize that even with `MiniBatchKMeans`, efficient computation time is possible only because we have dimensionally reduced our data.

Listing 15.42 Comparing KMeans to MiniBatchKMeans

```

np.random.seed(0)
import time
from sklearn.cluster import KMeans, MiniBatchKMeans

k=20
times = []
for KMeans_class in [KMeans, MiniBatchKMeans]: ←
    start_time = time.time() ←
    KMeans_class(k).fit(shrunk_norm_matrix) ←
    times.append(time.time() - start_time) ←
running_time_ratio = times[0] / times[1]
print(f"Mini Batch K-means ran {running_time_ratio:.2f} times faster "
      "than regular K-means")

```

Computes the running time of each clustering algorithm implementation

Running time.time() returns the current time, in seconds.

Mini Batch K-means ran 10.53 times faster than regular K-means

MiniBatchKMeans runs approximately 10 times faster than regular KMeans. That decrease in running time carries a minor cost: it has been shown that MiniBatchKMeans produces clusters of slightly lower quality than KMeans. However, our immediate concern isn't cluster quality; rather, we are interested in estimating K using an elbow plot across range(1, 61). The speedy MiniBatchKMeans implementation should serve us just fine as an estimation tool.

We now generate the plot using mini-batch K-means. We also add grid lines to the plot, to better isolate potential elbow coordinates. As seen in section 10, we can visualize such grid lines by calling plt.grid(True). Finally, we want to compare the elbow with the official newsgroup category count. For this purpose, we will plot a vertical line at a K value of 20 (figure 15.1).

Listing 15.43 Plotting an elbow curve using MiniBatchKMeans

```

np.random.seed(0)
import matplotlib.pyplot as plt

k_values = range(1, 61)
inertia_values = [MiniBatchKMeans(k).fit(shrunk_norm_matrix).inertia_
                  for k in k_values]
plt.plot(k_values, inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.axvline(20, c='k')
plt.grid(True)
plt.show()

```

Our plotted curve decreases smoothly. The precise location of a bent elbow-shaped transition is difficult to spot. We do see that the curve is noticeably steeper when K is less than 20. Somewhere after 20 clusters, the curve begins to flatten out, but there is

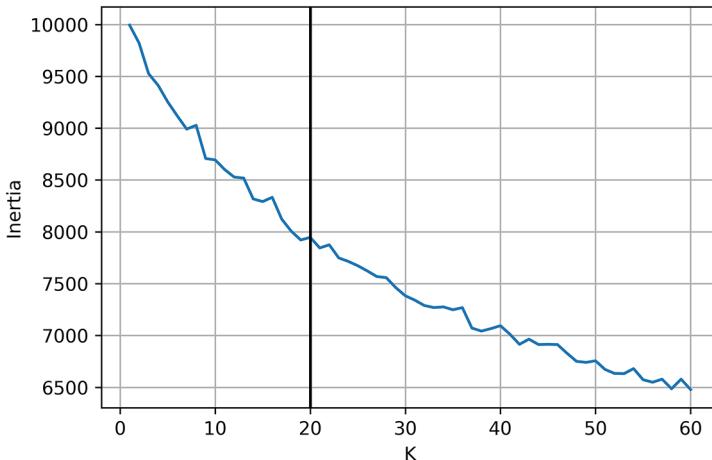


Figure 15.1 An elbow plot generated using mini-batch K-means across K values ranging from 1 to 61. The precise location of an elbow is difficult to determine. However, the plotted curve is noticeably steeper before a K of 20. Also, it begins to flatten after a K of 20. We thus infer that a value of approximately 20 is an appropriate input for K .

no singular location at which the elbow suddenly bends. The dataset lacks a perfect K at which the texts fall into natural clusters. Why? For one thing, real-world text is messy and nuanced. Categorical boundaries are not always obvious. For instance, we can engage in a conversation about technology or a conversation about politics. In addition, we can openly discuss how politics is influenced by technology. Seemingly distinct discussion topics can meld together, forming novel topics of their own. Due to such complexities, there rarely exists a single, smooth transition between text clusters. Consequently, figuring out an ideal K is hard. But we can make certain useful inferences: based on the elbow plot, we can infer that 20 is a reasonable estimate of the K parameter. Yes, the curve is fuzzy, and perhaps an input of 18 or 22 would also do. However, we need to start somewhere, and a K of 20 makes more sense than a K of 3 or 50. Our solution isn't perfect, but it's feasible. Sometimes, when we deal with real-world data, a feasible solution is the best that we can expect.

NOTE If you are uncomfortable choosing the elbow qualitatively by staring at the plot, consider using the external Yellowbrick library. The library contains a `KElbowVisualizer` class (<http://mng.bz/7IV9>) that uses both Matplotlib and scikit-learn's mini-batch K-means implementation to highlight the elbow location in an automated manner. If we initialize `KElbowVisualizer` and apply it to our data, the corresponding object returns a K of 23. Additionally, Yellowbrick offers more powerful K -selection methodologies, such as the silhouette score (which we alluded to in section 10.) The library can be installed by running `pip install yellowbrick`.

We will now divide `shrunk_norm_matrix` into 20 clusters. We run the origin `KMeans` implementation for maximum accuracy and then store the text indices and cluster IDs in a Pandas table for easier analysis.

Listing 15.44 Clustering newsgroup posts into 20 clusters

```
np.random.seed(0)
cluster_model = KMeans(n_clusters=20)
clusters = cluster_model.fit_predict(shrunk_norm_matrix)
df = pd.DataFrame({'Index': range(clusters.size), 'Cluster': clusters})
```

We have clustered our texts and are ready to explore the cluster contents. However, first we must briefly discuss one important consequence of executing K-means on large matrix inputs: the resulting clusters may vary slightly across different computers, even if we run `np.random.seed(0)`. This divergence is driven by how different machines round floating-point numbers. Some computers round small numbers up, while others round these numbers down. Normally, these differences are not noticeable. Unfortunately, in a 10,000-by-100 element matrix, small differences can impact the clustering results. K-means is not deterministic, as we've discussed in section 10—it can converge in multiple ways to multiple sets of equally valid clusters. Thus, your locally run text clusters may differ from outputs in this book, but your observations and conclusions should be similar.

With this in mind, let's proceed with the analysis. We begin by analyzing a single cluster. Later, we analyze all the clusters simultaneously.

15.5.1 Exploring a single text cluster

One of our 20 clusters contains the car post at index 0 of `newsgroups.data`. Let's isolate and count the number of texts that group together with that car-themed message.

Listing 15.45 Isolating the car cluster

```
df_car = df[df.Cluster == clusters[0]]
cluster_size = df_car.shape[0]
print(f"{cluster_size} posts cluster together with the car-themed post "
      "at index 0")
```

393 posts cluster together with the car-themed post at index 0

WARNING As we've just discussed, the contents of the cluster may differ slightly on your local machine. The total cluster size may minimally diverge from 393. If this happens, the subsequent sequence of code listings may produce different results. Regardless of these differences, you should still be able to draw similar conclusions from your locally generated outputs.

393 posts cluster with the car-themed texts at index 0. Presumably, these posts are also about cars. If so, then a randomly chosen post should mention an automobile. Let's verify if this is the case.

Listing 15.46 Printing a random post in the car cluster

```
np.random.seed(1)
def get_post_category(index):
    target_index = newsgroups.target[index]
    return newsgroups.target_names[target_index]

random_index = np.random.choice(df_car.Index.values)
post_category = get_post_category(random_index)

print(f"This post appeared in the {post_category} discussion group:\n")
print(newsgroups.data[random_index].replace('\n\n', '\n'))
```

This post appeared in the *rec.autos* discussion group:

My wife and I looked at, and drove one last fall. This was a 1992 model. It was WAYYYYYYYYY underpowered. I could not imagine driving it in the mountains here in Colorado at anything approaching highway speeds. I have read that the new 1993 models have a newer, improved hp engine. I'm quite serious that I laughed in the salesman face when he said "once it's broken in it will feel more powerful". I had been used to driving a Jeep 4.0L 190hp engine. I believe the 92's Land Cruisers (Land Yachts) were 3.0L, the same as the 4Runner, which is also underpowered (in my own personal opinion). They are big cars, very roomy, but nothing spectacular.

Returns the post category of the newsgroup post found at index "index". We will reuse the function elsewhere in the section.

The random post discusses a model of Jeep. It was posted in the *rec.autos* discussion group. How many of the nearly 400 posts in the cluster belong to *rec.autos*? Let's find out.

Listing 15.47 Checking cluster membership to *rec.autos*

```
rec_autos_count = 0
for index in df_car.Index.values:
    if get_post_category(index) == 'rec.autos':
        rec_autos_count += 1

rec_autos_percent = 100 * rec_autos_count / cluster_size
print(f"{rec_autos_percent:.2f}% of posts within the cluster appeared "
      "in the rec.autos discussion group")

84.73% of posts within the cluster appeared in the rec.autos discussion
group
```

In this cluster, 84% of the posts appeared in *rec.autos*. The cluster is thus dominated by that car discussion group. What about the remaining 16% of the clustered posts? Did they fall erroneously into the cluster? Or are they relevant to the topic of automobiles? We'll soon find out. Let's isolate the indices of posts in *df_car* that do not belong to *rec.autos*. Then we'll choose a random index and print the associated post.

Listing 15.48 Examining a post that did not appear in rec.autos

```
np.random.seed(1)
not_autos_indices = [index for index in df_car.Index.values
                     if get_post_category(index) != 'rec.autos']

random_index = np.random.choice(not_autos_indices)
post_category = get_post_category(random_index)

print(f"This post appeared in the {post_category} discussion group:\n")
print(newsgroups.data[random_index].replace('\n\n', '\n'))
```

This post appeared in the sci.electronics discussion group:

```
>The father of a friend of mine is a police officer in West Virginia. Not
>only is his word as a skilled observer good in court, but his skill as an
>observer has been tested to be more accurate than the radar gun in some
>cases . . . No foolin! He can guess a car's speed to within 2-3mph just
>by watching it blow by - whether he's standing still or moving too! (Yes,
1) How was this testing done, and how many times? (Calibrated
speedometer?)
2) It's not the "some cases" that worry me, it's the "other cases" :-)
```

They are big cars, very roomy, but nothing spectacular.

The random post appeared in an electronics discussion group. The post describes the use of radar to measure car speed. Thematically, it's about automobiles, so it appears to have clustered correctly. What about the other 60 or so posts represented by the not_autos_indices list? How do we evaluate their relevance? We could read each post, one by one, but that is not a scalable solution. Instead, we can aggregate their content by displaying the top-ranking words across all posts. We rank each word by summing its TFIDF across each index in not_autos_indices. Then we'll sort the words based on their aggregated TFIDF. Printing out the top 10 words will help us determine if our content is relevant to cars.

Next, we define a rank_words_by_tfidf function. The function takes as input a list of indices and ranks the words across these indices using the approach described previously. The ranked words are stored in a Pandas table for easier display. The summed TFIDF values used to rank the words are also stored in that table. Once our function is defined, we will run rank_words_by_tfidf(not_autos_indices) and output the top 10 ranked results.

NOTE Given an indices array, we want to aggregate the rows of tfidf_np_matrix[indices]. As discussed earlier, we can sum over rows by running tfidf_np_matrix[indices].sum(axis=0). Additionally, we can generate that sum by running tfidf_matrix[indices].sum(axis=0), where tfidf_matrix is a SciPy CSR object. Summing over the rows of a sparse CSR matrix is computationally much faster, but that summation returns a 1-by-n shaped matrix that is not a NumPy object. We need to convert the output to a NumPy array by running np.asarray(tfidf_matrix[indices].sum(axis=0))[0].

Listing 15.49 Ranking the top 10 words with TFIDF

```
def rank_words_by_tfidf(indices, word_list=words):
    summed_tfidf = np.asarray(tfidf_matrix[indices].sum(axis=0))[0] ←
    data = {'Word': word_list,
            'Summed TFIDF': summed_tfidf}
    return pd.DataFrame(data).sort_values('Summed TFIDF', ascending=False)

df_ranked_words = rank_words_by_tfidf(not_autos_indices)
print(df_ranked_words[:10].to_string(index=False))

   Word  Summed TFIDF
0   car      8.026003
1  cars      1.842831
2  radar      1.408331
3  radio      1.365664
4   ham      1.273830
5   com      1.164511
6 odometer      1.162576
7   speed      1.145510
8   just      1.144489
9  writes      1.070528
```

This summation is equivalent to running `tfidf_np_matrix[indices].sum(axis=0)`. The simpler NumPy array aggregation takes approximately 1 second to compute. A single second may not seem like much, but once we repeat the computation over 20 clusters, the running time will add up to 20 seconds. The summation over the rows of the sparse matrix is noticeably faster.

The first two top-ranking words are *car* and *cars*.

NOTE The word *cars* is the plural of *car*. We can aggregate these words together based on the *s* at the end of *cars*. This process of reducing a plural to its root word is called *stemming*. The external Natural Language Toolkit library (<https://www.nltk.org>) provides useful functions for efficient stemming.

Elsewhere on the ranked list, we see mentions of *radar*, *odometer*, and *speed*. Some of these terms also appeared in our randomly chosen *sci.electronics* post. The use of radar technology to measure car speed appears to be a common theme in the texts represented by `not_autos_indices`. How do these speed-themed keywords compare with the rest of the posts in the car cluster? We can check by inputting `df_car.Index.values` into `rank_words_by_tfidf`.

Listing 15.50 Ranking the top 10 words in the car cluster

```
df_ranked_words = rank_words_by_tfidf(df_car.Index.values)
print(df_ranked_words[:10].to_string(index=False))

   Word  Summed TFIDF
0   car      47.824319
1  cars      17.875903
2  engine      10.947385
3  dealer      8.416367
4   com      7.902425
5   just      7.303276
6  writes      7.272754
7   edu      7.216044
8 article      6.768039
9   good      6.685494
```

Generally, the posts in the `df_car` cluster focus on car engines and car dealers. However, a minority of the posts discuss radar measurements of car speed. These radar posts are more likely to appear in the `sci.electronics` newsgroup. Nonetheless, these posts legitimately discuss cars (as opposed to discussing politics, software, or medicine). Thus, our `df_car` cluster appears to be genuine. By examining the top keywords, we were able to validate the cluster without having to read each clustered post manually.

In this same manner, we can utilize `rank_words_by_tfidf` to get the top keywords for each of the 20 clusters. The keywords will allow us to understand the topic of each cluster. Unfortunately, printing 20 different word tables isn't very visually efficient—the printed tables will take up too much space, adding redundant pages to this book. Alternatively, we can visualize these cluster keywords as images in a single coherent plot. Let's learn how to visualize the contents of multiple text clusters.

15.6 Visualizing text clusters

Our aim is to visualize ranked keywords across multiple text clusters. First we need to solve a simpler problem: how do we visualize the important keywords in a single cluster? One approach is just to print the keywords in their order of importance. Unfortunately, this sorting lacks any sense of relative significance. For instance, in our `df_ranked_words` table, the word `cars` is immediately followed by `engine`. However, `cars` has a summed TFIDF score of 17.8, while `engine` has a score of 10.9. Thus, `cars` is approximately 1.6 times more significant than `engine`, relative to the car cluster. How do we incorporate relative significance into our visualization? Well, we could signify importance using font size: we could display `cars` with a font size of 17.8 and `engine` with a font size of 10.9. In the display, `cars` would be 1.6 times bigger and thus 1.6 times more important. Of course, a font size of 10.9 may be too small to comfortably read. We can make the font size bigger by doubling the summed TFIDF significance scores.

Python does not let us modify font size directly during printing. However, we can modify font size using Matplotlib's `plt.text` function. Running `plt.text(x, y, word, fontsize=z)` displays a word at coordinates `(x, y)` and sets the font size to equal `z`. The function allows us to visualize the words in a 2D grid where word size is proportional to significance. This type of visualization is called a *word cloud*. Let's utilize `plt.text` to generate a word cloud of the top words in `df_ranked_words`. We plot the word cloud as a five-word by five-word grid (figure 15.2). Each word's font size equals double its significance score.

Listing 15.51 Plotting a word cloud with Matplotlib

```
i = 0
for x_coord in np.arange(0, 1, .2):
    for y_coord in np.arange(0, 1, .2):
        word, significance = df_ranked_words.iloc[i].values
        plt.text(y_coord, x_coord, word, fontsize=2*significance)
    i += 1

plt.show()
```

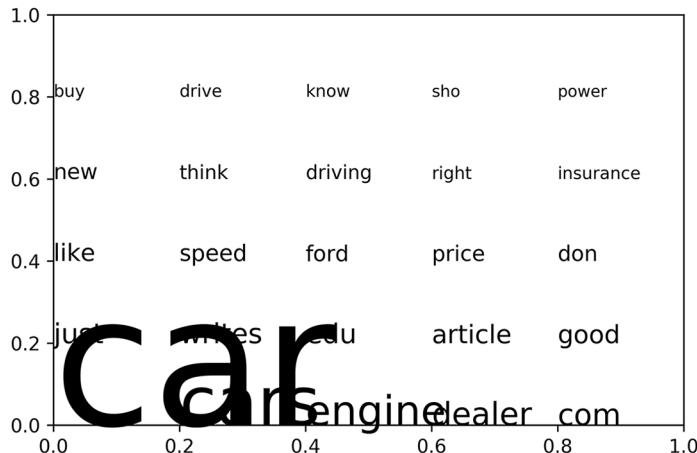


Figure 15.2 A word cloud generated using Matplotlib. The word cloud is a mess because of word overlap.

Our visualization is a mess! Large words like *car* take up too much space. They overlap with other words, making the image indecipherable. We need to plot our words much more intelligently. No two words should ever overlap. Eliminating the overlap of 2D plotted words is not a trivial task. Fortunately, the hard work has been done for us by the creators of the external Wordcloud library. The library is able to generate word clouds in a manner that's visually appealing. We now install Wordcloud and then import and initialize the library's WordCloud class.

NOTE Call pip install wordcloud from the command line terminal to install the Wordcloud library.

Listing 15.52 Initializing the WordCloud class

The positions of the words in the word cloud are generated randomly. To maintain output consistency, we must pass the random seed directly using the random_state parameter.

```
from wordcloud import WordCloud
cloud_generator = WordCloud(random_state=1)
```

Running WordCloud() returns a `cloud_generator` object. We'll use the object's `fit_words` method to generate a word cloud. Running `cloud_generator.fit_words(words_to_score)` will create an image from `words_to_score`, which is a dictionary mapping of words to their significance scores.

NOTE Please note that running `cloud_generator.generate_from_frequencies(word_to_score)` will achieve the same results.

Let's create an image from the most significant words in `df_ranked_words`. We'll store that image in a `wordcloud_image` variable, but we won't plot the image just yet.

Listing 15.53 Generating a word cloud image

```
words_to_score = {word: score
                  for word, score in df_ranked_words[:10].values}
wordcloud_image = cloud_generator.fit_words(words_to_score)
```

Now we're ready to visualize `wordcloud_image`. Matplotlib's `plt.imshow` function is able to plot images based on a variety of inputted image formats. Running `plt.imshow(wordcloud_image)` will display our generated word cloud (figure 15.3).

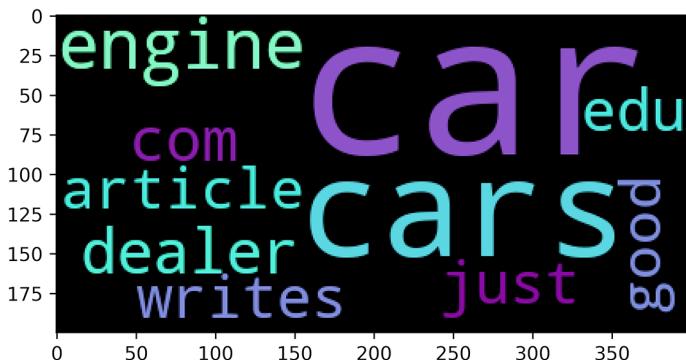


Figure 15.3 A word cloud generated using the `WordCloud` class. The words no longer overlap. However, the background is too dark. Also, some letters appear rough around the edges.

NOTE There are multiple ways of representing images in Python. One approach is to store an image as a 2D NumPy array. Alternatively, we can store the image using a special class from the Python Imaging Library (PIL). The `plt.imshow` function can display images stored as NumPy objects or as PIL Image objects. It can also display custom image objects that include a `to_image` method, but that method's output must return a NumPy array or a PIL Image object.

Listing 15.54 Plotting an image using `plt.imshow`

```
plt.imshow(wordcloud_image)
plt.show()
```

We've visualized the word cloud. Our visualization is not ideal: the dark background makes it hard to read the words. We can change the background from black to white by running `WordCloud(background_color='white')` during initialization. Also, the edges of the individual letters are pixelated and blocky: we can smooth all the edges in our image plot by passing `interpolation="bilinear"` into `plt.imshow`. Let's regenerate the word cloud with a lighter background while also smoothing out the visualized letters (figure 15.4).

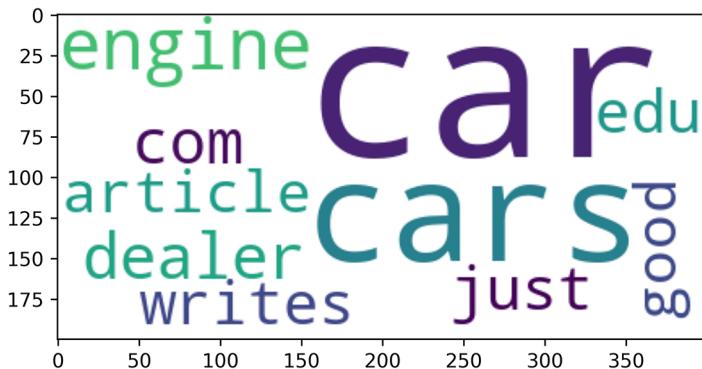


Figure 15.4 A word cloud generated using the `WordCloud` class. The background is set to white, for better visibility, and the letters are smoothed out at the edges.

Listing 15.55 Improving the word cloud image quality

```
cloud_generator = WordCloud(background_color='white',
                             random_state=1)
wordcloud_image = cloud_generator.fit_words(words_to_score)
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.show()
```

The top words in the car cluster have been successfully visualized. The words *car* and *cars* clearly dominate over lesser terms, such as *engine* and *dealer*. We can interpret the contents of the cluster merely by glancing at the word cloud. Of course, we have already examined the car cluster in great detail, and we aren't gleaming anything new from this visualization. Let's instead apply word cloud visualization to a randomly chosen cluster (figure 15.5). The word cloud will display the cluster's 15 most significant words, and we'll use the display to figure out the main topic of the cluster.

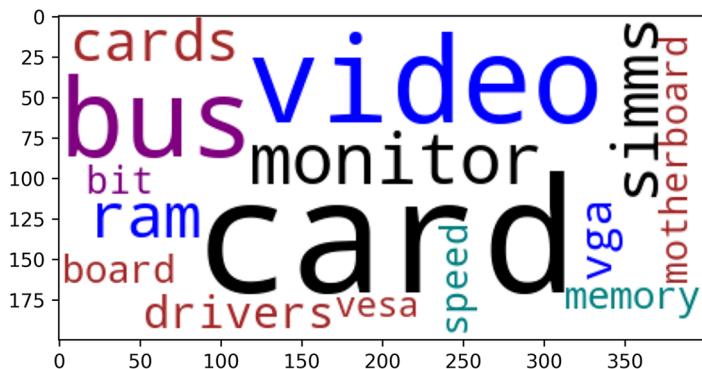


Figure 15.5 A random cluster's word cloud. The topic of the cluster appears to be technology and computer hardware.

NOTE The word colors in the word cloud are generated at random, and some of the random colors would render poorly in the black-and-white version of this book. For this reason, we specifically limit the color selection to a small subset of colors using the `color_func` parameter in the `WordCloud` class.

Listing 15.56 Plotting a word cloud for a random cluster

```
Takes as input a df_cluster table and returns a word cloud image for the top max_words
words corresponding to the cluster. The previously defined rank_words_by_tfidf
function is used to rank the words in the cluster.

np.random.seed(1)

def cluster_to_image(df_cluster, max_words=15):
    indices = df_cluster.Index.values
    df_ranked_words = rank_words_by_tfidf(indices) [:max_words]
    words_to_score = {word: score
                      for word, score in df_ranked_words[:max_words].values}
    cloud_generator = WordCloud(background_color='white',
                                color_func=_color_func,
                                random_state=1)
    wordcloud_image = cloud_generator.fit_words(words_to_score)
    return wordcloud_image

→ def _color_func(*args, **kwargs):
    return np.random.choice(['black', 'blue', 'teal', 'purple', 'brown'])

cluster_id = np.random.randint(0, 20)
df_random_cluster = df[df.Cluster == cluster_id]
wordcloud_image = cluster_to_image(df_random_cluster)
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.show()

Helper function to randomly
assign one of five acceptable
colors to each word
```

The `WordCloud` class includes an optional `color_func` parameter. The parameter expects a color-selection function that assigns a color to each word. Here, we define a custom function to control the color settings.

Our randomly chosen cluster includes top words such as `monitor`, `video`, `memory`, `card`, `motherboard`, `bit`, and `ram`. The cluster seems to focus on technology and computer hardware. We can verify by printing the most common newsgroup category in the cluster.

NOTE By observing the words `card`, `video`, and `memory`, we can infer that `card` refers to either `video card` or a `memory card`. In NLP, such sequences of two consecutive words are called *bigrams*. Generally, a sequence of n consecutive words is called an *n-gram*. `TfidfVectorizer` is able to vectorize across n-grams of arbitrary length. We simply need to pass in an `ngram_range` parameter during initialization. Running `TfidfVectorizer(ngram_range=(1, 3))` creates a vectorizer that tracks all 1-grams (single words), 2-grams (such as `video card`), and 3-grams (such as `natural language processing`). Of course, these n-grams cause the vocabulary size to rise into the millions. However, we can limit the vocabulary to the top 100,000 n-grams by passing `max_features=100000` into the vectorizer's initialization method.

Listing 15.57 Checking the most common cluster category

```
from collections import Counter

def get_top_category(df_cluster):
    categories = [get_post_category(index)
                  for index in df_cluster.Index.values]
    top_category, _ = Counter(categories).most_common()[0]
    return top_category

top_category = get_top_category(df_random_cluster)
print("The posts within the cluster commonly appear in the "
      f"'{top_category}' newsgroup")
```

The posts within the cluster commonly appear in the
'comp.sys.ibm.pc.hardware' newsgroup

Many of the posts in the cluster appeared in the *comp.sys.ibm.pc.hardware* newsgroup. We've thus successfully identified the cluster's topic of hardware. We did this simply by looking at the word cloud.

So far, we've generated two separate word clouds for two distinct clusters. However, our end goal is to display multiple word clouds simultaneously. We'll now visualize all word clouds in a single figure using a Matplotlib concept called a *subplot*.

Common methods for visualizing words

- `plt.text(word, x, y, fontsize=z)`—Plots a word positioned at coordinates (x, y) with a font size of z.
- `cloud_generator = WordCloud()`—Initializes an object that can generate a word cloud. The background of that word cloud is black.
- `cloud_generator = WordCloud(background_color='white')`—Initializes an object that can generate a word cloud. The background of that word cloud is white.
- `wordcloud_image = cloud_generator.fit_words(words_to_score)`—Generates a word cloud image from the `words_to_score` dictionary, which maps words to their significance scores. The size of every word in `wordcloud_image` is computed relative to its significance.
- `plt.imshow(wordcloud_image)`—Plots the computed `wordcloud_image`.
- `plt.imshow(wordcloud_image, interpolation="bilinear")`—Plots the computed `wordcloud_image` while smoothing out the visualized letters.

15.6.1 Using subplots to display multiple word clouds

Matplotlib allows us to include multiple plots in a single figure. Each distinct plot is called a *subplot*. Subplots can be organized in any number of ways, but they're most commonly arranged in a grid-like pattern. We can create a subplot grid containing r rows and c columns by running `plt.subplots(r, c)`. The `plt.subplots` function

generates the grid while also returning a tuple: `(figure, axes)`. The `figure` variable is a special class that tracks the main figure, which encompasses the grid. Meanwhile, the `axes` variable is a 2D list containing `r` rows and `c` columns. Each element of `axes` is a Matplotlib `AxesSubplot` object. Every subplot object can be used to output a unique visualization: running `axes[i][j].plot(x, y)` plots `x` versus `y` in the subplot positioned in the `i`th row and `j`th column of the grid.

WARNING Running `subplots(1, z)` or `subplots(z, 1)` returns a 1D axes list where `len(axes) == z` rather than a 2D grid.

Let's demonstrate the use of `plt.subplots`. We generate a two-by-two grid of subplots by running `plt.subplots(2, 2)`. Then we iterate over each row `r` and column `c` in the grid. For every unique subplot positioned at `(r, c)`, we plot a quadratic curve in which $y = r * x^2 + c * x$. By linking curve parameters to the grid position, we generate four distinct curves, all of which appear in the bounds of a single figure (figure 15.6).

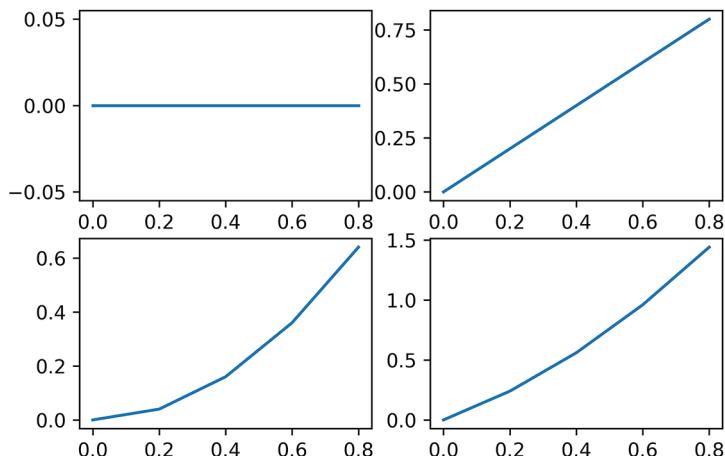


Figure 15.6 Four different curves plotted across four subplots in a single figure

Listing 15.58 Generating four subplots using Matplotlib

```
figure, axes = plt.subplots(2, 2)
for r in range(2):
    for c in range(2):
        x = np.arange(0, 1, .2)
        y = r * x * x + c * x
        axes[r][c].plot(x, y)

plt.show()
```

Four different curves appear in the subplots of our grid. We can replace any of these curves with a word cloud. Let's visualize `wordcloud_image` in the lower-left quadrant

of the grid by running `axes[1][0].imshow(wordcloud_image)` (figure 15.7). Let's also assign a title to that subplot: the title equals `top_category`, which is `comp.sys.ibm.pc.hardware`. We set the subplot title by running `axes[r][c].set_title(top_category)`.

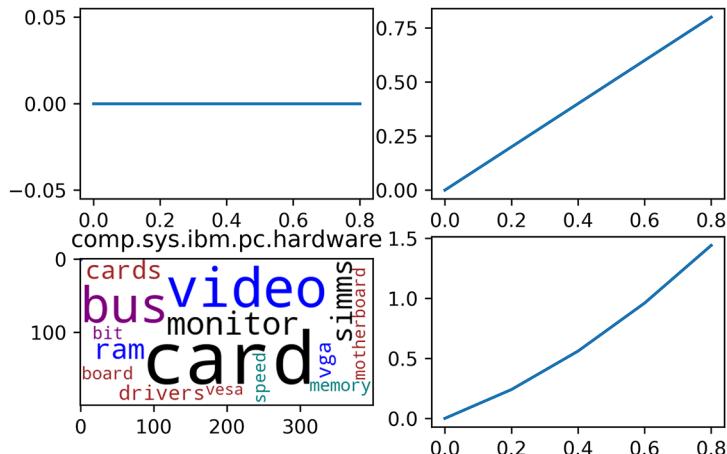


Figure 15.7 Three curves and a word cloud plotted in four subplots. There are issues with word cloud readability due to formatting problems and figure size.

Listing 15.59 Plotting a word cloud within a subplot

```
figure, axes = plt.subplots(2, 2)
for r in range(2):
    for c in range(2):
        if (r, c) == (1, 0):
            axes[r][c].set_title(top_category)
            axes[r][c].imshow(wordcloud_image,
                               interpolation="bilinear")
        else:
            x = np.arange(0, 1, .2)
            y = r * x * x + c * x
            axes[r][c].plot(x, y)

plt.show()
```

We've visualized a word cloud in the subplot grid, but there are some issues with the visualization. The words in the cloud are hard to read because the subplot is so small. We need to make the subplot bigger, which requires us to alter the figure size. We can do so using the `figsize` parameter. Passing `figsize=(width, height)` into `plt.subplots` creates a figure that is `width` inches wide and `height` inches high. Each subplot in the figure is also adjusted to fit the updated size.

In addition, we can make other minor changes to improve the plot. Reducing the visualized word count from 15 to 10 will make the smaller word cloud easier to read.

We should also remove the axis tick marks from the plot—they take up too much space and provide no useful information. We can delete the x-axis and y-axis tick marks from `axis[r][c]` by calling `axis[r][c].set_xticks([])` and `axis[r][c].set_yticks([])`, respectively. With this in mind, let's generate a figure that's 20 inches wide and 15 inches high (listing 15.60).

NOTE The actual dimensions of the figure in the book are not 20 by 15 inches, due to image formatting.

The large figure has 20 subplots aligned in a five-by-four grid. Each subplot contains a word cloud corresponding to one of our clusters, and every subplot title is set to the dominant newsgroup category in a cluster. We also include the cluster index in each title for later reference. Finally, we remove the axis tick marks from all plots. The final visualization gives us a bird's-eye-view of all the dominant word patterns across all 20 clusters (figure 15.8).



Figure 15.8 20 word clouds visualized across 20 subplots. Each word cloud corresponds to one of 20 clusters. The title of each subplot equals the top newsgroup category in each cluster. In most of the word clouds, the title corresponds with the displayed word content; but certain word clouds (such as those of clusters 1 and 7) do not offer informative displays.

Common subplot methods

- `figure, axes = plt.subplots(x, y)`—Creates a figure containing an x-by-y grid of subplots. If $x > 1$ and $y > 1$, then `axes[r][c]` corresponds to the subplot in row r and column c of the subplot grid.
- `figure, axes = plt.subplots(x, y, figsize=(width, height))`—Creates a figure containing an x-by-y grid of subplots. The figure is width inches wide and height inches high.
- `axes[r][c].plot(x_values, y_values)`—Plots data in the subplot positioned at row r and column c .
- `axes[r][c].set_title(title)`—Adds a title to the subplot positioned at row r and column c .

Listing 15.60 Visualizing all clusters using 20 subplots

```

np.random.seed(0)

def get_title(df_cluster):
    top_category = get_top_category(df_cluster)
    cluster_id = df_cluster.Cluster.values[0]
    return f'{cluster_id}: {top_category}' → Generates a subplot title by combining
                                                the cluster ID with the most common
                                                newsgroup category in a cluster

figure, axes = plt.subplots(5, 4, figsize=(20, 15))
cluster_groups = list(df.groupby('Cluster'))
for r in range(5):
    for c in range(4):
        _, df_cluster = cluster_groups.pop(0)
        wordcloud_image = cluster_to_image(df_cluster, max_words=10)
        ax = axes[r][c]
        ax.imshow(wordcloud_image,
                  interpolation="bilinear")
        ax.set_title(get_title(df_cluster), fontsize=20) ← Increases the title
                                                       font to 20 for better
                                                       readability
        ax.set_xticks([])
        ax.set_yticks([])

plt.show()

```

We've visualized the top words across all 20 clusters. For the most part, the visualized words make sense! The main topic of cluster 0 is cryptography: its top words include *encryption*, *secure*, *keys*, and *nsa*. The main topic of cluster 2 is space: its top words include *space*, *nasa*, *shuttle*, *moon*, and *orbit*. Cluster 4 is centered on shopping, with top words like *sale*, *offer*, *shipping*, and *condition*. Clusters 9 and 18 are sports clusters: their main topics are *baseball* and *hockey*, respectively. Posts in cluster 9 frequently mention *games*, *runs*, *baseball*, *pitching*, and *team*. Posts in cluster 18 frequently mention *game*, *team*, *players*, *hockey*, and *nhl*. A majority of the clusters are easy to interpret, based on their word clouds. Arguably, 75% of the clusters contain top words corresponding with their dominant category titles.

Of course, there are issues with our output. Several of the word clouds do not make sense: for instance, cluster 1 has a subplot title of `sci.electronics`, yet its word cloud is composed of general words like *just*, *like*, *does*, and *know*. Meanwhile, cluster 7 has a subplot title of `sci.med`, yet its word cloud is composed of words like *pitt*, *msg*, and *gordon*. Unfortunately, word cloud visualization isn't always perfect. Sometimes the underlying clusters are malformed, or the dominant language in the clusters is biased toward unexpected text patterns.

NOTE Reading some sampled posts in the clusters can help reveal these biases.

For instance, many of the electronics questions in cluster 1 include the question, “does anyone know?” Also, many of the posts in cluster 7 were written by a student named Gordon, who studied at the University of Pittsburgh (*pitt*).

Fortunately, there are steps we can take to salvage the indecipherable word clouds. For instance, we can filter out obviously useless words and then regenerate the cloud. Or we can simply disregard the top x words in the cluster and visualize the cloud using the next top-ranking words. Let's remove the top 10 words from cluster 7 and recompute its word cloud (figure 15.9).

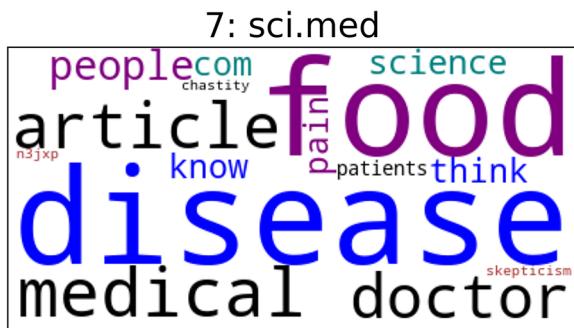


Figure 15.9 Cluster 7's word cloud, recomputed after filtering. It is now clear that medicine is the main topic of the cluster.

Listing 15.61 Recomputing a word cloud after filtering

```
np.random.seed(3)
df_cluster= df[df.Cluster == 7]
df_ranked_words = rank_words_by_tfidf(df_cluster.Index.values)

words_to_score = {word: score
                  for word, score in df_ranked_words[10:25].values}
cloud_generator = WordCloud(background_color='white',
                             color_func=_color_func,
                             random_state=1)
wordcloud_image = cloud_generator.fit_words(words_to_score)
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.title(get_title(df_cluster), fontsize=20)
plt.xticks([])
plt.yticks([])
plt.show()
```

Note that `plt` lacks a subplot's `set_title`, `set_xticks`, and `set_yticks` methods. Instead, we must call `plt.title`, `plt.xticks`, and `plt.yticks` to achieve the same results.

We visualize the top 15 words since we're not spatially restricted by a subplot.

Cluster 7 is dominated by words like *disease*, *medical*, *doctor*, *food*, *pain*, and *patients*. Its medicinal topic is now clear; by disregarding the useless keywords, we've managed to elucidate the cluster's true contents. Of course, this simple approach will not always work. NLP is messy, and there is no silver bullet that will slay all of our problems. Nonetheless, there's still much we can accomplish, despite the unstructured nature of complex texts. Consider what we have achieved: we've taken 10,000 diverse real-world texts and clustered them into multiple meaningful topics. Furthermore, we've visualized these topics in a single image, and most of the topics in that image are interpretable. We've attained these results using a straightforward series of steps, which can be applied to any large text dataset. Effectively, we've developed a pipeline for clustering and visualizing unstructured text data. That pipeline works as follows:

- 1 Transform our text into a normalized TFIDF matrix using the `TfidfVectorizer` class.
- 2 Reduce the matrix to 100 dimensions using the SVD algorithm.
- 3 Normalize the dimensionally reduced output for clustering purposes.
- 4 Cluster the normalized output using K-means. We can estimate K by generating an elbow plot using mini-batch K-means, which is optimized for speed.
- 5 Visualize the top words in each cluster using a word cloud. All word clouds are displayed as subplots in a single figure. Words are ranked based on their summed TFIDF values across all the texts in a cluster.
- 6 Interpret the topic of each cluster using the word cloud visualization. Any uninterpretable clusters are examined in more detail.

Given our text-analysis pipeline, we can effectively cluster and interpret almost any real-world text dataset.

Summary

- Scikit-learn's newsgroup dataset contains over 10,000 newsgroup posts spread out across 20 newsgroup categories.
- We can convert the posts into a TF matrix using scikit-learn's `CountVectorizer` class. The generated matrix is stored in the *CSR* format. This format is used to efficiently analyze sparse matrices, which are composed mostly of zeros.
- Generally, TF matrices are sparse; a single row may only reference a few dozen words from the entire dataset vocabulary. We can access these nonzero words with the help of the `np.flatnonzero` function.
- The most frequently occurring words in a text tend to be *stop words*, which are common English words like *the* or *this*. Stop words should be filtered from text datasets before vectorization.
- Even after stop-word filtering, certain overly common words will remain. We can minimize the impact of these words using their document frequencies. A word's *document frequency* equals the total fraction of texts in which that word

appears. Words that are more common are less significant. Hence, less significant words have higher document frequencies.

- We can combine term frequencies and document frequencies into a single significance score called *TFIDF*. Generally, TFIDF vectors are more informative than TF vectors. We can convert texts to TFIDF vectors using scikit-learn's `TfidfVectorizer` class. That vectorizer returns a TFIDF matrix whose rows are automatically normalized for easier similarity computation.
- Large TFIDF matrices should be dimensionally reduced before clustering. The recommended number of dimensions is 100. Scikit-learn's dimensionally reduced SVD output needs to be normalized before subsequent analysis.
- We can cluster normalized, dimensionally reduced text data using either K-means or DBSCAN. Unfortunately, it's hard to optimize DBSCAN's parameters during text clustering. Thus, K-means remains the preferable clustering algorithm. We can estimate K using an elbow plot. If our dataset is large, we should generate the plot using `MiniBatchKMeans` for a faster runtime.
- For any given text cluster, we want to view the words that are most relevant to the cluster. We can rank each word by summing its TFIDF values across all matrix rows represented by the cluster. Furthermore, we can visualize the ranked words in a *word cloud*: a 2D image composed of words, where word size is proportional to significance.
- We can plot multiple word clouds in a single figure using the `plt.subplots` function. This visualization gives us a bird's-eye view of all the dominant word patterns across all clusters.

16

Extracting text from web pages

This section covers

- Rendering web pages with HTML
- The basic structure of HTML files
- Extracting text from HTML files with the Beautiful Soup library
- Downloading HTML files from online sources

The internet is a great resource for text data. Millions of web pages offer limitless text content in the form of news articles, encyclopedia pages, scientific papers, restaurant reviews, political discussions, patents, corporate financial statements, job postings, etc. All these pages can be analyzed if we download their Hypertext Markup Language (HTML) files. A *markup language* is a system for annotating documents that distinguishes the annotations from the document text. In the case of HTML, these annotations are instructions on how to visualize a web page.

Web page visualization is usually carried out using a web browser. First, the browser downloads the page's HTML based on its web address, the URL. Next, the browser parses the HTML document for layout instructions. Finally, the browser's rendering engine formats and displays all images and text per the markup specifications. The rendered page can easily be read by a human being.

Of course, during large-scale data analysis, we don't need to render every page. Computers can process document texts without requiring any visualization. Thus, when analyzing HTML documents, we can focus on the text while skipping over the display instructions. Nonetheless, we shouldn't totally ignore the annotations—they can provide us with valuable information. For example, the annotated title of a document can summarize the document's contents concisely. Therefore, we can benefit by discerning that title from an annotated paragraph in the document. If we can distinguish between various document parts, we can run a more informed investigation. Consequently, a basic knowledge of HTML structure is imperative for online text analysis. With this in mind, we begin this section by reviewing the HTML structure. Then we learn how to parse that structure using Python libraries.

NOTE If you are already familiar with basic HTML, feel free to skip ahead to subsection 16.2.

16.1 The structure of HTML documents

An HTML document is composed of HTML elements. Each element corresponds to a document component. For instance, the document's title is an element; every single paragraph in the document is also an element. The starting location of an element is demarcated by a start tag: for instance, the start tag of a title is `<title>`, and the start tag of a paragraph is `<p>`. Every start tag begins and ends with angled brackets, `<>`. Adding a forward slash to the tag transforms it into an end tag. The endpoints of most elements are demarcated by end tags: thus the immediate text of a title is followed by `</title>`, and the text of a paragraph is followed by `</p>`.

Shortly, we explore many common HTML tags. But first we must introduce the most important HTML tag: `<html>`, which specifies the start of the entire HTML document. Let's utilize that tag to create a document composed of just a single word: *Hello*. We generate the contents of the document by coding `html_contents = "<html>Hello</html>"`

Listing 16.1 Defining a simple HTML string

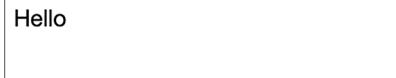
```
html_contents = "<html>Hello</html>"
```

HTML contents are intended to be rendered in a web browser. Thus, we can visualize `html_contents` by saving it to a file and then loading it in a browser of our choice. Alternatively, we can render `html_contents` directly in an IPython Jupyter Notebook. We simply need to import `HTML` and `display` from `IPython.core.display`. Then, executing `display(HTML(html_contents))` will display the rendered output (figure 16.1).

Listing 16.2 Rendering an HTML string

```
from IPython.core.display import display, HTML
def render(html_contents): display(HTML(html_contents)) ←
    render(html_contents)
```

Defines a single-line
render function to
repeatedly visualize our
HTML using less code

A screenshot of a web browser window displaying a single word "Hello" in a plain black font on a white background.

Hello

Figure 16.1 A rendered HTML document. It contains a single word: *Hello*.

We've rendered our HTML document. It's not very impressive—the body is composed of a single word. Furthermore, the document lacks a title. Let's assign the document a title using the `<title>` tag. We'll set the title to something simple, like *Data Science is Fun*. To do so, we begin by creating a title string that is equal to "`<title>Data Science is Fun</title>`".

Listing 16.3 Defining a title in HTML

```
title = "<title>Data Science is Fun</title>"
```

Now we nest the title in `<html>` and `</html>` by running `html_contents = f"<html>{title}Hello</html>"` and then render the updated contents (figure 16.2).

A screenshot of a web browser window displaying a single word "Hello" in a plain black font on a white background.

Hello

Figure 16.2 A rendered HTML document. The document's title does not appear in the rendered output—only the word *Hello* is visible.

Listing 16.4 Adding a title to the HTML string

```
html_contents = f"<html>{title}Hello</html>"  
render(html_contents)
```

Our output is identical to what we saw before! The title does not appear in the body of the rendered HTML; it only appears in the title bar of the web browser (figure 16.3).



Figure 16.3 A web browser rendering of the HTML document. The document's title appears in the browser's title bar.

Despite its partial invisibility, the title provides us with very important information: it summarizes the contents of the document. For instance, in a job listing, the title directly summarizes the nature of the job. Thus, the title reflects vital information despite its absence from the body of the document. This critical distinction is commonly emphasized using `<head>` and `<body>` tags. The content delimited by the HTML

`<body>` tag will appear in the body of the output. Meanwhile, `<head>` delimits vital information that is not rendered in the body. Let's emphasize this distinction by nesting `title` in the `head` element of the HTML. We also nest the visible *Hello* in the `body` element of the contents.

Listing 16.5 Adding a head and body to the HTML string

```
head = f"<head>{title}</head>"  
body = "<body>Hello</body>"  
html_contents = f"<html> {title} {body}</html>"
```

Occasionally, we want to display a document's title in the body of a page. For instance, in a job posting, the employer will likely want to show the title of the job. This visualized title is referred to as the page *header* and is demarcated with the `<h1>` tag. Of course, that tag is nested in `<body>`, where all visualized content is found. Let's add a header to the body of our HTML (figure 16.4).

Listing 16.6 Adding a header to the HTML string

HTML elements can be nested like Russian nesting dolls. Here we nest the header element inside the body element, and we nest both the body and title elements in the `<html>` and `</html>` tags.

```
header = "<h1>Data Science is Fun</h1>"  
body = f"<body>{header}Hello</body>"  
html_contents = f"<html> {title} {body}</html>"  
render(html_contents)
```



Data Science is Fun
Hello

Figure 16.4 A rendered HTML document. A large header appears in the rendered output.

Our single word looks awkward relative to the large header. Generally, HTML documents are intended to have more than one word in the body—they usually contain multiple sentences in multiple paragraphs. As previously mentioned, such paragraphs are marked with a `<p>` tag.

Let's add two consecutive paragraphs to our HTML (figure 16.5). We compose these dummy paragraphs from sequences of repeating words: the first paragraph features the phrase *Paragraph 0* repeating 40 times; in the subsequent paragraph, we replace 0 with 1.

Listing 16.7 Adding paragraphs to the HTML string

```
paragraphs = ''  
for i in range(2):
```

```
paragraph_string = f"Paragraph {i} " * 40
paragraphs += f"<p>{paragraph_string}</p>"\n\nbody = f"<body>{header}{paragraphs}</body>"\nhtml_contents = f"<html> {title} {body}</html>"\nrender(html_contents)
```

Data Science is Fun

Figure 16.5 A rendered HTML document. Two paragraphs appear in the rendered output.

We've inserted paragraph elements into our HTML. These elements are distinguishable by their internal text. However, their `<p>` tags are both identical; an HTML parser cannot easily distinguish between the first and second tags. Occasionally, it's worth making the difference between tags much more pronounced (particularly if each paragraph is formatted uniquely). We can discriminate between `<p>` tags by assigning each tag a unique ID, which can be inserted directly into the tag brackets. For example, we identify the first paragraph as *paragraph 0* by writing `<p id="paragraph 0">`. The added `id` is referred to as an *attribute* of the paragraph element. Attributes are inserted into element start tags to track useful tag information.

We now add `id` attributes to our paragraph tags. Later, we utilize these attributes to distinguish between the paragraphs.

Listing 16.8 Adding id attributes to the paragraphs

```
paragraphs = ''
for i in range(2):
    paragraph_string = f"Paragraph {i} " * 40
    attribute = f"id='paragraph {i}'"
    paragraphs += f"<p {attribute}>{paragraph_string}</p>"

body = f"<body>{header}{paragraphs}</body>"
html_contents = f"<html> {title} {body}</html>"
```

HTML attributes play many critical roles. They are especially necessary when linking between documents. The internet is built on top of *hyperlinks*, which are clickable texts that connect web pages. Clicking a hyperlink takes you to a new HTML document. Each hyperlink is marked by an anchor tag, `<a>`, which makes the text clickable.

However, additional information is required to specify the address of the linked document. We can provide that information using the `href` attribute, where `href` stands for *hypertext reference*. For instance, demarcating text with `` links that text to the Manning website.

Next, we create a hyperlink that reads *Data Science Bookcamp* and link that clickable text to the website for this book. Then we insert the hyperlink into a new paragraph and assign that paragraph an ID of paragraph 3 (figure 16.6).

Figure 16.6 A rendered HTML document. One more paragraph has been added to the rendered output, containing a clickable link to *Data Science Bookcamp*.

Listing 16.9 Adding a hyperlink to the HTML string

Creates a clickable hyperlink. Clicking the words Data Science Bookcamp will take a user to the book's online URL.

```
link_text = "Data Science Bookcamp"                                user to the book's online URL
url = "https://www.manning.com/books/data-science-bookcamp"
hyperlink = f"<a href='{url}'>{link_text}</a>"                     ←
new_paragraph = f"<p id='paragraph 2'>Here is a link to {hyperlink}</p>" ←
paragraphs += new_paragraph
body = f"<body>{header}{paragraphs}</body>"                      ←
html_contents = f"<html> {title} {body}</html>"                   ←
render(html_contents)
```

HTML text elements can vary in complexity. Beyond just headers and paragraphs, we can also visualize lists of texts in an HTML document. Suppose, for instance, that we wish to display a list of popular data science libraries. We start by defining that list in Python.

Listing 16.10 Defining a list of data science libraries

```
libraries = ['NumPy', 'SciPy', 'Pandas', 'Scikit-Learn']
```

Now we demarcate every item in our list with an `` tag, which stands for *list item*. We store these items in an `items` string.

Listing 16.11 Demarcating list items with an `` tag.

```
    items = ''
    for library in libraries:
        items += f"<li>{library}</li>"
```

Finally, we nest the `items` string in a `` tag, where `ul` stands for *unstructured list*. Then we append the unstructured list to the body of our HTML. We also insert a second header between the paragraphs and the list: *Common Data Science Libraries*. We use the `<h2>` tag to differentiate between the second header and the first (figure 16.7).

Listing 16.12 Adding an unstructured list to the HTML string

```
unstructured_list = f"<ul>{items}</ul>"  
header2 = '<h2>Common Data Science Libraries</h2>'  
body = f"<body>{header}{paragraphs}{header2}{unstructured_list}</body>"  
html_contents = f"<html> {title} {body}</html>"  
render(html_contents)
```

Data Science is Fun

Here is a link to [Data Science Bookcamp](#)

Common Data Science Libraries

- NumPy
 - Scipy
 - Pandas
 - Scikit-Learn

Figure 16.7 A rendered HTML document. The updated document contains a bulleted list of common data science libraries.

The data science libraries have been rendered as a list of bulleted points. Each bullet occupies a separate line. Traditionally, such bullet points are used to signify diverse conceptual categories, ranging from data science libraries to breakfast foods to required skills in a job posting.

At this point, it's worth noting that our HTML body is divided into two distinct parts: the first part corresponds to a sequence of three paragraphs, and the second part corresponds to the bulleted list. Typically, such divisions are captured using special `<div>` tags that allow frontend engineers to track the divided elements and customize

their formatting accordingly. Usually, each `<div>` tag is distinguished by some attribute. If the attribute is unique to a division, then that attribute is an `id`; if the attribute is shared by more than one division, a special `class` signifier is used.

For consistency's sake, we divide our two sections by nesting them in two different divisions. The first division is assigned a paragraph ID, and the second is assigned a list ID. Additionally, since both divisions only contain text, we assign a `text` class attribute to each one. We also add a third empty division to the body; we'll update it later. The ID and class of this empty division are both set to `empty`.

Listing 16.13 Adding divisions to the HTML string

```
div1 = f"<div id='paragraphs' class='text'>{paragraphs}</div>"  
div2 = f"<div id='list' class='text'>{header2}{unstructured_list}</div>"  
div3 = "<div id='empty' class='empty'></div>"    <--  
body = f"<body>{header}{div1}{div2}{div3}</body>"  
html_contents = f"<html> {title}{body}</html>"
```

The third division is empty, but it can still be accessed by both class and ID. Later, we will access this division to insert additional text.

Common HTML elements and attributes

- `<html>..</html>`—Demarcates the entire HTML document.
- `<title>..</title>`—The title of the document. This title appears in a web browser's title bar but not in the browser's rendered contents.
- `<head>..</head>`—The head of the document. The information in the head is not intended to appear in the browser's rendered contents.
- `<body>..</body>`—The body of the document. The information in the body is intended to appear in a browser's rendered contents.
- `<h1>..</h1>`—A header in the document. It is generally rendered in large, bold letters.
- `<h2>..</h2>`—A header in the document whose formatting slightly differs from `<h1>`.
- `<p>..</p>`—A single paragraph in the document.
- `<p id="unique_id">..</p>`—A single paragraph in the document containing a unique `id` attribute that is not shared by any other document elements.
- `..`—A clickable text hyperlink. Clicking the text sends a user to the URL specified in the `href` attribute.
- `..`—An unstructured list composed of individual list items that appear as bullet points in a browser's rendered contents.
- `..`—An individual list item in an unstructured list.
- `<div>..</div>`—A division demarcating a specific subsection of the document.
- `<div class="category_class">..</div>`—A division demarcating a specific subsection of the document. The division is assigned a category class by way of the `class` attribute. Unlike a unique `ID`, this class can be shared across other divisions in the HTML.

We've made many changes to our `html_contents` string. Let's review its altered contents.

Listing 16.14 Printing the altered HTML string

```
print(html_contents)

<html> <title>Data Science is Fun</title><body><h1>Data Science is Fun</h1>
<div id='paragraphs' class='text'><p id='paragraph 0'>Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Here is a link to <a href='https://www.manning.com/books/data-science-bookcamp'>
Data Science Bookcamp</a></p></div><div id='list' class='text'>
<h2>Common Data Science Libraries</h2><ul><li>NumPy</li>
<li>SciPy</li><li>Pandas</li><li>Scikit-Learn</li>
</ul></div><div id='empty' class='empty'></div></body></html>
```

The printed output is a mess! The HTML contents are nearly unreadable. Also, extracting individual elements from `html_contents` is exceedingly difficult. Imagine if we wanted to extract the title of the HTML document: we'd need to first split `html_contents` on the `>` bracket. Then we'd have to iterate over the split results, stopping at the string that's equal to `<title>`. Next, we'd need to go one index over and extract the string containing the title's text. Finally, we'd have to clean the title string by splitting on the remaining `<` bracket. This convoluted title-extraction process is illustrated next.

Listing 16.15 Extracting the HTML title using basic Python

```
split_contents = html_contents.split('>')
for i, substring in enumerate(split_contents):
    if substring.endswith('<title>'):
        next_string = split_contents[i + 1]
        title = next_string.split('<')[0]
        print(title)
        break
```

```
Data Science is Fun
```

Is there a cleaner way to extract elements from HTML documents? Yes! We don't need to manually parse the documents. Instead, we can use the external BeautifulSoup library.

16.2 Parsing HTML using BeautifulSoup

We start by installing the BeautifulSoup library. Then we import the BeautifulSoup class from bs4. Following a common convention, we import BeautifulSoup as simply bs.

NOTE Call pip install bs4 from the command line terminal to install the BeautifulSoup library.

Listing 16.16 Importing the BeautifulSoup class

```
from bs4 import BeautifulSoup as bs
```

We now initialize the BeautifulSoup class by running bs(html_contents). In keeping with convention, we assign the initialized object to a soup variable (listing 16.17).

NOTE By default, the bs class uses Python's built-in HTML parser to extract the HTML contents. However, more efficient parsers are available through external libraries. One popular library is called lxml, which can be installed by running pip install lxml. After installation, the lxml parser can be used during bs initialization. We simply need to execute bs(html_contents, 'lxml').

Listing 16.17 Initializing BeautifulSoup using an HTML string

```
soup = bs(html_contents)
```

Our soup object tracks all elements in the parsed HTML. We can output these elements in a clean, readable format by running the soup.prettify() method.

Listing 16.18 Printing readable HTML with BeautifulSoup

```
print(soup.prettify())  
  
<html>  
  <title>  
    Data Science is Fun  
  </title>  
  <body>  
    <h1>  
      Data Science is Fun  
    </h1>  
    <div class="text" id="paragraphs">  
      <p id="paragraph 0">  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
        Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0  
      </p>  
      <p id="paragraph 1">
```

```

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    </p>
<div id="text" class="text" id="list">
    <p id="paragraph 2">
        Here is a link to
        <a href="https://www.manning.com/books/data-science-bookcamp">
            Data Science Bookcamp
        </a>
    </p>
</div>
<div class="text" id="empty" id="empty">
    <h2>
        Common Data Science Libraries
    </h2>
    <ul>
        <li>
            NumPy
        </li>
        <li>
            SciPy
        </li>
        <li>
            Pandas
        </li>
        <li>
            Scikit-Learn
        </li>
    </ul>
</div>
<div class="empty" id="empty" id="empty">
</div>
</body>
</html>
```

Suppose we want to access an individual element, such as the title. The `soup` object provides that access through its `find` method. Running `soup.find('title')` returns all content enclosed in the title's start and end tags.

Listing 16.19 Extracting the title with BeautifulSoup

```

title = soup.find('title')
print(title)

<title>Data Science is Fun</title>
```

The outputted title appears to be an HTML string demarcated by the title tags. However, our `title` variable is not a string: it's an initialized Beautiful Soup Tag class. We can verify by printing `type(title)`.

Listing 16.20 Outputting the title's data type

```
print(type(title))

<class 'bs4.element.Tag'>
```

Each Tag object contains a `text` attribute, which maps to the text in the tag. Thus, printing `title.text` returns *Data Science is Fun*.

Listing 16.21 Outputting the title's text attribute

```
print(title.text)

Data Science is Fun
```

We've accessed our title tag by running `soup.find('title')`. We can also access that same tag simply by running `soup.title`. Therefore, running `soup.title.text` returns a string equal to `title.text`.

Listing 16.22 Accessing the title's text attribute from soup

```
assert soup.title.text == title.text
```

In this same manner, we can access the body of our document by running `soup.body`. Next, we output all the text in the body of our HTML.

Listing 16.23 Accessing the body's text attribute from soup

```
body = soup.body
print(body.text)

Data Science is FunParagraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Here is a link to Data Science BookcampCommon Data Science
LibrariesNumPySciPyPandasScikit-Learn
```

Our output is an aggregation of all the text in the body. This text blob includes all headers, bullet points, and paragraphs. It is virtually unreadable. Rather than outputting

all the text, we should instead narrow the scope of our output. Let's print the text of just the first paragraph by printing `body.p.text`. Alternatively, printing `soup.p.text` generates the same output.

Listing 16.24 Accessing the text of the first paragraph

```
assert body.p.text == soup.p.text
print(soup.p.text)

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
```

Accessing `body.p` returns the first paragraph in `body`. How do we access the remaining two paragraphs? Well, we can utilize the `find_all` method. Running `body.find_all('p')` returns a list of all the paragraph tags in the body.

Listing 16.25 Accessing all paragraphs in the body

```
paragraphs = body.find_all('p')
for i, paragraph in enumerate(paragraphs):
    print(f"\nPARAGRAPH {i}:\n")
    print(paragraph.text)

PARAGRAPH 0:
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

PARAGRAPH 1:
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

PARAGRAPH 2:
Here is a link to Data Science Bookcamp
```

Similarly, we access our list of bullet points by running `body.find_all('li')`. Let's utilize `find_all` to print all the bulleted libraries in the body.

Listing 16.26 Accessing all bullet points in the body

```
print([bullet.text for bullet
      in body.find_all('li')])

['NumPy', 'Scipy', 'Pandas', 'Scikit-Learn']
```

The `find` and `find_all` methods allow us to search the elements by tag type and attribute. Suppose we wish to access an element with a unique ID of `x`. To search on that attribute ID, we simply need to execute `find(id='x')`. With this in mind, let's output the text of the final paragraph whose assigned ID is `paragraph_2`.

Listing 16.27 Accessing a paragraph by ID

```
paragraph_2 = soup.find(id='paragraph_2')
print(paragraph_2.text)
```

Here is a link to Data Science Bookcamp

The contents of `paragraph_2` include a web link to *Data Science Bookcamp*. The actual URL is stored in the `href` attribute. Beautiful Soup permits us to access any attribute using the `get` method. Thus, running `paragraph_2.get('id')` returns *paragraph_2*. Running `paragraph_2.a.get('href')` returns the URL; let's print it.

Listing 16.28 Accessing an attribute in a tag

```
assert paragraph_2.get('id') == 'paragraph_2'
print(paragraph_2.a.get('href'))

https://www.manning.com/books/data-science-bookcamp
```

All attribute IDs have unique values assigned to them in our HTML. However, not all of our attributes are unique. For instance, two of our three division elements share the `class` attribute of `text`. Meanwhile, the third division element contains a unique class that's set to `empty`. Running `body.find_all('div')` returns all three division elements. How do we obtain just those two divisions where the class is set to `text`? We just need to run `body.find_all('div', class_='text')`. The added `class_` parameter limits our results to those divisions where the class is set appropriately. Listing 16.29 searches for these divisions and outputs their text contents.

NOTE Why do we run `find_all` on `class_` rather than `class`? Well, in Python, the `class` keyword is a restricted identifier, which is used to define novel classes. Beautiful Soup allows for a special `class_` parameter to get around this keyword restriction.

Listing 16.29 Accessing divisions by their shared class attribute

```
for division in soup.find_all('div', class_='text'):
    id_ = division.get('id')
```

```

print(f"\nDivision with id '{id_}':")
print(division.text)

Division with id 'paragraphs':
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Here is a link to Data Science Bookcamp

Division with id 'list':
Common Data Science LibrariesNumPyScipyPandasScikit-Learn

```

So far, we've used BeautifulSoup to access elements in the HTML. However, the library also allows us to edit individual elements. For example, given a tag object, we can delete that object by running `tag.decompose()`. The `decompose` method removes that element from all our data structures, including `soup`. Thus, calling `body.find(id='paragraph 0').decompose()` will remove all traces of the first paragraph. Also, calling `soup.find(id='paragraph 1').decompose()` will delete the second paragraph from both the `soup` and `body` objects. After these deletions, only the third paragraph will remain. Let's confirm.

Listing 16.30 Paragraph deletion with BeautifulSoup

```

body.find(id='paragraph 0').decompose()
soup.find(id='paragraph 1').decompose()
print(body.find(id='paragraphs').text)

```

Here is a link to Data Science Bookcamp

The `decompose` method deletes the paragraph from all nested tag objects. Deleting the paragraph from `soup` also deletes it from `body`, and vice versa.

Additionally, we're able to insert new tags into the HTML. Suppose we wish to insert a new paragraph into our final empty division. To do so, we must first create a new paragraph element. Running `soup.new_tag('p')` returns an empty paragraph Tag object.

Listing 16.31 Initializing an empty paragraph Tag

```

new_paragraph = soup.new_tag('p')
print(new_paragraph)

<p></p>

```

Next, we must update the initialized paragraph's text by assigning it to `new_paragraph.string`. Running `new_paragraph.string = x` sets the paragraph's text to equal `x`.

Listing 16.32 Updating the text of an empty paragraph

```
new_paragraph.string = "This paragraph is new"  
print(new_paragraph)  
  
<p>This paragraph is new</p>
```

Finally, we append the updated `new_paragraph` to an existing Tag object. Given two Tag objects, `tag1` and `tag2`, we can insert `tag1` into `tag2` by running `tag2.append(tag1)`. Thus, running `soup.find(id='empty').append(new_paragraph)` should append the paragraph to the empty division. Let's update our HTML and then confirm the changes by rendering the updated results (figure 16.8).

Data Science Bookcamp'. Below that is a section titled 'Common Data Science Libraries' with a bulleted list: NumPy, Scipy, Pandas, and Scikit-Learn. At the bottom of the page is a single paragraph: 'This paragraph is new'." data-bbox="178 383 534 557"/>

Data Science is Fun
Here is a link to [Data Science Bookcamp](#)

Common Data Science Libraries

- NumPy
- Scipy
- Pandas
- Scikit-Learn

This paragraph is new

Figure 16.8 A rendered HTML document. The document has been edited, two of the three original paragraphs have been removed, and a new paragraph has been inserted.

Listing 16.33 Paragraph insertion with BeautifulSoup

```
soup.find(id='empty').append(new_paragraph)  
render(soup.prettify())
```

Common BeautifulSoup methods

- `soup = bs(html_contents)`—Initializes a BeautifulSoup object from the HTML elements in the parsed `html_contents`.
- `soup.prettify()`—Returns the parsed HTML document in a clean, easily readable format.
- `title = soup.title`—Returns a Tag object associated with the title element of a parsed document.
- `title = soup.find('title')`—Returns a Tag object associated with the title element of a parsed document.
- `tag_object = soup.find('element_tag')`—Returns a Tag object associated with the first HTML element demarcated by the specified `element_tag` tag.

- `tag_objects = soup.find_all('element_tag')`—Returns a list of all Tag objects demarcated by the specified `element_tag` tag.
- `tag_object = soup.find(id='unique_id')`—Returns a Tag object that contains the specified unique `id` attribute.
- `tag_objects = soup.find_all('element_tag', class_='category_class')`—Returns a list of Tag objects that are demarcated by the specified `element_tag` tag and that contain the specified `class` attribute.
- `tag_object = soup.new_tag('element_tag')`—Creates a new Tag object whose HTML element type is specified by the `element tag`.
- `tag_object.decompose()`—Deletes the Tag object from `soup`.
- `tag_object.append(tag_object2)`—Given two Tag objects, `tag_object` and `tag_object2`, inserts `tag_object2` into `tag_object`.
- `tag_object.text`—Returns all visible text in a Tag object.
- `tag_object.get('attribute')`—Returns an HTML attribute that has been assigned to the Tag object.

16.3 Downloading and parsing online data

The Beautiful Soup library allows us to easily parse, analyze, and edit HTML documents. In most cases, these documents must be downloaded directly from the web. Let's briefly review the procedure for downloading HTML files using Python's built-in `urllib` module. We start by importing the `urlopen` function from `urllib.request`.

NOTE The `urlopen` function is sufficient when downloading a single HTML document from a single, unsecured online page. However, for more complicated downloads, you should consider using the external Requests library (<https://requests.readthedocs.io>).

Listing 16.34 Importing the `urlopen` function

```
from urllib.request import urlopen
```

Given the URL of an online document, we can download the associated HTML contents by running `urlopen(url).read()`. Next, we use `urlopen` to download the Manning website for this book. Then we print the first 1,000 characters of the downloaded HTML.

WARNING The following code will only run with a valid internet connection. Also, the downloaded HTML may change with alterations to the website.

Listing 16.35 Downloading an HTML document

```
url = "https://www.manning.com/books/data-science-bookcamp"  
html_contents = urlopen(url).read()  
print(html_contents[:1000])
```

The `urlopen` function establishes a network connection with the specified URL. That connection is tracked using a special `URLopener` object. Calling the object's `read` method downloads text from the established connection.

```
b' \n<!DOCTYPE html>\n<!--[if lt IE 7 ]> <html lang="en" class="no-js ie6\nie"> <![endif]-->\n<!--[if IE 7 ]>      <html lang="en" class="no-js ie7\nie"> <![endif]-->\n<!--[if IE 8 ]>      <html lang="en" class="no-js ie8\nie"> <![endif]-->\n<!--[if IE 9 ]>      <html lang="en" class="no-js ie9\nie"> <![endif]-->\n<!--[if (gt IE 9) | !(IE)]><!--> <html lang="en"\nclass="no-js"><!--<![endif]-->\n<head>\n<title>Manning | Data Science Bookcamp</title>\n<n\n<meta name="msapplication-TileColor" content="#343434"/>\n<meta name="msapplication-square70x70logo" content="/assets/favicon/windows-\n    small-tile-6f6b7c9200a7af9169e488a11d13a7d3.png"/>\n<meta name="msapplication-square150x150logo"\n    content="/assets/favicon/windows-medium-tile-\n    8fae4270fe3f1a6398f15015221501fb.png"/>\n<meta name="msapplication-wide310x150logo" content="/assets/favicon/windows-\n    wide-tile-a856d33fb5e508f52f09495e2f412453.png"/>\n<meta name="msapplication-square310x310logo"\n    content="/assets/favicon/windows-large-tile-072d5381c2c83afa'
```

Let's extract the title from our messy HTML using BeautifulSoup.

Listing 16.36 Accessing the title with BeautifulSoup

```
soup = bs(html_contents)\nprint(soup.title.text)\n\nManning | Data Science Bookcamp
```

Using our soup object, we can further analyze the page. For instance, we can extract the division that contains an *about the book* header to print a description of this book.

WARNING Online HTML is continually updated. Future updates to the Manning site may cause the following code to malfunction. Readers who encounter differences between expected and actual outputs are encouraged to manually explore the HTML to extract the book description.

Listing 16.37 Accessing a description of this book

```
for division in soup.find_all('div'):\n    header = division.h2\n    if header is None:\n        continue\n\n    if header.text.lower() == 'about the book':\n        print(division.text)\n\nabout the book
```

Data Science Bookcamp is a comprehensive set of challenging projects carefully designed to grow your data science skills from novice to master. Veteran data scientist Leonard Apeltsin sets five increasingly difficult exercises that test your abilities against the kind of problems you'd encounter in the real world. As you solve each challenge, you'll acquire and expand the data science and Python skills you'll use as a professional

data scientist. Ranging from text processing to machine learning, each project comes complete with a unique, downloadable data set and a fully explained step-by-step solution. Because these projects come from Dr. Apeltsin's vast experience, each solution highlights the most likely failure points along with practical advice for getting past unexpected pitfalls. When you wrap up these five awesome exercises, you'll have a diverse, relevant skill set that's transferable to working in industry.

We are now ready to use Beautiful Soup to parse job postings as part of our case study solution.

Summary

- HTML documents are composed of nested elements that provide auxiliary information about the text. Most elements are defined by a start tag and an end tag.
- The text in some elements is intended to be rendered in a browser. Traditionally, this rendered information is nested in the *body* element of the document. Other non-rendered texts (such as the document's title) are nested in the document's *head* element.
- Attributes can be inserted into HTML start tags to track additional tag information. Unique *id* attributes can help distinguish tags of the same type. Furthermore, *class* attributes can be used to track elements by category. Unlike the unique *id*, the *class* attribute can be shared by multiple elements in the same category.
- Manually extracting text from HTML is difficult to do in basic Python. Fortunately, the Beautiful Soup library simplifies the text-extraction process. Beautiful Soup allows us to query elements by tag type and assigned attribute values. Furthermore, the library permits us to edit the underlying HTML.
- Using Python's built-in *urlopen* function, we can download HTML files directly from the web. Then we can analyze the text in these files using Beautiful Soup.

Case study 4 solution

This section covers

- Parsing text from HTML
- Computing text similarities
- Clustering and exploring large text datasets

We have downloaded thousands of job postings by searching on this book's table of contents for case studies 1 through 4 (see the problem statement for details). Besides the downloaded postings, we also have at our disposal two text files: resume.txt and table_of_contents.txt. The first file contains a resume draft, and the second contains the truncated table of contents used to query for job listing results. Our goal is to extract common data science skills from the downloaded job postings. Then we'll compare these skills to our resume to determine which skills are missing. We will do so as follows:

- 1 Parse all text from the downloaded HTML files.
- 2 Explore the parsed output to learn how job skills are described in online postings. We'll pay particular attention to whether certain HTML tags are more associated with skill descriptions.
- 3 Attempt to filter any irrelevant job postings from our dataset.
- 4 Cluster job skills based on text similarity.

- 5 Visualize the clusters using word clouds.
- 6 Adjust clustering parameters, if necessary, to improve the visualized output.
- 7 Compare the clustered skills to our resume to uncover missing skills.

WARNING Spoiler alert! The solution to case study 4 is about to be revealed. We strongly encourage you to try to solve the problem before reading the solution. The original problem statement is available for reference at the beginning of the case study.

17.1 Extracting skill requirements from job posting data

We begin by loading all the HTML files in the job_postings directory. We store the contents of these files in an `html_contents` list.

WARNING Be sure to manually unzip the compressed `job_postings.zip` directory before executing the following code.

Listing 17.1 Loading HTML files

```
import glob
html_contents = []

for file_name in sorted(glob.glob('job_postings/*.html')):
    with open(file_name, 'r') as f:
        html_contents.append(f.read())

print(f"We've loaded {len(html_contents)} HTML files.")

We've loaded 1458 HTML files.
```

We use the Python 3 `glob` module to obtain filenames with HTML extensions in the `job_postings` directory. These filenames are sorted to maintain output consistency across readers' personal machines. This ensures that the first two sampled files remain the same for all the readers.

Each of our 1,458 HTML files can be parsed using Beautiful Soup. Let's execute the parsing and store the parsed results in a `soup_objects` list. We also confirm that each parsed HTML file contains a title and a body.

Listing 17.2 Parsing HTML files

```
from bs4 import BeautifulSoup as bs

soup_objects = []
for html in html_contents:
    soup = bs(html)
    assert soup.title is not None
    assert soup.body is not None
    soup_objects.append(soup)
```

Each parsed HTML file contains a title and a body. Are there any duplicates across the titles or bodies of these files? We can find out by storing all title text and body text in two columns in a Pandas table. Calling the Pandas `describe` method will reveal the presence of any duplicates in the text.

Listing 17.3 Checking title and body texts for duplicates

```
import pandas as pd
html_dict = {'Title': [], 'Body': []}

for soup in soup_objects:
    title = soup.find('title').text
    body = soup.find('body').text
    html_dict['Title'].append(title)
    html_dict['Body'].append(body)

df_jobs = pd.DataFrame(html_dict)
summary = df_jobs.describe()
print(summary)

Title \
count          1458
unique         1364
top      Data Scientist - New York, NY
freq            13

Body
count          1458
unique         1458
top      Data Scientist - New York, NY 10011\nAbout the...
freq             1
```

1,364 of the 1,458 titles are unique. The remaining 94 titles are duplicates. The most common title is repeated 13 times: it is for a data scientist position in New York. We can easily verify that all the duplicate titles correspond to unique body text. All 1,458 bodies are unique, so none of the job postings occur more than once, even if some postings share a common generic title.

We've confirmed that no duplicates are present in the HTML. Now, let's explore the HTML content in more detail. The goal of our exploration is to determine how job skills are described in the HTML.

17.1.1 Exploring the HTML for skill descriptions

We start our exploration by rendering the HTML at index 0 of `html_contents` (figure 17.1).

Listing 17.4 Rendering the HTML of the first job posting

```
from IPython.core.display import display, HTML
assert len(set(html_contents)) == len(html_contents)
display(HTML(html_contents[0]))
```

The rendered job posting is for a data science position. The posting starts with a brief position overview, from which we learn that the job entails drawing insights from government data. The various required skills include model building, statistics, and

Data Scientist - Beavercreek, OH

Data Scientist

Position Overview:

Centauri is looking for a detail oriented, motivated, and organized Data Scientist to work as part of a team to clean, analyze, and produce insightful reporting on government data. The ideal candidate is adept at using large data sets to find trends for intelligence reporting and will be proficient in process optimization and using models to test the effectiveness of different courses of action. They must have strong experience using a variety of data mining/data analysis methods, using a variety of data tools, building and implementing models, using/creating algorithms and producing easily understood visuals to represent findings. Candidate will work closely with Data Managers and stakeholders to tailor their analysis to answer key questions. The candidate must have a strong understanding of Geographic Information Systems (GIS) and statistical analysis.

Responsibilities:

- Use statistical research methods to analyze datasets produced through multiple sources of intelligence production
- Mine and analyze data from databases to answer key intelligence questions
- Assess the effectiveness and accuracy of new data sources and data gathering techniques
- Develop custom data models and algorithms to apply to data sets
- Use predictive modeling to produce reporting about future trends based on historical data
- Spatially analyze geographic data using GIS tools
- Visualize findings in easily understood graphics and aesthetically appealing finished reports

Qualifications for Data Scientist:

- Experience using statistical computer languages (R, Python, SQL, etc.) to manipulate data and draw insights from large data sets
- Experience in basic visualization methods, especially using tools such as Tableau, ggplot, and matplotlib
- Knowledge of a variety of machine learning techniques (clustering, decision tree learning, artificial neural networks, etc.) and their real-world advantages/drawbacks
- Knowledge of advanced statistical techniques and concepts (regression, properties of distributions, statistical tests and proper usage, etc.) and experience with applications

Figure 17.1 The rendered HTML for the first job posting. The initial paragraph summarizes the data science job. The paragraph is followed by lists of bullet points, each containing a skill that is required to get the job.

visualization. These skills are further elaborated in the two bolded subsections: Responsibilities and Qualifications. Each subsection is composed of multiple single-sentence bullet points. The bullets are varied in their content: responsibilities include statistical method usage (bullet 1), future trend discovery (bullet 5), spatial analysis of geographic data (bullet 6), and aesthetically appealing visualization (bullet 7). Additionally, the bulleted qualifications cover computer languages such as R or Python (bullet 1), visualization tools such as Matplotlib (bullet 2), machine learning techniques including clustering (bullet 3), and knowledge of advanced statistical concepts (bullet 4).

It's worth noting that the qualifications are not that different from the responsibilities. Yes, the qualifications focus on tools and concepts, while the responsibilities are more attuned to actions on the jobs; but in a way, their bullet points are interchangeable. Each bullet describes a skill that an applicant must have to perform well at the job. Thus, we can subdivide `html_contents[0]` into two conceptually different parts:

- An initial job summary
- A list of bulleted skills required to get the job

Is the next job posting structured in a similar manner? Let's find out by rendering `html_contents[1]` (figure 17.2).

Data Scientist - Seattle, WA 98101

Are you interested in being a part of an Artificial Intelligence Marketing (AIM) company that is transforming how B2C enterprises engage with their customers; improving customer experience, marketing throughput and for the first time directly optimizing key business KPIs? Do you want to join a startup company backed by the top firms in the venture capital and SaaS industries? Would you like to be part of a company that prides itself on being a meritocracy, where passion, innovation, integrity, and our customers are at the heart of all that we do? Then, consider joining us at Amplero, an Artificial Intelligence Marketing company that leverages machine learning and multi-armed bandit experimentation to dynamically test thousands of permutations to adaptively optimize every customer interaction and maximize customer lifetime value and loyalty. We are growing our customer base and are looking for Data Scientists to join our innovative and energetic team! This is a unique opportunity to both drive innovations for our technology and to realize their impact as you work closely with our client engagement teams to best leverage our scientific capabilities within the Amplero product for marketing optimization and customer insights.

As an Amplero Data Scientist you would:

- Interface with our internal engagement teams and clients to understand business questions, and perform analytical "deep dives" to develop relevant and interpretive insights in support of our client engagements
- Smartly leverage appropriate technologies to answer tough questions or understand root causes of unexpected outcomes and statistical anomalies
- Develop analysis tools which will influence both our products and clients; including python pipelines focused on the productization of data science and insights tools for marketing performance and optimization
- Feature generation and selection from a wide variety of raw data types including time series and graphs
- Work with the Amplero Product Team to provide ongoing feedback to the features and priorities most aligned with our clients' current and future needs to inform the product roadmap, test product hypotheses as well as to help plan the product lifecycle

We'd love to hear from you if:

- You're an expert with data analysis and visualization tools including Python (including NumPy, SciPy, Pandas, scikit-learn) and other packages that enable data mining and machine learning
- You have a proven track record of applying data science to solve difficult real-world business problems
- You're familiar with areas of marketing data science where beyond-human scale, advanced experimentation and machine learning capabilities are used for achieving marketing performance, for example, DMP's in display advertising, Multivariate Testing, Statistical Significance Evaluation
- You've got excellent written and verbal communication skills for team and customer interactions - specifically, you're a genius at communicating results and the value of complex technical solutions to a non-technical audience

Figure 17.2 The rendered HTML for the second job posting. As in the first posting, the initial paragraph summarizes the data science job, and a list of bullet points describes the skills required to get the job.

Listing 17.5 Rendering the HTML of the second job posting

```
display(HTML(html_contents[1]))
```

The job posting is for a data science position in an AI marketing company. The structure of the posting is similar to that of `html_contents[0]`: the job is summarized in the post's initial paragraph, and then the required skills are presented in bullet points. These bulleted skills are varied in terms of technical requirements and details. For example, the fourth bullet from the bottom calls for expertise in the Python data science stack (NumPy, SciPy, Pandas, scikit-learn), the next bullet requires a track record of solving difficult real-world business problems, and the final bullet calls for excellent written and verbal communication skills. These three bulleted skills are very different. The difference is intentional—the author of the posting is emphasizing the diverse requirements needed to obtain the job. Thus, the bullet points in `html_contents[0]` and `html_contents[1]` serve a singular purpose: they offer us brief, sentence-length descriptions of unique skills required for each position.

Do these types of bulleted skill descriptions appear in other job posts? Let's find out. First we'll extract the bullets from each of our parsed HTML files. As a reminder, a bullet point is represented by the HTML tag ``. Any bulleted file contains multiple such tags; thus, we can extract a list of bullet points from a soup object by calling

`soup.find_all('li')`. Next, we'll iterate over our `soup_objects` list and extract all bullets from each element of that list. We store these results in a `Bullets` column in our existing `df_jobs` table.

Listing 17.6 Extracting bullets from the HTML

```
df_jobs['Bullets'] = [[bullet.text.strip()           ←
                      for bullet in soup.find_all('li')]
                      for soup in soup_objects]
```

Strips the line break from each bullet to avoid printing the line breaks in our later investigations

The bullets in each job posting are stored in `df_jobs.Bullets`. However, it is possible that some (or most) of the postings don't include any bullets. What percentage of job postings actually contain bulleted text? We need to find out! If that percentage is too low, further bullet analysis is not worth our time. Let's measure the percentage.

Listing 17.7 Measuring the percent of bulleted postings

```
bulleted_post_count = 0
for bullet_list in df_jobs.Bullets:
    if bullet_list:
        bulleted_post_count += 1

percent_bulleted = 100 * bulleted_post_count / df_jobs.shape[0]
print(f"{percent_bulleted:.2f}% of the postings contain bullets")

90.53% of the postings contain bullets
```

90% of the job postings contain bullets. Do all (or most) of these bullets focus on skills? We currently don't know. However, we can better gauge the contents of the bullet points by printing the top-ranked words in their text. We can rank these words by occurrence count; alternatively, we can carry out the ranking using term frequency-inverse document frequency (TFIDF) values rather than raw counts. As discussed in section 15, such TFIDF rankings are less likely to contain irrelevant words.

Next, we rank the words using summed TFIDF values. First we compute a TFIDF matrix in which rows correspond to individual bullets. Then we sum across the rows of the matrix: these sums are used to rank the words, which correspond to matrix columns. Finally, we check the top five ranked words for skill-related terminology.

Listing 17.8 Examining the top-ranked words in the HTML bullets

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

def rank_words(text_list):
    vectorizer = TfidfVectorizer(stop_words='english')           ←
    tfidf_matrix = vectorizer.fit_transform(text_list).toarray()
    df = pd.DataFrame({'Words': vectorizer.get_feature_names(),
```

Returns a sorted Pandas table of top-ranked words

```

        'Summed TFIDF': tfidf_matrix.sum(axis=0)})    ↪
sorted_df = df.sort_values('Summed TFIDF', ascending=False)
return sorted_df

all_bullets = []
for bullet_list in df_jobs.Bullets:
    all_bullets.extend(bullet_list)

sorted_df = rank_words(all_bullets)
print(sorted_df[:5].to_string(index=False))

```

Words	Summed TFIDF
experience	878.030398
data	842.978780
skills	440.780236
work	371.684232
ability	370.969638

Terms such as *skills* and *ability* appear among the top five bulleted words. There's reasonable evidence that the bullets correspond to individual job skills. How do these bulleted words compare to the remaining words in each job posting? Let's find out. We iterate over the body of each posting and delete any bulleted lists using Beautiful Soup's decompose method. Then we extract the remaining body text and store it in a `non_bullets` list. Finally, we apply our `rank_words` function to that list and display the top five non-bullet words.

Listing 17.9 Examining the top-ranked words in the HTML bodies

```

non_bullets = []
for soup in soup_objects:
    body = soup.body
    for tag in body.find_all('li'):
        tag.decompose()    ↪
    non_bullets.append(body.text)

sorted_df = rank_words(non_bullets)
print(sorted_df[:5].to_string(index=False))


```

Words	Summed TFIDF
data	99.111312
team	39.175041
work	38.928948
experience	36.820836
business	36.140488

Alternatively, calling
body.find_all('ul') will
achieve the same result.

The words *skills* and *ability* are no longer present in the ranked output. They have been replaced by the words *business* and *team*. Thus, the non-bulleted text appears to be less skill oriented than the bullet contents. However, it's still interesting to note that certain top-ranked words are shared between `bullets` and `non_bullets`: these words are *data*, *experience*, and *work*. Strangely, the words *scientist* and *science* are missing

from the list. Do some posts pertain to data-driven jobs that aren't directly data science jobs? Let's actively explore this possibility. We start by iterating over all the titles across all jobs and checking if each title mentions a data science position. Then we measure the percentage of jobs where the terms *data science* and *data scientist* are missing from the titles. Finally, we print a sample of 10 such titles for evaluation purposes.

NOTE As discussed in section 11, we match our terms to the title text using regular expressions.

Listing 17.10 Checking titles for references to data science positions

```
regex = r'Data Scien(ce|tist)'
df_non_ds_jobs = df_jobs[~df_jobs.Title.str.contains(regex, case=False)] ←

percent_non_ds = 100 * df_non_ds_jobs.shape[0] / df_jobs.shape[0]
print(f"{percent_non_ds:.2f}% of the job posting titles do not mention a "
      "data science position. Below is a sample of such titles:\n")
```

```
for title in df_non_ds_jobs.Title[:10]:
    print(title)
```

The Pandas str.contains methods can match a regular expression to column text. Passing case=False ensures that the match is not case sensitive.

64.81% of the job posting titles do not mention a data science position. Below is a sample of such titles:

```
Patient Care Assistant / PCA - Med/Surg (Fayette, AL) - Fayette, AL
Data Manager / Analyst - Oakland, CA
Scientific Programmer - Berkeley, CA
JD Digits - AI Lab Research Intern - Mountain View, CA
Operations and Technology Summer 2020 Internship-West Coast - Universal City, CA
Data and Reporting Analyst - Olympia, WA 98501
Senior Manager Advanced Analytics - Walmart Media Group - San Bruno, CA
Data Specialist, Product Support Operations - Sunnyvale, CA
Deep Learning Engineer - Westlake, TX
Research Intern, 2020 - San Francisco, CA 94105
```

Nearly 65% of the posting titles do not mention a data science position. However, from our sampled output, we can glean the alternative language that can be used to describe a data science job. A posting may call for a *data specialist*, a *data analyst*, or a *scientific programmer*. Furthermore, certain job postings are for research internships, which we can assume are data-centric. But not all sampled jobs are fully relevant: multiple postings are for management positions, which don't align with our immediate career goals. Management is a separate career track that requires its own unique set of skills. We should consider excluding management positions from our analysis.

More troublingly, the first posting on the list is for a *Patient Care Assistant* or *PCA*. Clearly, this posting has been crawled erroneously. Perhaps the crawling algorithm confused the job title with the PCA data-reduction technique. The erroneous posting contains skills that we lack and also have no interest in obtaining. These irrelevant skills pose a danger to our analysis and will serve as a source of noise if not removed. We can illustrate this danger by printing the first five bullets of df_non_ds_jobs[0].

Listing 17.11 Sampling bullets from a non-data science job

```
bullets = df_non_ds_jobs.Bullets.iloc[0]
for i, bullet in enumerate(bullets[:5]):
    print(f"{i}: {bullet.strip()}")


0: Provides all personal care services in accordance with the plan of
treatment assigned by the registered nurse
1: Accurately documents care provided
2: Applies safety principles and proper body mechanics to the performance
of specific techniques of personal and supportive care, such as ambulation
of patients, transferring patients, assisting with normal range of motions
and positioning
3: Participates in economical utilization of supplies and ensures that
equipment and nursing units are maintained in a clean, safe manner
4: Routinely follows and adheres to all policies and procedures
```

We are data scientists; our primary objective isn't patient care (index 0) or nursing equipment maintenance (index 4). We need to delete these skills from our dataset, but how? One approach is to use text similarity. We could compare the postings to our resume and delete the jobs that don't align with our resume content. Also, we should consider comparing the postings with this book's table of contents for added signal. Basically, we should evaluate the relevance of each job relative to both the resume and book material; this would allow us to filter the extraneous postings and retain only the most relevant jobs.

Alternatively, we could consider filtering the individual skills contained in the bullet points. Basically, we'd rank the individual bullet points rather than individual jobs. But there's a problem with this second approach. Imagine if we filter out all bullets that don't align with our resume or book material: the bullets that remain will cover skills that we already possess. This is counter to our goal of uncovering our missing skills using relevant data science postings. Instead, we should accomplish our goal as follows:

- 1** Obtain relevant job postings that partially match our existing skill set.
- 2** Examine which bullet points in these postings are missing from our existing skill set.

With this strategy in mind, we'll now filter the jobs by relevance.

17.2 Filtering jobs by relevance

Our goal is to evaluate job relevance using text similarity. We want to compare the text in each posting to our resume and/or the book's table of contents. In preparation, let's store our resume in a resume string.

Listing 17.12 Loading the resume

```
resume = open('resume.txt', 'r').read()
print(resume)
```

Experience

1. Developed probability simulations using NumPy.
2. Assessed online ad-clicks for statistical significance using Permutation testing.
3. Analyzed disease outbreaks using common clustering algorithms.

Additional Skills

1. Data visualization using Matplotlib.
2. Statistical analysis using SciPy.
3. Processing structured tables using Pandas.
4. Executing K-Means clustering and DBSCAN clustering using Scikit-Learn.
5. Extracting locations from text using GeonamesCache.
6. Location analysis and visualization using GeonamesCache and Cartopy.
7. Dimensionality reduction with PCA and SVD, using Scikit-Learn.
8. NLP analysis and text topic detection using Scikit-Learn.

In this same manner, we can store the table of contents in a `table_of_contents` string.

Listing 17.13 Loading the table of contents

```
table_of_contents = open('table_of_contents.txt', 'r').read()
```

Together, `resume` and `table_of_contents` summarize our existing skill set. Let's concatenate these skills into a single `existing_skills` string.

Listing 17.14 Combining skills into a single string

```
existing_skills = resume + table_of_contents
```

Our task is to compute the text similarity between each job posting and our existing skills. In other words, we want to compute all similarities between `df_jobs.Body` and `existing_skills`. This computation first requires that we vectorize all texts. We need to vectorize `df_jobs.Body` together with `existing_skills` to ensure that all vectors share the same vocabulary. Next, we combine our job posts and our skill string into a single list of texts and vectorize these texts using scikit-learn's `TfidfVectorizer` implementation.

Listing 17.15 Vectorizing our skills and the job-posting data

```
text_list = df_jobs.Body.values.tolist() + [existing_skills]
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(text_list).toarray()
```

Our vectorized texts are stored in a matrix format in `tfidf_matrix`. The final matrix row (`tfidf_matrix[-1]`) corresponds to our existing skill set, and all the other rows in (`tfidf_matrix[:-1]`) correspond to the job postings. Thus, we can easily compute the cosine similarities between the job postings and `existing_skills`. We simply need to

execute `tfidf_matrix[:-1] @ tfidf_matrix[-1]`: this matrix-vector product returns an array of cosine similarities. Listing 17.16 computes that `cosine_similarities` array.

NOTE You may wonder if it's worthwhile to visualize the distribution of rankings. The answer is yes! We will plot this distribution shortly to obtain valuable insights, but first we want to carry out a simple sanity check by printing the top-ranking job titles. Doing so will confirm that our hypothesis is correct and all the printed jobs are relevant.

Listing 17.16 Computing skill-based cosine similarities

```
cosine_similarities = tfidf_matrix[:-1] @ tfidf_matrix[-1]
```

The cosine similarities capture the text overlap between our existing skills and the posted jobs. Jobs with greater overlap are more relevant, and jobs with lesser overlap are less relevant. Thus, we can use cosine similarities to rank jobs by relevance. Let's carry out the ranking. First, we need to store the cosine similarities in a `Relevance` column of `df_jobs`. Then, we sort the table by `df_jobs.Relevance` in descending order. Finally, we print the 20 least relevant job titles in the sorted table and confirm whether these low-ranking jobs have anything to do with data science.

Listing 17.17 Printing the 20 least relevant jobs

```
df_jobs['Relevance'] = cosine_similarities
sorted_df_jobs = df_jobs.sort_values('Relevance', ascending=False)
for title in sorted_df_jobs[-20:].Title:
    print(title)

Data Analyst Internship (8 month minimum) - San Francisco, CA
Leadership and Advocacy Coordinator - Oakland, CA 94607
Finance Consultant - Audi Palo Alto - Palo Alto, CA
RN - Hattiesburg, MS
Configuration Management Specialist - Dahlgren, VA
Deal Desk Analyst - Mountain View, CA
Dev Ops Engineer AWS - Rockville, MD
Web Development Teaching Assistant - UC Berkeley (Berkeley) - Berkeley, CA
Scorekeeper - Oakland, CA 94612
Direct Care - All Experience Levels (CNA, HHA, PCA Welcome) - Norwell, MA 02061
Director of Marketing - Cambridge, MA
Certified Strength and Conditioning Specialist - United States
PCA - PCU Full Time - Festus, MO 63028
Performance Improvement Consultant - Los Angeles, CA
Patient Services Rep II - Oakland, CA
Lab Researcher I - Richmond, CA
Part-time instructor of Statistics for Data Science and Machine Learning - San
Francisco, CA 94105
Plant Engineering Specialist - San Pablo, CA
Page Not Found - Indeed Mobile
Director of Econometric Modeling - External Careers
```

Most of the printed jobs are completely irrelevant. Various extraneous employment opportunities include *Leadership and Advocacy Coordinator*, *Financial Consultant, RN* (registered nurse), and *Scorekeeper*. One of the job titles even reads *Page Not Found*, which indicates that the web page was not downloaded correctly. However, a few of the jobs are related to data science: for instance, one calls for a *Part-time instructor of Statistics and Data Science and Machine Learning*. Still, this job is not exactly what we're looking for. After all, our immediate goal is to practice data science, not teach it. We can discard the 20 lowest-ranking jobs in the sorted table. Now, for comparison's sake, let's print the 20 most relevant job titles in `sorted_df_jobs`.

Listing 17.18 Printing the 20 most relevant jobs

```
for title in sorted_df_jobs[:20].Title:  
    print(title)  
  
Chief Data Officer - Culver City, CA 90230  
Data Scientist - Beavercreek, OH  
Data Scientist Population Health - Los Angeles, CA 90059  
Data Scientist - San Diego, CA  
Data Scientist - Beavercreek, OH  
Senior Data Scientist - New York, NY 10018  
Data Architect - Raleigh, NC 27609  
Data Scientist (PhD) - Spring, TX  
Data Science Analyst - Chicago, IL 60612  
Associate Data Scientist (BS / MS) - Spring, TX  
Data Scientist - Streetsboro, OH 44241  
Data Scientist - Los Angeles, CA  
Sr Director of Data Science - Elkridge, MD  
2019-57 Sr. Data Scientist - Reston, VA 20191  
Data Scientist (PhD) - Intern - Spring, TX  
Sr Data Scientist. - Alpharetta, GA 30004  
Data Scientist GS 13/14 - Clarksburg, WV 26301  
Data Science Intern (BS / MS) - Intern - Spring, TX  
Senior Data Scientist - New York, NY 10038  
Data Scientist - United States
```

Almost all of the printed job titles are for data science jobs. Some jobs, such as *Chief Data Officer*, probably lie beyond our existing level of expertise, but the top-ranking jobs appear to be quite relevant to our data science career.

NOTE Based on its title, the position of *Chief Data Officer* appears to be a management position. As we stated earlier, a management position requires its own separate set of skills. However, if we output the job posting's body (`sorted_df_jobs.iloc[0].Body`), we immediately discover that the job isn't a management job at all! The company is simply searching for a highly experienced data scientist to cover all its data science needs. Sometimes job titles can be deceiving; glancing briefly over a title cannot fully substitute for carefully reading the body text.

Clearly, when `df_jobs.Relevance` is high, the associated job postings are relevant. As `df_jobs.Relevance` decreases, the associated jobs become less relevant. Thus, we can presume that there exists some `df_jobs.Relevance` cutoff that separates the relevant jobs from the non-relevant jobs. Let's try to identify that cutoff. We start by visualizing the shape of the sorted relevance distribution relative to rank. In other words, we plot `range(df_jobs.shape[0])` versus `sorted_df_jobs.Relevance` (figure 17.3). In the plot, we expect to see a relevance curve that's continuously decreasing; any sudden decreases of relevance in the curve indicate a separation between relevant and non-relevant jobs.

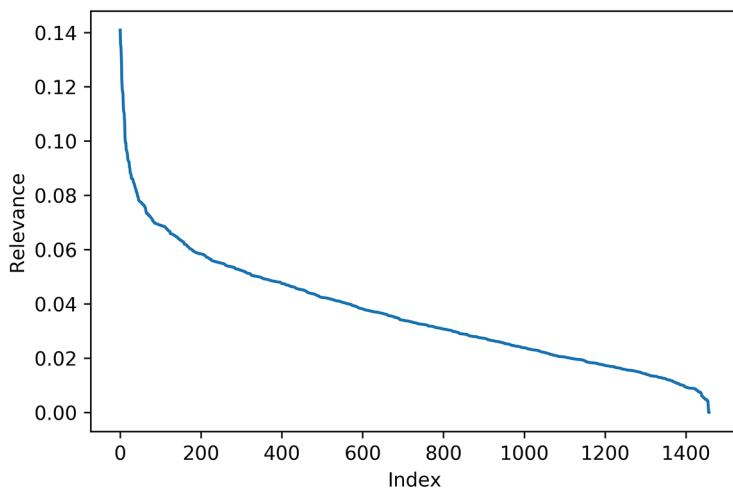


Figure 17.3 Ranked job posting indices plotted vs. relevance. Lower indices indicate higher relevance. The relevance is equal to the cosine similarity between each job and `existing_skills`. This relevance drops rapidly at an index of approximately 60.

Listing 17.19 Plotting job ranking vs. relevance

```
import matplotlib.pyplot as plt
plt.plot(range(df_jobs.shape[0]), sorted_df_jobs.Relevance.values)
plt.xlabel('Index')
plt.ylabel('Relevance')
plt.show()
```

Our relevance curve resembles a K-means elbow plot. Initially, the relevance drops rapidly. Then, at an x-value of approximately 60, the curve begins to level off. Let's emphasize this transition by drawing a vertical line through the x-position of 60 in our plot (figure 17.4).

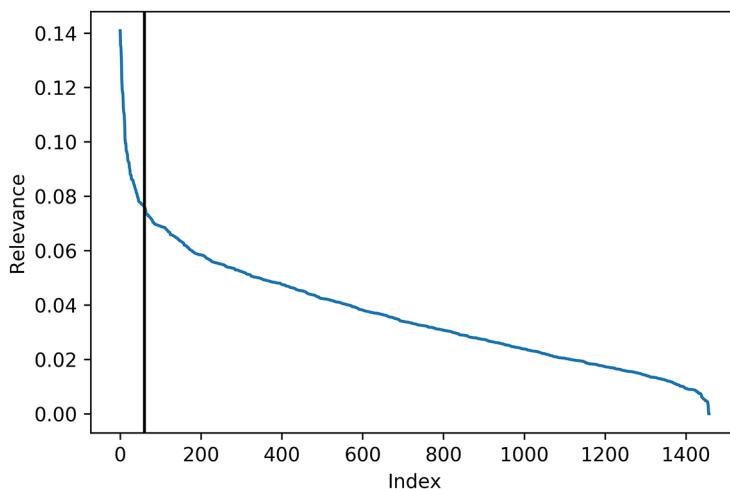


Figure 17.4 Ranked job posting indices are plotted vs. relevance. A vertical cutoff of 60 is also included in the plot. Indices below 60 correspond to much higher relevance values.

Listing 17.20 Adding a cutoff to the relevance plot

```
plt.plot(range(df_jobs.shape[0]), sorted_df_jobs.Relevance.values)
plt.xlabel('Index')
plt.ylabel('Relevance')
plt.axvline(60, c='k')
plt.show()
```

Our plot implies that the first 60 jobs are noticeably more relevant than all subsequent jobs. We'll now probe this implication. As we've already seen, the first 20 jobs are highly relevant. Based on our hypothesis, jobs 40 through 60 should be highly relevant as well. Next, we print `sorted_ds_jobs[40: 60].Title` for evaluation purposes.

Listing 17.21 Printing jobs below the relevance cutoff

```
for title in sorted_df_jobs[40: 60].Title.values:
    print(title)

Data Scientist III - Pasadena, CA 91101
Global Data Engineer - Boston, MA
Data Analyst and Data Scientist - Summit, NJ
Data Scientist - Generalist - Glendale, CA
Data Scientist - Seattle, WA
IT Data Scientist - Contract - Riverton, UT
Data Scientist (Analytic Consultant 4) - San Francisco, CA
Data Scientist - Seattle, WA
Data Science & Tagging Analyst - Bethesda, MD 20814
Data Scientist - New York, NY
```

Senior Data Scientist - Los Angeles, CA
 Principal Statistician - Los Angeles, CA
 Senior Data Analyst - Los Angeles, CA
 Data Scientist - Aliso Viejo, CA 92656
 Data Engineer - Seattle, WA
 Data Scientist - Digital Factory - Tampa, FL 33607
 Data Scientist - Grapevine, TX 76051
 Data Scientist - Bioinformatics - Denver, CO 80221
 EPIDEMIOLOGIST - Los Angeles, CA
 Data Scientist - Bellevue, WA

Almost all of the printed jobs are for data scientist/analyst positions. The only outlier is a posting for an epidemiologist, which probably appeared due to our stated experience of tracking disease epidemics. The outlier notwithstanding, the remaining jobs are highly relevant. Implicitly, the relevance should decrease when we print the next 20 job titles since they lie beyond the bounds of index 60. Let's verify if this is the case.

Listing 17.22 Printing jobs beyond the relevance cutoff

```
for title in sorted_df_jobs[60: 80].Title.values:
    print(title)

Data Scientist - Aliso Viejo, CA
Data Scientist and Visualization Specialist - Santa Clara Valley, CA 95014
Data Scientist - Los Angeles, CA
Data Scientist Manager - NEW YORK LOCATION! - New York, NY 10036
Data Science Intern - San Francisco, CA 94105
Research Data Analyst - San Francisco, CA
Sr Data Scientist (Analytic Consultant 5) - San Francisco, CA
Data Scientist, Media Manipulation - Cambridge, MA
Manager, Data Science, Programming and Visualization - Boston, MA
Data Scientist in Broomfield, CO - Broomfield, CO
Senior Data Scientist - Executive Projects and New Solutions - Foster City, CA
Manager of Data Science - Burbank California - Burbank, CA
Data Scientist Manager - Hiring in Burbank! - Burbank, CA
Data Scientists needed in NY - Senior Consultants and Managers! - New York, NY
10036
Data Scientist - Menlo Park, CA
Data Engineer - Santa Clara, CA
Data Scientist - Remote
Data Scientist I-III - Phoenix, AZ 85021
SWE Data Scientist - Santa Clara Valley, CA 95014
Health Science Specialist - San Francisco, CA 94102
```

A few of the job titles for postings 60 through 80 are noticeably less relevant. Some jobs are management positions, and one is a health science specialist position. Nonetheless, a majority of the jobs refer to data science/analyst roles outside the scope of health science or management. We can quickly quantify this observation using regular expressions. We define a percent_relevant_titles function, which returns the percent of non-management data science and analysis jobs in a data frame slice. Then we

apply that function to `sorted_df_jobs[60: 80]`. The output gives us a very simple alternative measure of relevance based on job post titles.

Listing 17.23 Measuring title relevance in a subset of jobs

```
import re
def percent_relevant_titles(df):
    regex_relevant = re.compile(r'Data (Scienc|Analyst)', ←
        flags=re.IGNORECASE)
    regex_irrelevant = re.compile(r'\b(Manage)', ←
        flags=re.IGNORECASE)
    match_count = len([title for title in df.Title
        if regex_relevant.search(title)
        and not regex_irrelevant.search(title)]) ←
    percent = 100 * match_count / df.shape[0]
    return percent

percent = percent_relevant_titles(sorted_df_jobs[60: 80])
print(f"Approximately {percent:.2f}% of job titles between indices "
    "60 - 80 are relevant")
```

Approximately 65.00% of job titles between indices 60 - 80 are relevant

Approximately two-thirds of the job titles in `sorted_df_jobs[60: 80]` are relevant. Although the job relevance has decreased beyond index 60, more than 50% of the titles still refer to data science jobs. Perhaps that percentage will drop if we sample the next 20 jobs across an index range of 80 to 100. Let's check.

Listing 17.24 Measuring title relevance in the next subset of jobs

```
percent = percent_relevant_titles(sorted_df_jobs[80: 100])
print(f"Approximately {percent:.2f}% of job titles between indices "
    "80 - 100 are relevant")
```

Approximately 80.00% of job titles between indices 80 - 100 are relevant

Nope! The data science title percentage rose to 80%. At what point will the percentage drop below 50%? We can easily find out! Let's iterate over `sorted_df_jobs[i: i + 20]` for all values of `i`. At every iteration, we compute the relevance percentage. Then we plot all the percentages (figure 17.5). We also plot a horizontal line at 50% to allow us to determine the index at which relevant job titles fall into the minority.

Listing 17.25 Plotting percent relevance across all title samples

```
def relevant_title_plot(index_range=20): ←
    percentages = []
    The function runs percent_relevant_titles across each consecutive
    slice of index_range jobs. Next, all the percentages are plotted. The
    index_range parameter is preset to 20. Later, we adjust that parameter value.
```

```

start_indices = range(df_jobs.shape[0] - index_range)
for i in start_indices:
    df_slice = sorted_df_jobs[i: i + index_range]
    percent = percent_relevant_titles(df_slice)
    percentages.append(percent)

plt.plot(start_indices, percentages)
plt.axhline(50, c='k')
plt.xlabel('Index')
plt.ylabel('% Relevant Titles')

relevant_title_plot()
plt.show()

```

Analyzes `sorted_df_jobs[i: i + index_range]`, where `i` ranges from 0 to the total posting count minus the index range

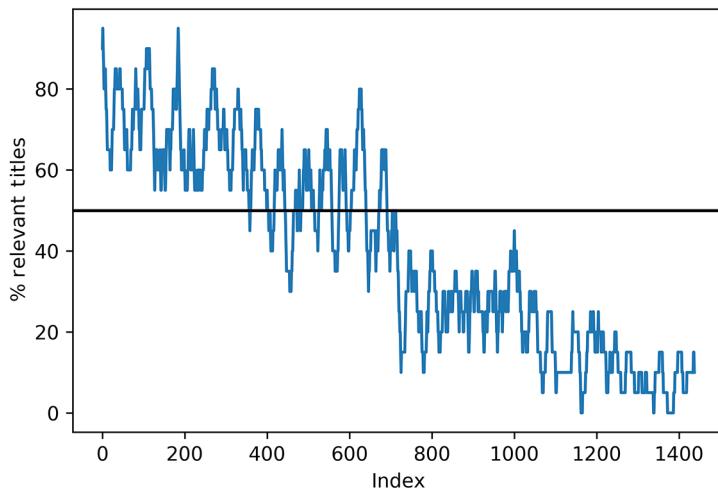


Figure 17.5 Ranked job posting indices plotted vs. title relevance. Title relevance is equal to the percent of data science titles across 20 consecutive job postings (starting at some index). A horizontal line demarcates 50% relevance. The relevance drops below 50% at an index of approximately 700.

The plot fluctuates with a high degree of variance. But despite the fluctuations, we can observe that the relevant data science titles drop below 50% at an index of around 700. Of course, it's possible that the cutoff of 700 is merely an artifact of our chosen index range. Will the cutoff still be present if we double our index range? We'll find out by running `relevant_title_plot(index_range=40)` (figure 17.6). We also plot a vertical line at index 700 to confirm that the percentage drops below 50% beyond that line.

Listing 17.26 Plotting percent relevance across an increased index range

```

relevant_title_plot(index_range=40)
plt.axvline(700, c='k')
plt.show()

```

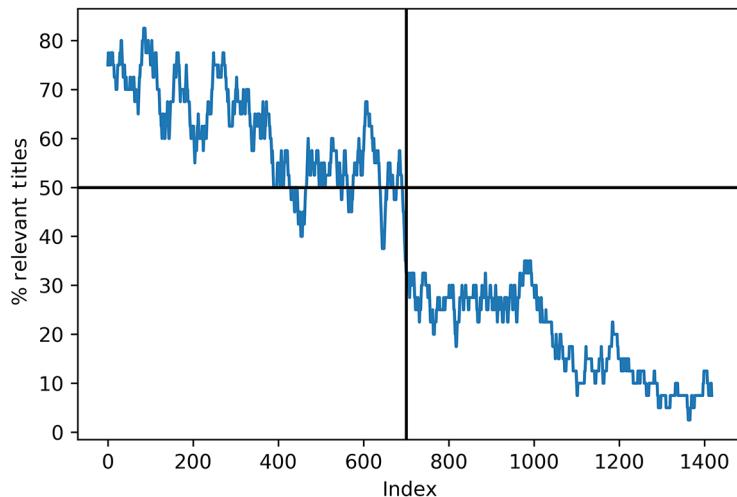


Figure 17.6 Ranked job-posting indices plotted vs. title relevance. Title relevance is equal to the percent of data science titles across 40 consecutive job postings (starting at some index). A horizontal line demarcates 50% relevance, and a vertical line demarcates an index of 700. Below that line, relevance drops to less than 50%.

Our updated plot continues to drop below 50% at an index cutoff of 700.

NOTE There's more than one way to approximate that cutoff. Suppose, for instance, that we simplify our regex to `r'Data (Science|Scientist)'`. We thus ignore all mentions of analysts or managers. Also, suppose we eliminate the use of index ranges and instead count the total number of data science titles appearing below each index. If we plot these simple results, we see a curve that levels off at an index of 700. Despite our simplifications, we achieve very similar results. In data science, there's frequently more than one path toward an insightful observation.

At this point, we face a choice between two relevance cutoffs. Our first cutoff, at index 60, is highly precise: most jobs below that cutoff are data science positions. However, the cutoff has limited recall: hundreds of data science jobs appear beyond an index of 60. Meanwhile, our second cutoff of 700 captures many more data science positions, but some irrelevant jobs also appear below the cutoff range. There's almost a 12-fold difference between the two relevance cutoffs. So, which cutoff do we choose? Do we prefer higher precision or higher recall? If we choose higher recall, will the noise hurt our analysis? If we choose higher precision, will the limited diversity of seen skills render our analysis incomplete? These are all important questions. Unfortunately, there's no immediate right answer. Higher precision at the expense of recall could potentially hurt us, and vice versa. What should we do?

How about trying both cutoffs? That way, we can compare the trade-offs and benefits of each! First, we'll cluster the skill sets from job postings below an index of 60. Then, we'll repeat our analysis for job postings below an index of 700. Finally, we'll integrate these two different analyses into a single, coherent conclusion.

17.3 Clustering skills in relevant job postings

Our aim is to cluster the skills in the 60 most relevant job postings. The skills in each posting are diverse and partially represented by bullet points. Thus we face a choice:

- Cluster the 60 texts in `sorted_df_jobs[:60].Body`.
- Cluster the hundreds of individual bullet points in `sorted_df_jobs[:60].Bullets`.

The second option is preferable for the following reasons:

- Our stated aim is to identify missing skills. The bullet points focus more on individual skills than the heterogeneous body of each posting.
- The short bullet points are easy to print and read. This is not the case for the larger postings. Thus, clustering by bullets allows us to examine each cluster by outputting a sample of the clustered bullet text.

We'll cluster the scraped bullets. We start by storing `sorted_df_jobs[:60].Bullets` in a single list.

Listing 17.27 Obtaining bullets from the 60 most relevant jobs

```
total_bullets = []
for bullets in sorted_df_jobs[:60].Bullets:
    total_bullets.extend(bullets)
```

How many bullets are in the list? Are any of the bullets duplicated? We can check by loading `total_bullets` into a Pandas table and then applying the `describe` method.

Listing 17.28 Summarizing basic bullet statistics

```
df_bullets = pd.DataFrame({'Bullet': total_bullets})
print(df_bullets.describe())

Bullet
count                1091
unique               900
top     Knowledge of advanced statistical techniques a...
freq                   9
```

The list contains 1,091 bullets. However, only 900 are unique—the remaining 91 bullets are duplicates. The most frequent duplicate is mentioned nine times. If we don't deal with this issue, it could affect our clustering. We should remove all duplicate texts before proceeding with our analysis.

NOTE Where do the duplicates originate? We can find out by tracing back a few duplicates to their original job posts. For brevity's sake, this analysis is not included in the book. However, you're encouraged to try it yourself. The output shows how certain companies reuse job templates for different jobs. Each template is modified for each position, but certain repeated bullet points remain. These repeated bullet points could bias our clusters toward company-specific skills and thus should be removed from `total_bullets`.

Next, we filter empty strings and duplicates from our bullet list. Then we vectorize the list using a TFIDF vectorizer.

Listing 17.29 Removing duplicates and vectorizing the bullets

```
→ total_bullets = sorted(set(total_bullets))
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(total_bullets)
num_rows, num_columns = tfidf_matrix.shape
print(f"Our matrix has {num_rows} rows and {num_columns} columns")
```

Converts `total_bullets` into a set to remove the 91 duplicates. We sort that set to ensure consistent ordering (and thus consistent output). Alternatively, we can drop the duplicates directly from our Pandas table by running `df_bullets.drop_duplicates(inplace=True)`.

We've vectorized our deduplicated bullet list. The resulting TFIDF matrix has 900 rows and over 2,000 columns; it thus contains over 1.8 million elements. This matrix is too large for efficient clustering. Let's dimensionally reduce the matrix using the procedure described in section 15: we'll shrink the matrix to 100 dimensions with SVD, and then we'll normalize the matrix.

Listing 17.30 Dimensionally reducing the TFIDF matrix

```
import numpy as np
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import normalize
np.random.seed(0)

def shrink_matrix(tfidf_matrix):
    svd_object = TruncatedSVD(n_components=100)
    shrunk_matrix = svd_object.fit_transform(tfidf_matrix)
    return normalize(shrunk_matrix)

shrunk_norm_matrix = shrink_matrix(tfidf_matrix)
```

Applies SVD to an inputted TFIDF matrix. The matrix is reduced to 100 dimensions, normalized, and returned.

We are nearly ready to cluster our normalized matrix using K-means. However, first we need to estimate K . Let's generate an elbow plot using mini-batch K-means, which is optimized for speed (figure 17.7).

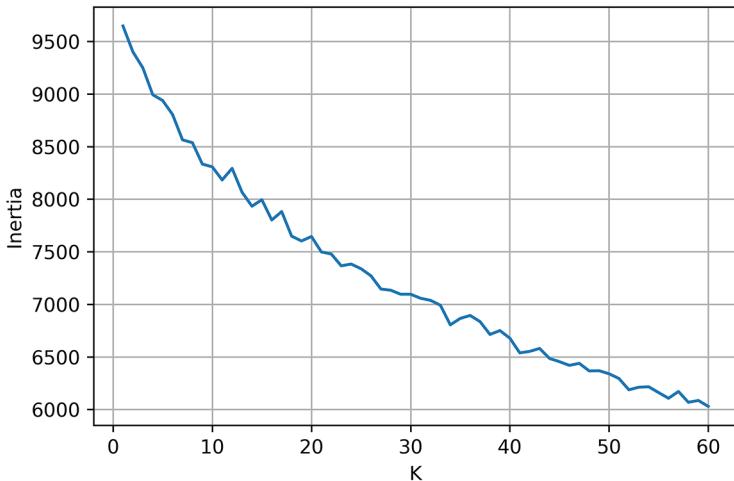


Figure 17.7 An elbow plot generated using mini-batch K-means, across K values ranging from 1 to 60. The precise location of the elbow is difficult to determine.

Listing 17.31 Plotting an elbow curve using mini-batch K-means

```
np.random.seed(0)
from sklearn.cluster import MiniBatchKMeans
def generate_elbow_plot(matrix):
    k_values = range(1, 61)                         ← Generates an elbow plot
    inertia_values = [MiniBatchKMeans(k).fit(matrix).inertia_   across an inputted data matrix
                     for k in k_values]           using mini-batch K-means
    plt.plot(k_values, inertia_values)
    plt.xlabel('K')                                  ← The number of
    plt.ylabel('Inertia')                            clusters ranges
    plt.grid(True)                                 ← from 1 to 60.
    plt.show()

generate_elbow_plot(shrunk_norm_matrix)
```

Plots grid lines to help us identify where the elbow lies on the x-axis

Our plotted curve decreases smoothly. The precise location of a bent elbow-shaped transition is difficult to spot: that curve drops rapidly at a K of 10 and then gradually bends into an elbow somewhere between a K of 10 and a K of 25. Which K value should we choose? 10, 25, or some value in between, such as 15 or 20? The right answer is not immediately clear. So why not try multiple values of K ? Let's cluster our data multiple times using K values of 10, 15, 20, and 25. Then we'll compare and contrast the results. If necessary, we'll consider choosing a different K for clustering. We'll start by grouping our job skills into 15 clusters.

NOTE Our aim is to investigate outputs for four different values of K . The order in which we generate the outputs is completely arbitrary. In this book,

we start with a K value of 15 because the resulting cluster count is not too large and not too small. This sets a nice baseline for the subsequent discussion of the outputs.

17.3.1 Grouping the job skills into 15 clusters

We execute K-means using a K of 15. Then we store the text indices and cluster IDs in a Pandas table. We also store the actual bullet text for easier accessibility. Finally, we utilize the Pandas groupby method to split the table by cluster.

Listing 17.32 Clustering bullets into 15 clusters

```
Tracks each clustered bullet's index
in clusters, cluster ID, and text
np.random.seed(0)
from sklearn.cluster import KMeans
def compute_cluster_groups(shrunk_norm_matrix, k=15,
                           bullets=total_bullets):
    cluster_model = KMeans(n_clusters=k)
    clusters = cluster_model.fit_predict(shrunk_norm_matrix)
    df = pd.DataFrame({'Index': range(clusters.size), 'Cluster': clusters,
                       'Bullet': bullets})
    return [df_cluster for _, df_cluster in df.groupby('Cluster')]

cluster_groups = compute_cluster_groups(shrunk_norm_matrix)
```

Executes K-means clustering on the input `shrunk_norm_matrix`. The `K` parameter is preset to 15. The function returns a list of Pandas tables, where each table represents a cluster. Clustered bullets are included in these tables; the bullets were passed in through an optional `bullets` parameter.

Each of our text clusters is stored as a Pandas table in the `cluster_groups` list. We can visualize the clusters using word clouds. In section 15, we defined a custom `cluster_to_image` function for word cloud visualization. The function took as input a cluster-specific Pandas table and returned a word cloud image. Listing 17.33 redefines that function and applies it to `cluster_groups[0]` (figure 17.8).

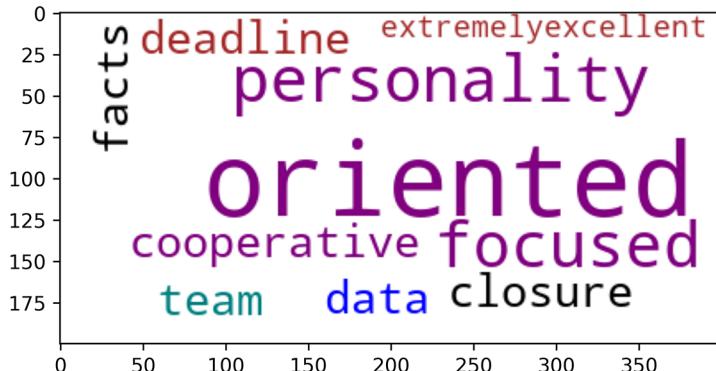


Figure 17.8 A word cloud generated for the cluster at index 0. The language in the word cloud is a little vague. It appears to be describing a focused, data-oriented personality.

NOTE Why should we redefine the function? Well, in section 15, `cluster_to_image` depended on a fixed TFIDF matrix and vocabulary list. In our current analysis, these parameters are not fixed—both the matrix and vocabulary will shift as we adjust our relevancy index. Thus, we need to update the function to allow for more dynamic input.

Listing 17.33 Visualizing the first cluster

Takes as input a `df_cluster` table and returns a word cloud image for the top `max_words` words corresponding to the cluster. The words are obtained from the inputted `vectorizer` class. They are ranked by summing over the rows of the inputted `tfidf_matrix`. When we extend our job threshold from 60 to 700, both `vectorizer` and `tfidf_matrix` must be adjusted accordingly.

```
from wordcloud import WordCloud
np.random.seed(0)

def cluster_to_image(df_cluster, max_words=10, tfidf_matrix=tfidf_matrix,
                     vectorizer=vectorizer):
    indices = df_cluster.Index.values
    summed_tfidf = np.asarray(tfidf_matrix[indices].sum(axis=0))[0]
    data = {'Word': vectorizer.get_feature_names(), 'Summed TFIDF':
            summed_tfidf}
    df_ranked_words = pd.DataFrame(data).sort_values('Summed TFIDF',
                                                       ascending=False)
    words_to_score = {word: score
                      for word, score in df_ranked_words[:max_words].values
                      if score != 0}
    cloud_generator = WordCloud(background_color='white',
                                 color_func=_color_func,
                                 random_state=1)
    wordcloud_image = cloud_generator.fit_words(words_to_score)
    return wordcloud_image

def _color_func(*args, **kwargs):
    return np.random.choice(['black', 'blue', 'teal', 'purple', 'brown'])

wordcloud_image = cluster_to_image(cluster_groups[0])
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.show()
```

Helper function to randomly assign one of five acceptable colors to each word

The language in the word cloud seems to be describing someone focused and data oriented, but it's a little vague. Perhaps we can learn more about the cluster by printing some sample bullets from `cluster_group[0]`.

NOTE We'll print a random sample of bullets. This should be sufficiently informative to understand the cluster. However, it's worth emphasizing that not all bullets are equal: some bullets are closer to their K-means cluster centroid and therefore more representative of the cluster. Thus, we can optionally sort the bullets based on their distance to the cluster mean. In this book, we bypass bullet ranking for brevity's sake, but you're encouraged to try ranking the bullets on your own.

Listing 17.34 Printing sample bullets from cluster 0

```

np.random.seed(1)
def print_cluster_sample(cluster_id):
    df_cluster = cluster_groups[cluster_id]
    for bullet in np.random.choice(df_cluster.Bullet.values, 5,
                                    replace=False):
        print(bullet)

print_cluster_sample(0)

Data-oriented personality
Detail-oriented
Detail-oriented – quality and precision-focused
Should be extremelyExcellent facts and data oriented
Data oriented personality

```

The printed bullets all use very similar language: they call for an employee who is detail oriented and data oriented. Linguistically, this cluster is legitimate. Unfortunately, it represents a skill that is difficult to grasp. Being detail oriented is a very general skill—it's hard to quantify, demonstrate, and learn. Ideally, the other clusters will contain more concrete technical skills.

Let's examine all 15 clusters simultaneously using word clouds. These word clouds are displayed across 15 subplots in a five-row-by-three-column grid (figure 17.9).

Listing 17.35 Visualizing all 15 clusters

```

→ def plot_wordcloud_grid(cluster_groups, num_rows=5, num_columns=3,
                         **kwargs):
    figure, axes = plt.subplots(num_rows, num_columns, figsize=(20, 15))
    cluster_groups_copy = cluster_groups[:]
    for r in range(num_rows):
        for c in range(num_columns):
            if not cluster_groups_copy:
                break
            df_cluster = cluster_groups_copy.pop(0)
            wordcloud_image = cluster_to_image(df_cluster, **kwargs)
            ax = axes[r][c]
            ax.imshow(wordcloud_image,
                      interpolation="bilinear")
            ax.set_title(f"Cluster {df_cluster.Cluster.iloc[0]}")
            ax.set_xticks([])
            ax.set_yticks([])

plot_wordcloud_grid(cluster_groups)
plt.show()

```

Plots the word clouds for each cluster in cluster_groups.
The word clouds are plotted in a num_rows by num_columns grid.

The ****kwargs** syntax allows us to pass additional parameters into the utilized `cluster_to_image` function. This way, we can modify both `vectorizer` and `tfidf_matrix` with ease.

Our 15 skill clusters show a diverse collection of topics. Some of the clusters are highly technical. For instance, cluster 7 fixates on external data science libraries such as scikit-learn, Pandas, NumPy, Matplotlib, and SciPy. The scikit-learn library clearly

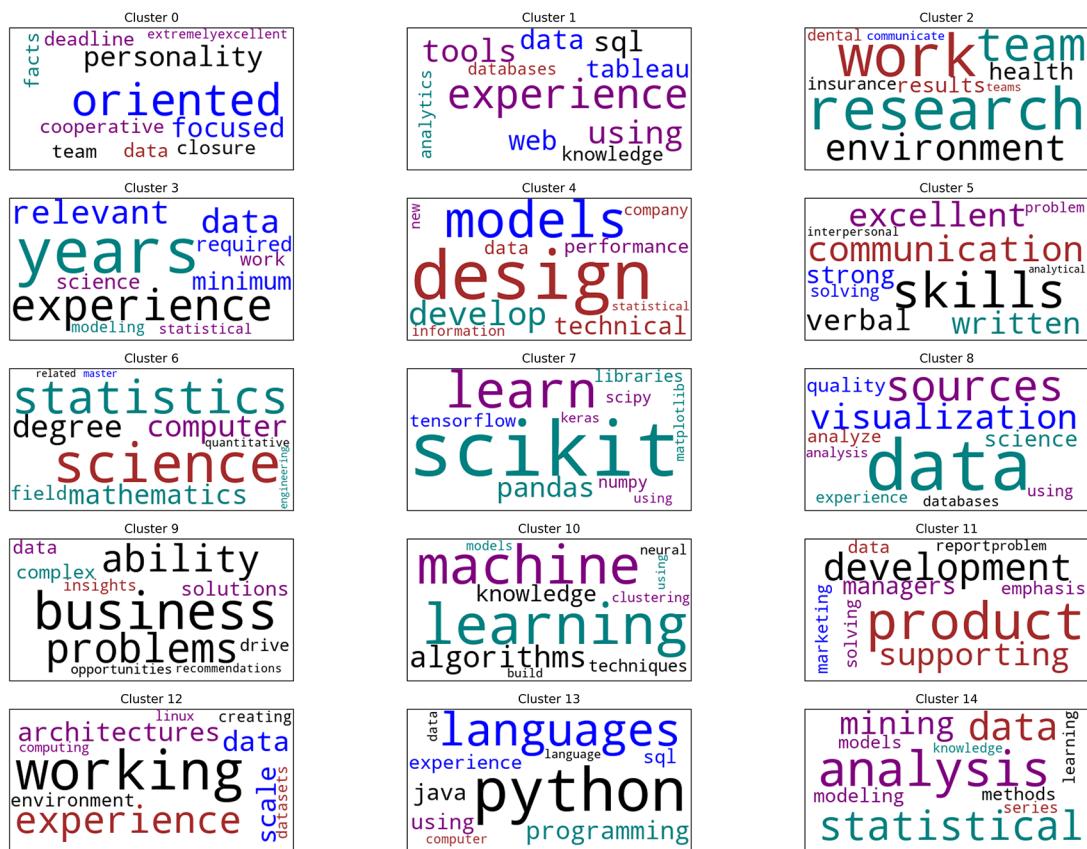


Figure 17.9 15 word clouds visualized across 15 subplots. Each word cloud corresponds to one of 15 clusters. The subplot titles correspond to cluster IDs. Some clusters, like cluster 7, describe technical skills; others, like cluster 0, are less technical.

dominates. Most of these libraries appear in our resume and have been discussed in this book. Let's print a sample of bullets from cluster 7 and confirm their focus on data science libraries.

Listing 17.36 Printing sample bullets from cluster 7

```
np.random.seed(1)
print_cluster_sample(7)
```

Experience using one or more of the following software packages:
scikit-learn, numpy, pandas, jupyter, matplotlib, scipy, nltk, spacy, keras, tensorflow

Using one or more of the following software packages: scikit-learn, numpy,

pandas, jupyter, matplotlib, scipy, nltk, spacy, keras, tensorflow

Experience with machine learning libraries and platforms, like Scikit-learn and Tensorflow

Proficiency in incorporating the use of external proprietary and open-source libraries such as, but not limited to, Pandas, Scikit-learn, Matplotlib, Seaborn, GDAL, GeoPandas, and ArcPy
 Experience using ML libraries, such as scikit-learn, caret, mlr, mllib

Meanwhile, other clusters, like cluster 0, focus on nontechnical skills. These soft skills, covering business acumen, focus, strategy, communication, and collaboration, are clearly missing from our resume. Thus, on average, the nontechnical clusters should have a lower resume similarity. This line of thought leads to an interesting possibility: perhaps we can separate the technical clusters and soft-skill clusters using text similarity. The separation would allow us to more systematically examine each skill type. Let's give this a shot! We'll start by computing the cosine similarity between each bullet in `total_bullets` and our resume.

NOTE Why utilize just the resume rather than the combined resume and table of contents sorted in the `existing_skills` variable? Well, our end goal is to determine which skill clusters are missing from the resume. The direct similarity between the resume and each cluster could be useful in that regard. If the similarity is low, then clustered skills are not appropriately represented in the resume's text.

Listing 17.37 Computing similarities between the bullets and our resume

```
def compute_bullet_similarity(bullet_texts):           ←—————  

    bullet_vectorizer = TfidfVectorizer(stop_words='english')  

    matrix = bullet_vectorizer.fit_transform(bullet_texts + [resume])  

    matrix = matrix.toarray()  

    return matrix[:-1] @ matrix[-1]           Computes the cosine similarities between the  

                                              inputted bullet_texts and the resume variable  

bullet_cosine_similarities = compute_bullet_similarity(total_bullets)
```

Our `bullet_cosine_similarities` array contains the text similarities across all clustered bullets. For any given cluster, we can combine these cosine similarities into a score by taking their mean. According to our hypothesis, a technical cluster should have a higher mean similarity than a soft-skill similarity cluster. Let's confirm if this is the case for the technical cluster 7 and the soft-skill cluster 0.

Listing 17.38 Comparing mean resume similarities

```
def compute_mean_similarity(df_cluster):  

    indices = df_cluster.Index.values  

    return bullet_cosine_similarities[indices].mean()  

tech_mean = compute_mean_similarity(cluster_groups[7])  

soft_mean = compute_mean_similarity(cluster_groups[0])  

print(f"Technical cluster 7 has a mean similarity of {tech_mean:.3f}")  

print(f"Soft-skill cluster 3 has a mean similarity of {soft_mean:.3f}")  

Technical cluster 7 has a mean similarity of 0.203  

Soft-skill cluster 3 has a mean similarity of 0.002
```

The technical cluster is 100 times more proximate to our resume than the soft-skill cluster. It appears that we're on the right track! Let's compute the average similarity for all 15 clusters. Then we'll sort the clusters by their similarity score, in descending order. If our hypothesis is correct, technical clusters will appear first in the sorted results. We'll be able to confirm by replotting the word cloud subplot grid. Listing 17.39 carries out the sorting and visualizes the sorted clusters (figure 17.10).

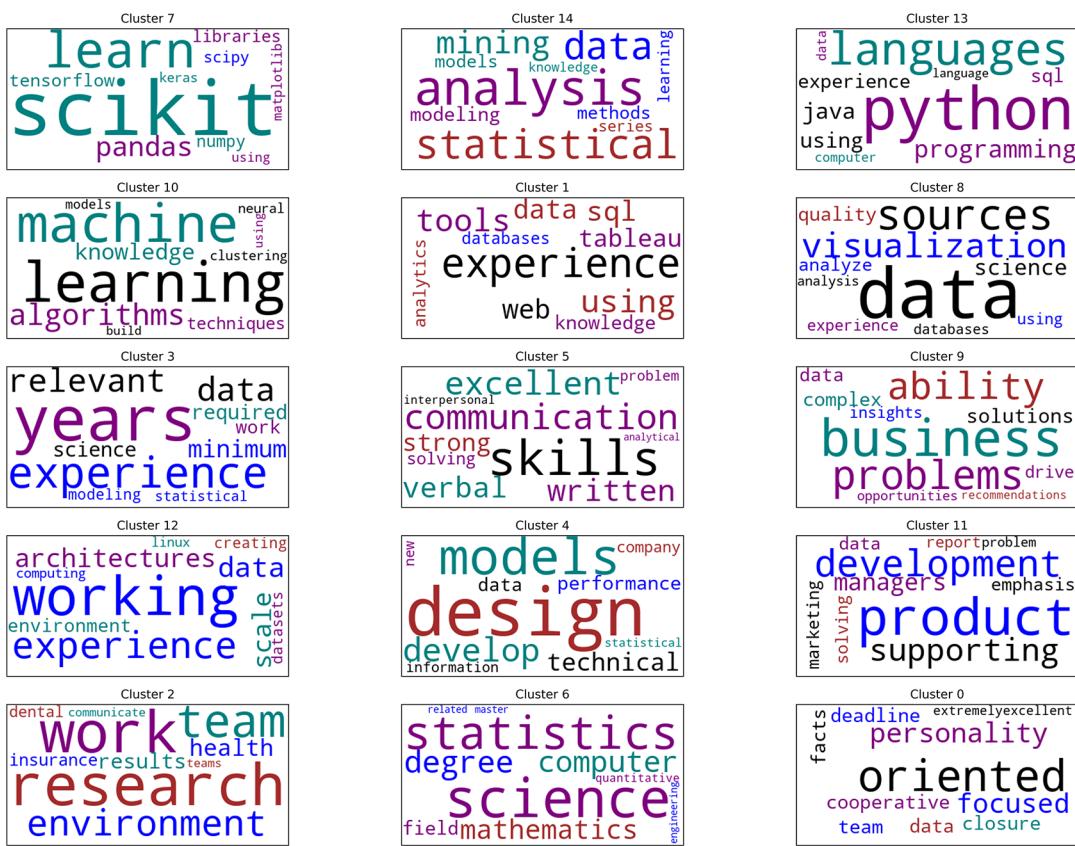


Figure 17.10 15 word clouds visualized across 15 subplots. Each word cloud corresponds to one of 15 clusters. The clusters are sorted by average resume similarity. The first two rows in the subplot grid correspond to more technical clusters.

NOTE We are about to sort the clusters by technical relevance. This isn't necessarily required to complete the case study—it's possible to examine each cluster individually, in unsorted order. However, by reordering the clusters, we can extract insights at a faster rate. Thus, sorting is a preferable way of simplifying our workflow.

Listing 17.39 Sorting subplots by resume similarity

```
def sort_cluster_groups(cluster_groups):
    mean_similarities = [compute_mean_similarity(df_cluster)
                          for df_cluster in cluster_groups]

    sorted_indices = sorted(range(len(cluster_groups)),
                           key=lambda i: mean_similarities[i],
                           reverse=True)
    return [cluster_groups[i] for i in sorted_indices]

sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups)
plt.show()
```

Sorts the inputted cluster_groups array by their mean cosine similarity to the resume

Our hypothesis was right! The first two rows in the updated subplot clearly correspond to technical skills. Furthermore, these technical skills are now conveniently sorted based on their similarity to our resume. This allows us to systematically rank the skills from most similar (and thus represented by our resume) to least similar (and thus likely to be missing from our resume).

17.3.2 Investigating the technical skill clusters

Let's turn our attention to the six technical-skill clusters in the first two rows of the subplot grid. Next, we replot their associated word clouds in a two-row-by-three-column grid (figure 17.11). This technically focused visualization will allow us to expand the size of the word cloud. Later, we return to the remaining soft-skill word clouds in figure 17.10.

Listing 17.40 Plotting just the first six technical clusters

```
plot_wordcloud_gri(sorted_cluster_groups[:6], num_rows=3, num_columns=2)
plt.show()
```

The first four technical-skill clusters in the grid plot are very informative. We'll now examine these clusters one by one, starting with the grid's upper-left quadrant. For brevity's sake, we rely solely on the word clouds; their contents should be sufficient to grasp the skills represented by each cluster. However, if you wish to dive deeper into any cluster, feel free to sample the cluster's bullet points.

The first four technical clusters can be described as follows:

- *Cluster 7 (row 0, column 0)*—This data science library cluster has already been discussed. Libraries such as scikit-learn, NumPy, SciPy, and Pandas have been covered in this book.

The following two libraries have not been covered: TensorFlow and Keras. These are deep learning libraries used by AI practitioners to train complex, predictive models on high-powered hardware. The boundary between data science positions and AI positions is not always clear. Although deep learning

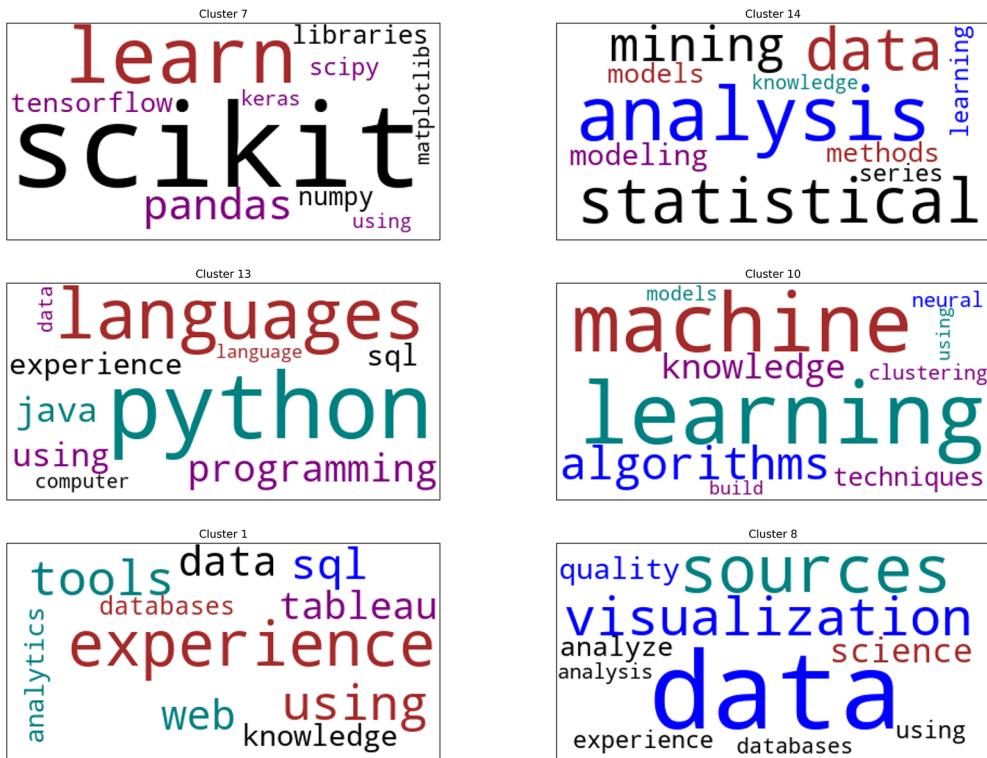


Figure 17.11 Six word clouds associated with six technical-skill clusters. They are sorted by average resume similarity. The first four word clouds are informative: they focus on data science libraries, statistical analysis, Python programming, and machine learning. The remaining two word clouds are vague and uninformative.

knowledge is not usually a prerequisite, sometimes it will help you get a job. With this in mind, if you wish to study these libraries in more detail, check out *Machine Learning with TensorFlow* by Nishant Shukla (Manning, 2018, www.manning.com/books/machine-learning-with-tensorflow) or *Deep Learning with Python, Second Edition*, by François Chollet (Manning, 2021, www.manning.com/books/deep-learning-with-python-second-edition).

- *Cluster 14 (row 0, column 1)*—This cluster discusses statistical analysis, which is represented in our resume. Statistical methods were covered in case study 2 of this book.
- *Cluster 13 (row 1, column 0)*—This cluster focuses on programming language proficiency. Among the languages, Python clearly dominates. Given our experience with Python, why doesn't this programming cluster rank higher? Well, it turns out Python is not mentioned anywhere in our resume! Yes, we refer to plenty of Python libraries, implying our familiarity with the language, but the Python skills

that we've honed over the course of this book are not explicitly referenced. Perhaps we should update our resume by mentioning our Python skills.

- *Cluster 10 (row 1, column 2)*—This cluster focuses on machine learning. The machine learning field encompasses a variety of data-driven prediction algorithms. Many of these algorithms are presented in the subsequent case study in this book; but until this case study has been completed, we cannot reference machine learning in our resume.

As a side note, we should mention that clustering techniques are sometimes referred to as *unsupervised* machine learning algorithms. Thus, a reference to unsupervised techniques is permissible. But any reference to more general machine learning will give a false impression of our skills.

The final two technical-skill clusters are vague and uninformative. They mention numerous unrelated tools and analysis techniques. Listing 17.41 samples bullets from these clusters (8 and 1) to confirm a lack of pattern.

NOTE Both clusters mention databases. Database usage is a useful skill to have but not a major topic in either cluster. Later in this section, we encounter a database cluster, which arises when we increase the value of K .

Listing 17.41 Printing sample bullets from clusters 8 and 1

```
np.random.seed(1)
for cluster_id in [8, 1]:
    print(f'\nCluster {cluster_id}:')
    print_cluster_sample(cluster_id)

Cluster 8:
Use data to inform and label customer outcomes and processes
Perform exploratory data analysis for quality control and improved
understanding
Champion a data-driven culture and help develop best-in-class data science
capabilities
Work with data engineers to plan, implement, and automate integration of
external data sources across a variety of architectures, including local
databases, web APIs, CRM systems, etc
Design, implement, and maintain a cutting-edge cloud-based
data-infrastructure for large data-sets

Cluster 1:
Have good knowledge on Project management tools JIRA, Redmine, and Bugzilla
Using common cloud computing platforms including AWS and GCP in addition to
their respective utilities for managing and manipulating large data sources,
model, development, and deployment
Experience in project deployment using Heroku/Jenkins and using web Services
like Amazon Web Services (AWS)
Expert level data analytics experience with T-SQL and Tableau
Experience reviewing and assessing military ground technologies
```

We've finished our analysis of the technical-skill clusters. Four of these clusters were relevant, and two were not. Now, let's turn our attention to the remaining soft-skill clusters. We want to see if any relevant soft-skill clusters are present in the data.

17.3.3 Investigating the soft-skill clusters

We start by visualizing the remaining nine soft-skill clusters in a three-row-by-three-column grid (figure 17.12).

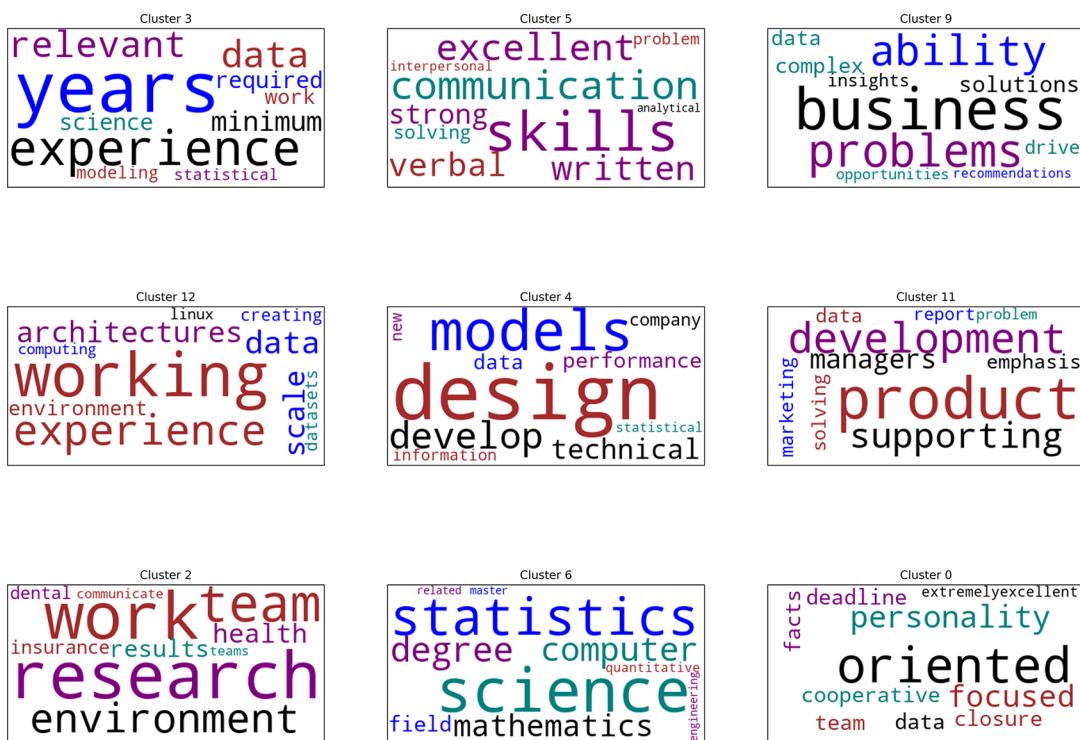


Figure 17.12 Nine word clouds associated with nine soft-skill clusters. They are sorted by average resume similarity. Most of the clusters are vague and uninformative, but the communication skills clusters in the first row are worth noting.

Listing 17.42 Plotting the remaining nine soft-skill clusters

```
plot_wordcloud_grid(sorted_cluster_groups[:6], num_rows=3, num_columns=3)
plt.show()
```

The remaining clusters appear much more ambiguous than the first four technical clusters. They are harder to interpret. For example, cluster 2 (row 2, column 0) uses vague terms such as *work*, *team*, *research*, and *environment*. Cluster 12 (row 1, column 0)

is equally enigmatic, composed of terms such as *environment*, *working*, and *experience*. Furthermore, the output is made more complicated by clusters that do not represent true skills! For instance, cluster 3 (row 0, column 0) is composed not of skills but of temporal experience: it consists of bullets requiring a minimum number of years working in industry. Similarly, cluster 6 (row 2, column 1) is not composed of skills; it represents educational constraints, requiring a quantitative degree to land an interview. We were slightly wrong in our assumptions—not all bullets represent true skills. We can confirm our error by sampling bullet points from clusters 6 and 3.

Listing 17.43 Printing sample bullets from clusters 6 and 3

```
np.random.seed(1)
for cluster_id in [6, 3]:
    print(f'\nCluster {cluster_id}:')
    print_cluster_sample(cluster_id)

Cluster 6:
MS in a quantitative research discipline (e.g., Artificial Intelligence, Computer Science, Machine Learning, Statistics, Applied Math, Operations Research)
Master's degree in data science, applied mathematics, or bioinformatics preferred.
PhD degree preferred
Ph.D. in a quantitative discipline (e.g., statistics, computer science, economics, mathematics, physics, electrical engineering, industrial engineering or other STEM fields)
7+ years of experience manipulating data sets and building statistical models, has advanced education in Statistics, Mathematics, Computer Science or another quantitative field, and is familiar with the following software/tools:

Cluster 3:
Minimum 6 years relevant work experience (if Bachelor's degree) or minimum 3 years relevant work experience (if Master's degree) with a proven track record in driving value in a commercial setting using data science skills.
Minimum five (5) years of experience manipulating data sets and building statistical models, and familiarity with:
5+ years of relevant work experience in data analysis or related field.
(e.g., as a statistician / data scientist / scientific researcher)
3+ years of statistical modeling experience
Data Science: 2 years (Required)
```

One of our soft-skill clusters is very easy to interpret: cluster 5 (row 0, column 1) focuses on interpersonal communication skills, both written and verbal. Good communication skills are crucial in a data science career. The insights we extract from complex data must be carefully communicated to all stakeholders. The stakeholders will then take consequential actions based on the persuasiveness of our argument. If we are unable to communicate our results, all our hard work will come to nothing.

Unfortunately, communication skills are not easy to learn. Simply reading a book is insufficient; practiced collaboration with other individuals is required. If you would

like to broaden your communication abilities, you should consider interacting with other budding data scientists, either locally or remotely. Choose a data-driven project, and complete that project as part of a team. Then be sure to emphasize your honed communication skills in your resume.

17.3.4 Exploring clusters at alternative values of K

K-means clustering gave us decent results when we set K to 15. However, that parameter input was partially arbitrary since we couldn't determine a perfectly optimal K . The arbitrary nature of our insights is a bit troubling: perhaps we just got lucky, and a different K would have yielded no insights at all. Or maybe we've missed critical clusters by choosing K incorrectly. The issue we need to probe is cluster consistency. How many of our insight-driving clusters will remain if we modify K ? To find out, we'll regenerate the clusters using alternative values of K . We begin by setting K to 25 and plotting the results in a five-row-by-five-column grid (figure 17.13). The subplots will be sorted based on cluster similarity to our resume.

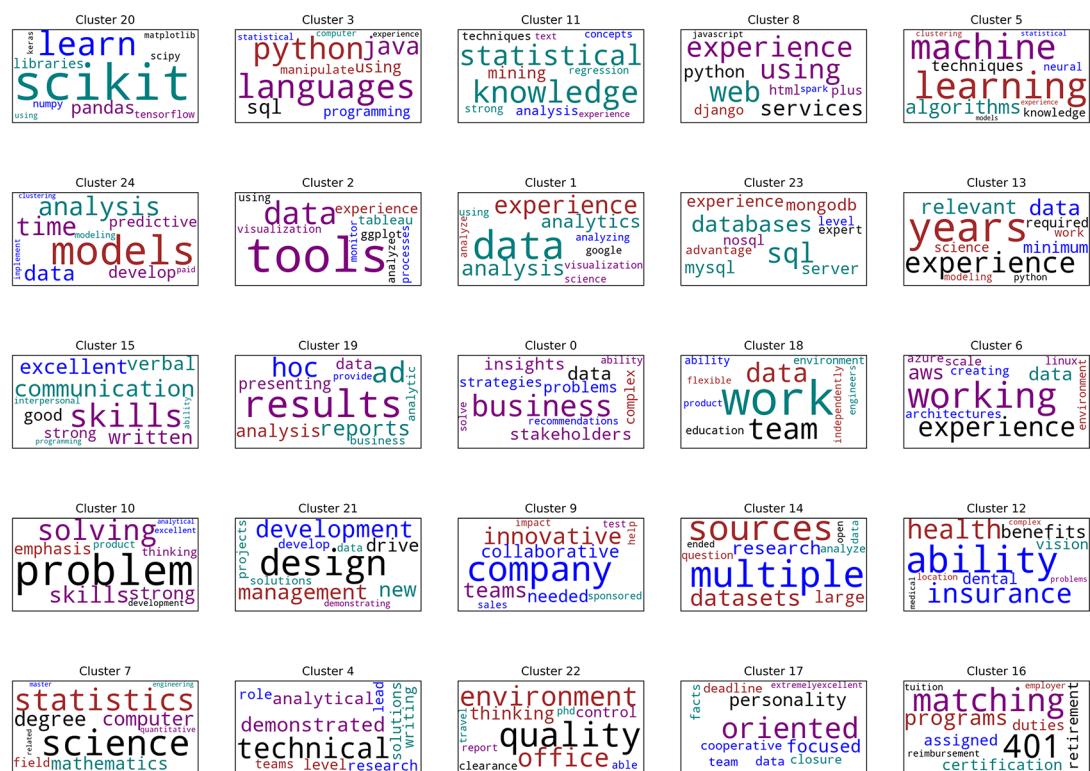


Figure 17.13 25 word clouds associated with 25 skill clusters. Our previously discussed skills remain present even after we've increased the value K . Additionally, we see new technical skills that are worth noting, including web service usage and familiarity with databases.

Listing 17.44 Visualizing 25 sorted clusters

```
np.random.seed(0)
cluster_groups = compute_cluster_groups(shrunk_norm_matrix, k=25)
sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups, num_rows=5, num_columns=5)
plt.show()
```

Most of the previously observed clusters remain in the updated output. These include data science library usage (row 0, column 0), statistical analysis (row 0, column 2), Python programming (row 0, column 1), machine learning (row 1, column 2), and communication skills (row 2, column 0). Additionally, we gain three insightful technical skill clusters, which appear among the first two rows of the grid:

- *Cluster 8 (row 0, column 4)*—This cluster focuses on web services. These are tools that propagate communication between a client and a remote server. In most industrial data science settings, data is stored remotely on a server and can be transferred using custom APIs. In Python, these API protocols are commonly coded using the Django framework. For budding data scientists, some familiarity with these tools is preferable but not necessarily required. To learn more about web services and API transfers, see *Amazon Web Services in Action* by Michael Wittig and Andreas Wittig (Manning, 2018, <https://www.manning.com/books/amazon-web-services-in-action-second-edition>) and *The Design of Web APIs* by Arnaud Lauret (Manning, 2019, <https://www.manning.com/books/the-design-of-web-apis>).
- *Cluster 23 (row 1, column 3)*—This cluster focuses on various types of databases. Large-scale structured data is commonly stored in relational databases and can be queried using Structured Query Language (SQL). However, not all databases are relational. Sometimes data is stored in alternative, unstructured databases, such as MongoDB. Data in an unstructured database can be queried using a NoSQL query language. Knowledge of the various database types can be quite useful in a data science career. If you would like to learn more about the subject, check out *Understanding Databases* (Manning, 2019, www.manning.com/books/understanding-databases); and to learn more about MongoDB, see *MongoDB in Action, Second Edition* by Kyle Banker et al. (Manning, 2016, www.manning.com/books/mongodb-in-action-second-edition).
- *Cluster 2 (row 1, column 1)*—This cluster focuses on non-Python visualization tools, such as Tableau and ggplot. Tableau is paid software provided by Salesforce and is commonly used by businesses that can afford the Salesforce contract; you can read more about it in *Practical Tableau* by Ryan Sleeper ((O'Reilly, 2018, <http://mng.bz/Xrdv>)). ggplot is a data-visualization package for the statistical programming language R. Generally, Python data scientists are not expected to know R; but if you would like to familiarize yourself with the subject, see *Practical Data Science with R, Second Edition* by Nina Zumel and

John Mount (Manning, 2019, www.manning.com/books/practical-data-science-with-r-second-edition).

Our plot also contains seven newly added clusters. These clusters contain mostly generic skills like problem solving (row 3, column 0) and teamwork (row 2, column 3). Also, at least one of the new skill clusters doesn't correspond to an actual skill (like the health insurance benefits cluster in row 3, column 4).

Increasing K from 15 to 25 retained all the previously observed insightful clusters and introduced several interesting new clusters. Will the stability of these clusters persist if we shift K to an intermediate value of 20? We find out next by plotting 20 sorted clusters in a four-row-by-five-column grid (figure 17.14).

Listing 17.45 Visualizing 20 sorted clusters

```
np.random.seed(0)
cluster_groups = compute_cluster_groups(shrunk_norm_matrix, k=20)
sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups, num_rows=4, num_columns=5)
plt.show()
```



Figure 17.14 20 word clouds associated with 20 skill clusters. Most of our previously discussed skills remain, but the statistical analysis cluster is now missing from the output.

Most of our observed insightful clusters remain at $k=20$, including data science library usage (row 0, column 0), Python programming (row 0, column 3), machine learning (row 1, column 0), communication skills (row 1, column 4), web services (row 0, column 1), and database usage (row 0, column 4). However, the non-Python visualization cluster is gone. More troublingly, the statistical analysis cluster observed at K values of 15 and 25 is missing.

NOTE This statistical analysis cluster appears to have been replaced by a statistical algorithm cluster, which is positioned at row 0, column 2 of the grid. It is dominated by three terms: *algorithms*, *clustering*, and *regression*. Of course, by now, we're intimately familiar with clustering. However, regression techniques are missing from our resume because we haven't learned them yet. We will learn these techniques in case study 5 and can then add them to our resume.

A seemingly stable cluster has been eliminated. Unfortunately, such fluctuations are quite common. Text clustering is sensitive to parameter changes due to the complex nature of human language. Language topics can be interpreted in a multitude of ways, making it hard to find consistently perfect parameters. Clusters that appear under one set of parameters may disappear if these parameters are tweaked. If we cluster over just a single value of K , we risk missing out on useful insights. Thus, it's preferable to visualize results over a range of K values during text analysis. With this in mind, let's see what happens when we reduce K to 10 (figure 17.15).

Listing 17.46 Visualizing 10 sorted clusters

```
np.random.seed(0)
cluster_groups = compute_cluster_groups(shrunk_norm_matrix, k=10)
sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups, num_rows=5, num_columns=2)
plt.show()
```

The 10 visualized clusters are quite limited. Nonetheless, 4 of the 10 clusters contain critical skills we've observed previously: Python programming (row 0, column 0), machine learning (row 0, column 1), and communication skills (row 2, column 1). The statistical analysis cluster has also reappeared (row 1, column 0). Surprisingly, some of our skill clusters are versatile and appear even when the K value is radically adjusted. Despite some stochasticity in our clustering, a level of consistency remains. Thus, the insights we observe aren't just random outputs—they are tangible patterns that we've captured across complex, messy, real-world texts.

So far, our observations have been limited to the 60 most relevant job postings. However, as we've seen, there is some noise in that data subset. What will happen if we extend our analysis to the top 700 postings? Will our observations change or stay the same? Let's find out.



Figure 17.15 10 word clouds associated with 10 skill clusters. Four of our previously discussed skills remain, despite the low value of K.

17.3.5 Analyzing the 700 most relevant postings

We start by preparing `sorted_df_jobs[:700].Bullets` for clustering by doing the following:

- 1 Extract all the bullets while removing duplicates.
- 2 Vectorize the bullet texts.
- 3 Dimensionally reduce the vectorized texts, and normalize the resulting matrix.

Listing 17.47 Preparing `sorted_df_jobs[:700]` for clustering analysis

```
np.random.seed(0)
total_bullets_700 = set()
for bullets in sorted_df_jobs[:700].Bullets:
```

```

total_bullets_700.update([bullet.strip()
                         for bullet in bullets])

total_bullets_700 = sorted(total_bullets_700)
vectorizer_700 = TfidfVectorizer(stop_words='english')
tfidf_matrix_700 = vectorizer_700.fit_transform(total_bullets_700)
shrunk_norm_matrix_700 = shrink_matrix(tfidf_matrix_700)
print(f"We've vectorized {shrunk_norm_matrix_700.shape[0]} bullets")

We've vectorized 10194 bullets

```

We've vectorized 10,194 bullet points. Now, we generate an elbow plot across the vectorized results. Based on previous observations, we don't expect the elbow plot to be particularly informative, but we create the plot to stay consistent with our previous analyses (figure 17.16).

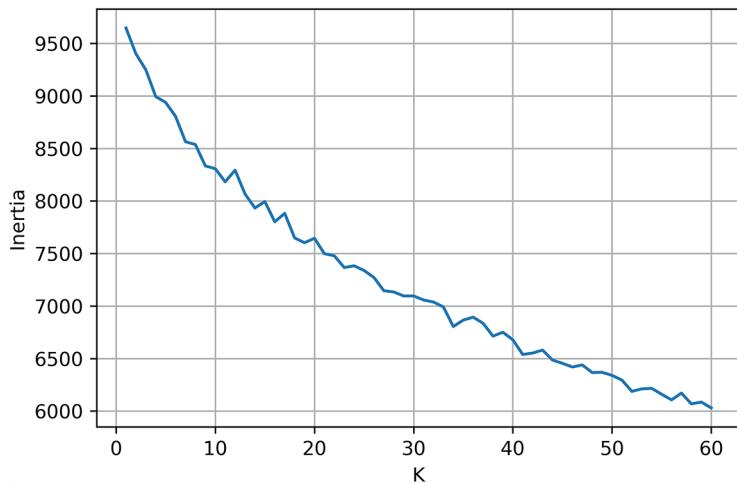


Figure 17.16 An elbow plot generated using bullets from the top 700 most relevant postings. The precise location of the elbow is difficult to determine.

Listing 17.48 Plotting an elbow curve for 10,194 bullets

```

np.random.seed(0)
generate_elbow_plot(shrunk_norm_matrix_700)
plt.show()

```

As expected, the precise location of the elbow is not clear in the plot. The elbow is spread out between a K of 10 and 25. We'll deal with ambiguity by arbitrarily setting K to 20. Let's generate and visualize 20 clusters; if necessary, we'll adjust K for comparative clustering (figure 17.17).

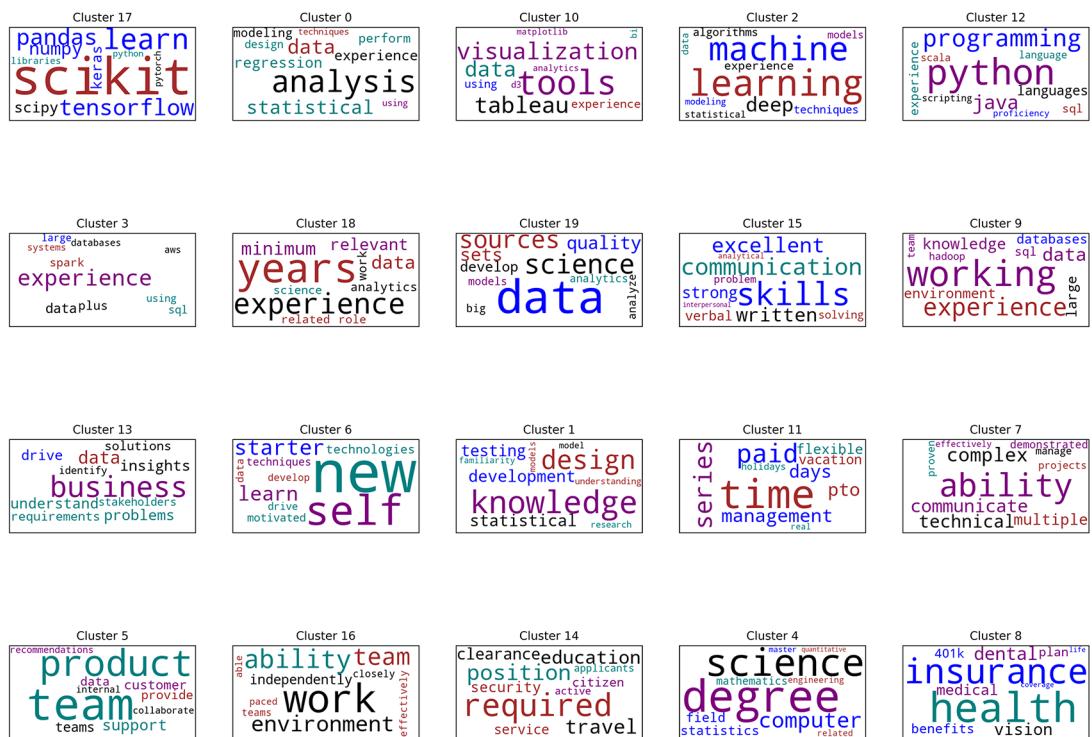


Figure 17.17 20 word clouds generated by clustering over 10,000 bullets. Despite the 10-fold increase in bullets, the observed skill clusters mostly remain the same.

WARNING As we discussed in section 15, the K-means output can vary across computers for large matrices containing 10,000-by-100 elements. Your local clustering results may differ from the output shown here, but you should be able to draw conclusions similar to those presented in this book.

Listing 17.49 Visualizing 20 sorted clusters for 10,194 bullets

```

np.random.seed(0)
cluster_groups_700 = compute_cluster_groups(shrunk_norm_matrix_700, k=20,
                                             bullets=total_bullets_700)
bullet_cosine_similarities = compute_bullet_similarity(total_bullets_700)
sorted_cluster_groups_700 = sort_cluster_groups(cluster_groups_700)
plot_wordcloud_grid(sorted_cluster_groups_700, num_rows=4, num_columns=5,
                     vectorizer=vectorizer_700, ←
                     tfidf_matrix=tfidf_matrix_700)
    
```

Recomputes
bullet_cosine_similarities
for sorting purposes

We need to pass the updated
TFIDF matrix and vectorizer
into our plotting function.

Our clustering output looks very similar to what we've seen before. The key insightful clusters we observed at 60 postings remain, including data science library usage (row 0,

column 0), statistical analysis (row 0, column 1), Python programming (row 0, column 4), machine learning (row 0, column 3), and communication skills (row 1, column 3). Some subtle changes are present, but for the most part, the output is the same.

NOTE One interesting change is the appearance of a generalized visualization cluster (row 0, column 2). This cluster encompasses a variety of visualization tools, including Matplotlib. Additionally, the freely available JavaScript library D3.js is mentioned in the cluster's word cloud. The D3.js library is used by some data scientists to make interactive web visualizations. To learn more about the library, see *D3.js in Action, Second Edition* by Elijah Meeks (Manning, 2017, www.manning.com/books/d3js-in-action-second-edition).

Certain skills consistently appear in the job postings. These skills are not very sensitive to our selected relevance threshold, so we can elucidate them even if our threshold remains uncertain.

17.4 Conclusion

We're ready to update the draft of our resume. First and foremost, we should emphasize our Python skills. A single line saying that we're *proficient in Python* should be sufficient. Additionally, we want to assert our communication skills. How do we show that we're good communicators? It's tricky; simply stating that we can *clearly communicate complex results to different audiences* is not enough. Instead, we should describe a personal project in which we did the following:

- Collaborated with teammates on a difficult data problem
- Conveyed complex results, in oral or written form, to a nontechnical audience

NOTE If you've experienced working on this type of project, you should definitely add it to your resume. Otherwise, you're encouraged to pursue this type of project voluntarily. The skills you'll gain will prove invaluable while also bettering your employment prospects.

Furthermore, before we complete our resume, we need to address our remaining skill deficiencies. Machine learning experience is crucial to a successful data science career. We haven't yet studied machine learning, but in the subsequent case study, we expand our machine learning skills. Then we'll be able to proudly describe our machine learning abilities in our resume.

Finally, it's worthwhile to demonstrate some experience with tools to fetch and store remote data. These tools include databases and hosted web services. Their use is beyond the scope of this book, but they can be learned through independent study. Database and web services experience isn't always required to get the job; nevertheless, some limited experience is always welcomed by potential employers.

Summary

- Text data should not be analyzed blindly. We should always sample and read some of the text before running any algorithms. This is especially true of HTML files, where tags can demarcate unique signals in the text. By rendering sampled job postings, we discovered that unique job skills are marked by bullet points in each HTML file. If we had blindly clustered the body of each file, our final results wouldn't have been as informative.
- Text clustering is hard. An ideal cluster count rarely exists because language is fluid, and so are boundaries between topics. But despite the uncertainty, certain topics consistently appear across multiple cluster counts. So, even if our elbow plot does not reveal the exact number of clusters, the situation is salvageable: sampling over multiple clustering parameters can reveal stable topics in the text.
- Choosing parameter values is not always easy. This issue extends well beyond mere clustering. When selecting our relevance cutoff, we were torn between two values: 60 and 700. Neither value seemed much superior to the other, so we tried both! In data science, some problems don't have an ideal threshold or parameter. However, we shouldn't give up and ignore such problems. On the contrary, we should experiment. Scientists learn by exploring outputs across a range of parameter inputs. As data scientists, we can gain invaluable insights by tweaking and adjusting our parameters.

Case study 5

Predicting future friendships from social network data

Problem statement

Welcome to FriendHook, Silicon Valley's hottest new startup. FriendHook is a social networking app for college undergrads. To join, an undergrad must scan their college ID to prove their affiliation. After approval, undergrads can create a FriendHook profile, which lists their dorm name and scholastic interests. Once a profile is created, an undergrad can send *friend requests* to other students at their college. A student who receives a friend request can either approve or reject it. When a friend request is approved, the pair of students are officially *FriendHook friends*. Using their new digital connection, FriendHook friends can share photographs, collaborate on coursework, and keep each other up to date on the latest campus gossip.

The FriendHook app is a hit. It's utilized on hundreds of college campuses worldwide. The user base is growing, and so is the company. You are FriendHook's first data science hire! Your first challenging task will be to work on FriendHook's friend recommendation algorithm.

Introducing the friend-of-a-friend recommendation algorithm

Sometimes FriendHook users have trouble finding their real-life friends on the digital app. To facilitate more connections, the engineering team has implemented a simple friend-recommendation engine. Once a week, all users receive an email recommending a new friend who is not yet in their network. The users can ignore the email, or they can send a friend request. That request is then either accepted or rejected/ignored.

Currently, the recommendation engine follows a simple algorithm called the *friend-of-a-friend recommendation algorithm*. The algorithm works like this. Suppose we want to recommend a new friend for student A. We pick a random student B who is already friends with student A. We then pick a random student C who is friends with student B but not student A. Student C is then selected as the recommended friend for student A, as shown in figure CS5.1.

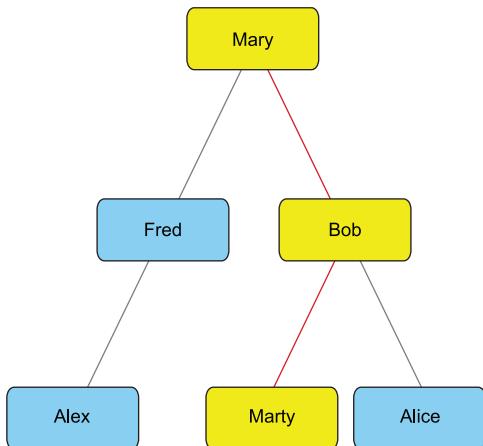


Figure CS5.1 The friend-of-a-friend recommendation algorithm in action. Mary has two friends: Fred and Bob. One of these friends (Bob) is randomly selected. Bob has two additional friends: Marty and Alice. Neither Alice nor Marty are friends with Mary. A friend of a friend (Marty) is randomly selected. Mary receives an email suggesting that she should send a friend request to Marty.

Essentially, the algorithm assumes that a friend of your friend is also likely to be your friend. This assumption is reasonable but also a bit simplistic. How well does this assumption hold? Nobody knows! However, as the company's first data scientist, it's your job to find out. You have been tasked with building a model that predicts student behavior in response to the recommendation algorithm.

Predicting user behavior

The friend-of-a-friend recommendation engine can elicit three types of behaviors:

- A user reads the emailed friend recommendation and either rejects or ignores that recommendation.
- A user sends a friend request based on the recommendation. That request is rejected or ignored.
- A user sends a friend request based on the recommendation. The friend request is accepted, and a new FriendHook connection is established.

Is it possible to predict these three behaviors? The FriendHook CTO would like you to find out. The CTO has provided you with FriendHook data from a randomly selected university. That data covers all FriendHook users at the university, including observed behaviors for all users in response to the weekly friend recommendations. The data also includes profile information for each user covering things like student major and residential dormitory name. This personal profile information has been encrypted to protect each user's privacy (more on that later). Finally, the data includes a network of existing FriendHook connections at the university, compiled right before friendship recommendations were emailed.

Your task is to build a model that predicts user behavior based on user profiles and social network data. The model must generalize to other colleges and universities. This generalizability is very important—a model that cannot be utilized at other colleges is worthless to the product team. Consider, for example, a model that accurately predicts behavior in one or two of the dorms at the sampled university. In other words, it requires specific dorm names to make accurate predictions. Such a model is not useful because other universities will have different dormitory names. Ideally, the model should generalize to all dormitories across all universities worldwide.

Once you've built the generalized model, you should explore its inner workings. Your goal is to gain insights into how university life facilitates new FriendHook connections.

The project goals are ambitious but also very doable. You can complete them by carrying out the following tasks:

- 1 Load the three datasets pertaining to user behavior, user profiles, and the user friendship network. Explore each dataset, and clean it as required.
- 2 Build and evaluate a model that predicts user behavior based on user profiles and established friendship connections. You can optionally split this task into two subtasks: build a model using just the friendship network, and then add the profile information and test whether this improves the model's performance.
- 3 Determine whether the model generalizes well to other universities.
- 4 Explore the inner workings of the model to gain better insights into student behavior.

Dataset description

Our data contains three files stored in a friendhook directory. These files are CSV tables and are named Profiles.csv, Observations.csv, and Friendships.csv. Let's discuss each table individually.

The Profiles table

Profiles.csv contains profile information for all the students at the chosen university. This information is distributed across six columns: Profile_ID, Sex, Relationship_Status, Major, Dorm, and Year. Maintaining student privacy is very important to the FriendHook team, so all the profile information has been carefully encrypted.

FriendHook's encryption algorithm takes in descriptive text and returns a unique, scrambled 12-character code known as a *hash code*. Suppose, for example, that a student lists their major as physics. The word *physics* is then scrambled and replaced with a hash code such as *b90a1221d2bc*. If another student lists their major as art history, a different hash code is returned (for example, *983a9b1dc2ef*). In this manner, we can check whether two students share the same major without necessarily knowing the identity of that major. All six profile columns have been encrypted as a precautionary measure. Let's discuss the separate columns in detail:

- **Profile_ID**—A unique identifier used to track each student. The identifier can be linked to the user behaviors in the **Observations** table. It can also be linked to FriendHook connections in the **Friendships** table.
- **Sex**—This optional field describes the sex of a student as **Male** or **Female**. Students who don't wish to specify a gender can leave the **Sex** field blank. Blank inputs are stored as empty values in the table.
- **Relationship_Status**—This optional field specifies the relationship status of the student. Each student has three relationship categories to choose from: **Single**, **In a Relationship**, or **It's Complicated**. All students have a fourth option of leaving this field blank. Blank inputs are stored as empty values in the table.
- **Major**—The chosen area of study for the student, such as physics, history, economics, etc. This field is required to activate a FriendHook account. Students who have not yet picked their major can select **Undecided** from among the options.
- **Dorm**—The name of the dormitory where the student resides. This field is required to activate a FriendHook account. Students who reside in off-campus housing can select **Off-Campus Housing** from among the options.
- **Year**—The undergraduate student's year. This field must be set to one of four options: **Freshman**, **Sophomore**, **Junior**, or **Senior**.

The **Observations** table

Observations.csv contains the observed user behavior in response to the emailed friend recommendation. It includes the following five fields:

- **Profile_ID**—The ID of the user who received a friend recommendation. The ID corresponds to the profile ID in the **Profiles** table.
- **Selected_Friend**—An existing friend of the user in the **Profile_ID** column.
- **Selected_Friend_of_Friend**—A randomly chosen friend of **Selected_Friend** who is not yet a friend of **Profile_ID**. This random friend of a friend is emailed as a friend recommendation for the user.
- **Friend_Request_Sent**—A Boolean column that is **True** if a user sends a friend request to the suggested friend of a friend or **False** otherwise.
- **Friend_Request_Accepted**—A Boolean column that is **True** only if a user sends a friend request and that request is accepted.

This table stores all the observed user behaviors in response to the weekly recommendation email. Our goal is to predict the Boolean outputs of the final two table columns based on the profile and social networking data.

The Friendships table

`Friendships.csv` contains the FriendHook friendship network corresponding to the selected university. This network was used as input into the friend-of-a-friend recommendation algorithm. The `Friendships` table has just two columns: `Friend A` and `Friend B`. These columns contain profile IDs that map to the `Profile_ID` columns of the `Profiles` and `Observations` tables. Each row corresponds to a pair of FriendHook friends. For instance, the first row contains IDs `b8bc075e54b9` and `49194b3720b6`. From these IDs, we can infer that the associated students have an established FriendHook connection. Using the IDs, we can look up the profile of each student. The profiles then allow us to explore whether the friends share the same major or reside together in the same dorm.

Overview

To address the problem at hand, we need to know how to do the following:

- Analyze network data using Python
- Discover friendship clusters in social networks
- Train and evaluate supervised machine learning models
- Probe the inner workings of trained models to draw insights from our data

An introduction to graph theory and network analysis

This section covers

- Representing diverse datasets as networks
- Network analysis with the NetworkX library
- Optimizing travel paths in a network

The study of connections can potentially yield billions of dollars. In the 1990s, two graduate students analyzed the properties of interconnected web pages. Their insights led them to found Google. In the early 2000s, an undergraduate began to digitally track connections between people. He went on to launch Facebook. Connection analysis can lead to untold riches, but it can also save countless lives. Tracking the connections between proteins in cancer cells can generate drug targets that will wipe out that cancer. Analyzing connections between suspected terrorists can uncover and prevent devious plots. These seemingly disparate scenarios have one thing in common: they can be studied using a branch of mathematics called *network theory* by some and *graph theory* by others.

Network theory is the study of connections between objects. These objects can be anything: people connected by relationships, web pages connected by web links, or cities connected by roads. A collection of objects and their dispersed connections is called either a *network* or a *graph*, depending on whom you ask. Engineers prefer to

use the term *network*, while mathematicians prefer *graph*. For our intents and purposes, we'll use the two terms interchangeably. Graphs are simple abstractions that capture the complexity of our entangled, interconnected world. Properties of graphs remain surprisingly consistent across systems in society and nature. Graph theory is a framework for mathematically tracking these consistencies. It combines ideas from diverse branches of mathematics, including probability theory and matrix analysis. These ideas can be used to gain useful real-world insights ranging from search engine page rankings to social circle clustering, so some knowledge of graph theory is indispensable to doing good data science.

In the next two sections, we learn the fundamentals of graph theory by building on previously studied data science concepts and libraries. We start slowly by addressing basic problems while exploring graphs of web page links and roads. Later, in section 19, we utilize more advanced techniques to detect clusters of friends in social graphs. However, we begin with a much simpler data science task of ranking websites by popularity.

18.1 Using basic graph theory to rank websites by popularity

There are many data science websites on the internet. Some sites are more popular than others. Suppose you wish to estimate the most popular data science website using data that is publicly available. This precludes privately tracked traffic data. What should you do? Network theory offers us a simple way of ranking websites based on their public links. To see how, let's build a simple network composed of two data science websites: a NumPy tutorial and a SciPy tutorial. In graph theory, these websites are referred to as the *nodes* in the graph. Nodes are network points that can form connections with each other; these connections are called *edges*. Our two website nodes will form an edge if one site links to the other or vice versa.

We begin by storing our two nodes in a two-element list. These elements equal 'NumPy' and 'SciPy', respectively.

Listing 18.1 Defining a node list

```
nodes = ['NumPy', 'SciPy']
```

Suppose the SciPy website is discussing NumPy dependencies. This discussion includes a web link to the NumPy page. Clicking that link will take the reader from the website represented by `nodes[1]` to the website represented by `nodes[0]`. We treat this connection as an edge that goes from index 1 to index 0, as shown in figure 18.1. The edge can be expressed as the tuple `(1, 0)`. Here, we form an edge by storing `(1, 0)` in an `edges` list.

Listing 18.2 Defining an edge list

```
edges = [(1, 0)]
```

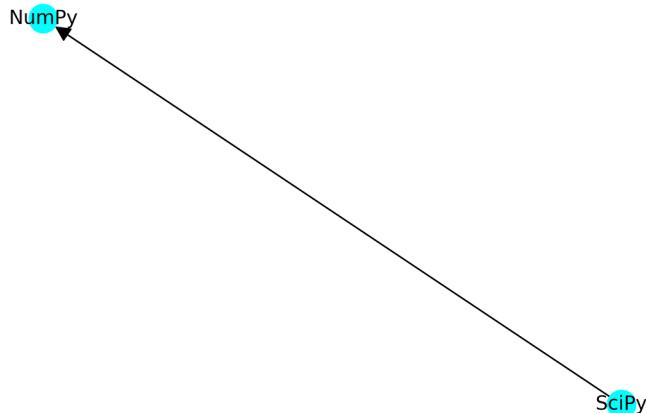


Figure 18.1 Two websites, NumPy and SciPy, are represented as circular nodes. A directed edge points from SciPy to NumPy, indicating a directed link between the sites. If NumPy and SciPy are stored as node indices 0 and 1, we can represent the edge as tuple $(1, 0)$. Later in this section, we learn how to generate the network diagram in this figure.

Our single edge $(1, 0)$ represents a link that directs a user from nodes [1] to nodes [0]. This edge has a specific direction and is referred to as a *directed edge*. Graphs containing directed edges are referred to as *directed graphs*. In a directed graph, edge (i, j) is treated differently from edge (j, i) . The presence of (i, j) in an edges list does not imply the presence of (j, i) . For instance, in our network, the NumPy page does not yet link to the SciPy page, so edge tuple $(0, 1)$ is not present in the edges list.

Given our directed edges list, we can easily check if a web page at index i links to a web page at index j . That connection exists if (i, j) in edges equals True. Thus, we can define a one-line `edge_exists` function, which checks for edges between indices i and j .

Listing 18.3 Checking for the existence of an edge

```
def edge_exists(i, j): return (i, j) in edges

assert edge_exists(1, 0)
assert not edge_exists(0, 1)
```

Our `edge_exists` function works, but it's not efficient. The function must traverse a list to check the presence of an edge. This traversal is not an issue for an edge list of size 1. However, if we were to increase our network size to 1,000 web pages, then our edge list size might increase to as many as 1 million edges. Traversing a million-edge list is not computationally justifiable. We need an alternative solution.

One alternative approach is to store the presence or absence of each edge (i, j) in the i th row and j th column of a table. Essentially, we can construct a table t in which $t[i][j] = \text{edge_exists}(i, j)$, so edge lookup will become instantaneous. Furthermore, we can represent this table as a 2D binary array if we store not $\text{edge_exists}(i, j)$ as 0 and $\text{edge_exists}(i, j)$ as 1, so we can represent our graph as a binary matrix M , where $M[i][j] = 1$ if an edge exists between node i and node j . This matrix representation of a network is known as an *adjacency matrix*. We now compute and print an adjacency matrix for our two-node single-edge directed graph. Initially, that matrix contains just 0s. Then we iterate each edge (i, j) in edges and set $\text{adjacency_matrix}[i][j]$ to 1.

Listing 18.4 Tracking nodes and edges using a matrix

```
import numpy as np
adjacency_matrix = np.zeros((len(nodes), len(nodes)))
for i, j in edges:
    adjacency_matrix[i][j] = 1

assert adjacency_matrix[1][0]
assert not adjacency_matrix[0][1]

print(adjacency_matrix)

[[0. 0.]
 [1. 0.]]
```

Our matrix printout permits us to view the edges present in the network. Additionally, we can observe potential edges that are missing from the network. For instance, we can clearly see an edge from Node 1 to Node 0. Meanwhile, possible edges $(0, 0)$, $(0, 1)$, and $(1, 1)$ are not present in the graph. Neither is there a link going from Node 0 to Node 0. The NumPy page does not link to itself, although theoretically, it could! We can imagine a poorly designed web page in which a hyperlink points to itself—the link would be useless since clicking it would take you right back where you started, but this type of self-linkage is possible. In graph theory, such self-referential edges are called *self-loops* or *buckles*. In the next section, we encounter an algorithm that is improved if we incorporate self-loops. However, for the time being, we limit our analysis to edges between different pairs of nodes.

Let's add the missing edge from Node 0 to Node 1. This will imply that the NumPy page now links to the SciPy page.

Listing 18.5 Adding an edge to the adjacency matrix

```
adjacency_matrix[0][1] = 1
print(adjacency_matrix)

[[0. 1.]
 [1. 0.]]
```

Suppose we wish to expand our website network by adding two more data science sites that discuss Pandas and Matplotlib. Adding them will increase our node count from two to four, so we need to expand the adjacency matrix dimensions from two-by-two to four-by-four. During that expansion, we'll also maintain all existing relationships between Node 0 and Node 1. Unfortunately, in NumPy, it's hard to resize a matrix while maintaining all existing matrix values—NumPy is not designed to easily handle growing arrays whose shape is constantly expanding. This conflicts with the expanding nature of the internet, where new websites are constantly being added. Therefore, NumPy is not the best tool for analyzing expanding networks. What should we do?

NOTE NumPy is inconvenient for tracking newly added nodes and edges. However, as we previously discussed, it is indispensable for efficiently executing matrix multiplication. In the next section, we multiply the adjacency matrix to analyze social graphs. Thus, our use of NumPy will prove essential for advanced network analysis. But for the time being, we rely on an alternative Python library to more easily construct our networks.

We need to switch to a different Python library. NetworkX is an external library that allows for easy network modification. It also provides additional useful features, including network visualization. Let's proceed with our website analysis using NetworkX.

18.1.1 Analyzing web networks using NetworkX

We begin by installing NetworkX. Then we import networkx as nx, per the common NetworkX usage convention.

NOTE Call pip install networkx from the command line terminal to install the NetworkX library.

Listing 18.6 Importing the NetworkX library

```
import networkx as nx
```

Now we will utilize nx to generate a directed graph. In NetworkX, directed graphs are tracked using the nx.DiGraph class. Calling nx.DiGraph() initializes a new directed graph object containing zero nodes and zero edges. The following code initializes that directed graph; per NetworkX convention, we refer to the initialized graph as G.

Listing 18.7 Initializing a directed graph object

```
G = nx.DiGraph()
```

Let's slowly expand the directed graph. To start, we add a single node. Nodes can be added to a NetworkX graph object using the add_node method. Calling G.add_node(0) creates a single node whose adjacency matrix index is 0. We can view this adjacency matrix by running nx.to_numpy_array(G).

WARNING The `add_node` method always expands the graph's adjacency matrix by a single node. This expansion happens regardless of the method input. Hence, `G.add_node(1000)` also creates a node whose adjacency matrix index is 0. However, that node will also be tracked using a secondary index of 1000, which of course can lead to confusion. It's a good practice to ensure that numeric inputs into `add_node` correspond to added adjacency matrix indices.

Listing 18.8 Adding a single node to a graph object

```
G.add_node(0)
print(nx.to_numpy_array(G))

[[0.]]
```

Our single node is associated with a NumPy web page. We can explicitly record this association by executing `G.nodes[0]['webpage'] = 'NumPy'`. The `G.nodes` datatype is a special class intended to track all the nodes in `G`. It is structured like a list. Running `G[i]` returns a dictionary of attributes associated with the node at `i`. These attributes are intended to help us track the identity of the node. In our case, we wish to assign a web page to the node, so we map a value to `G.nodes[i]['webpage']`.

The following code iterates across `G.nodes` and prints the attribute dictionary at `G.nodes[i]`. Our initial output represents a single node whose attribute dictionary is empty: we assign a web page to that node and print its dictionary again.

Listing 18.9 Adding an attribute to an existing node

```
def print_node_attributes():
    for i in G.nodes:
        print(f"The attribute dictionary at node {i} is {G.nodes[i]}")

print_node_attributes()
G.nodes[0]['webpage'] = 'NumPy'
print("\nWe've added a webpage to node 0")
print_node_attributes()

The attribute dictionary at node 0 is {}

We've added a webpage to node 0
The attribute dictionary at node 0 is {'webpage': 'NumPy'}
```

We can assign attributes directly while inserting a node into the graph. We just need to pass `attribute=some_value` into the `G.add_node` method. For instance, we are about to insert a node with an index of 1, which is associated with a SciPy web page. Executing `G.add_node(1, webpage='SciPy')` adds the node and its attribute.

Listing 18.10 Adding a node with an attribute

```
G.add_node(1, webpage='SciPy')
print_node_attributes()
```

```
The attribute dictionary at node 0 is {'webpage': 'NumPy'}
The attribute dictionary at node 1 is {'webpage': 'SciPy'}
```

Note that we can output all the nodes together with their attributes simply by running `G.nodes(data=True)`.

Listing 18.11 Outputting nodes together with their attributes

```
print(G.nodes(data=True))

[(0, {'webpage': 'NumPy'}), (1, {'webpage': 'SciPy'})]
```

Now, let's add a web link from Node 1 (SciPy) to Node 0 (NumPy). Given a directed graph, we can insert an edge from `i` to `j` by running `G.add_edge(i, j)`.

Listing 18.12 Adding a single edge to a graph object

```
G.add_edge(1, 0)
print(nx.to_numpy_array(G))

[[0. 0.]
 [1. 0.]]
```

From the printed adjacency matrix, we can observe an edge going from Node 1 to Node 0. Unfortunately, our matrix printouts will grow cumbersome as other nodes are added. Tracking 1s and 0s in a 2D table is not the most intuitive way to display a network. What if, instead, we plotted the network directly? Our two nodes could be plotted as two points in 2D space, and our single edge could be plotted as a line segment connecting these points. Such a plot can easily be generated using Matplotlib. That is why our `G` object has a built-in `draw()` method for plotting the graph with the Matplotlib library. We call `G.draw()` to visualize our graph (figure 18.2).

Listing 18.13 Plotting a graph object

```
import matplotlib.pyplot as plt
np.random.seed(0) ← The locations of the nodes are determined
nx.draw(G)           using a randomized algorithm. We seed that
plt.show()            randomization to ensure consistent visualization.
```

← Per Matplotlib requirements, we must call
plt.show() to display the plotted results.

Our plotted graph could clearly use some improvement. First, we need to make our arrow bigger. This can be accomplished using the `arrowsize` parameter: passing `arrowsize=20` into `G.draw` will double the length and width of the plotted arrow. We should also add labels to the nodes; labels can be plotted with the `labels` parameter, which takes as input a dictionary mapping between node IDs and the intended labels. Listing 18.14 generates the mapping by running `{i: G.nodes[i]['webpage']}`

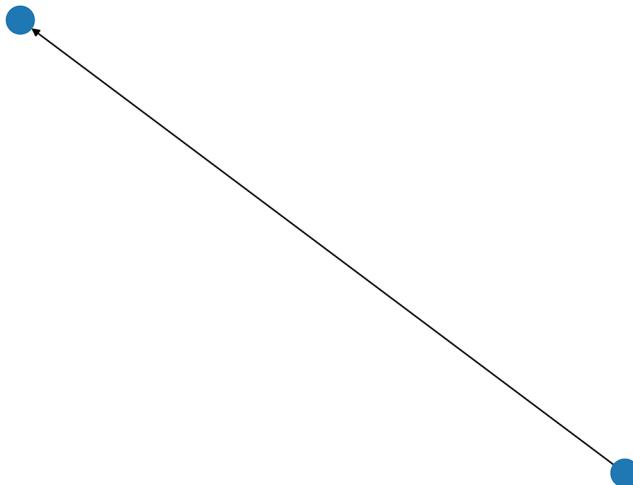


Figure 18.2 A visualized two-node directed graph. A barely visible directed arrow points from the lower node to the upper node.

for i in G.nodes} and then replots our network with the node labels and larger arrow (figure 18.3).

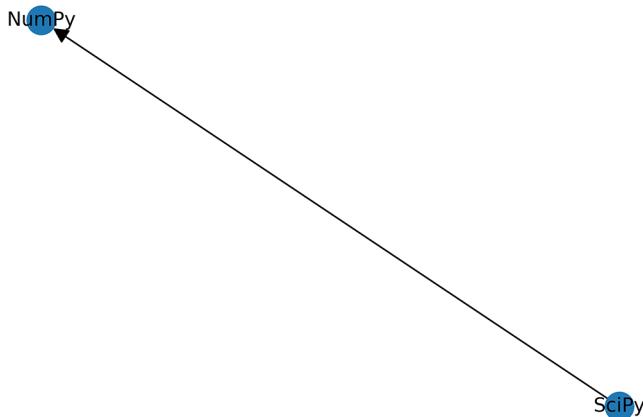


Figure 18.3 A visualized two-node directed graph. A directed arrow points from the lower node to the upper node. Both nodes are labeled, but the labels are hard to see.

NOTE Additionally, we can modify the node size by passing a `node_size` parameter into `nx.draw`. But for the time being, our nodes are appropriately sized at their default value of 300.

Listing 18.14 Tweaking the graph visualization

```
np.random.seed(0)
labels = {i: G.nodes[i]['webpage'] for i in G.nodes}
nx.draw(G, labels=labels, arrowsize=20)
plt.show()
```

The arrow is now bigger, and the node labels are partially visible. Unfortunately, these labels are obscured by the dark node color. We can make the labels more visible by changing the node color to something lighter, like cyan. We adjust the node color by passing `node_color="cyan"` into `G.draw` (figure 18.4).

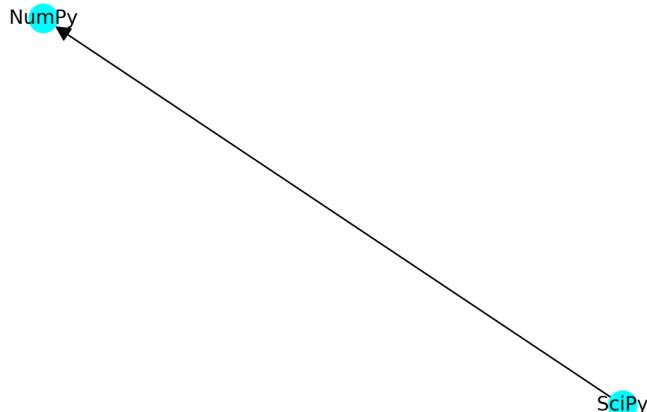


Figure 18.4 A visualized two-node directed graph. Both nodes are labeled. The color of the nodes has been adjusted so that the labels are clearly visible.

Listing 18.15 Altering the node color

```
np.random.seed(0)
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
```

In our latest plot, the labels are much more visible. We see the directed link from SciPy to NumPy. Now, let's add a reverse web link from NumPy to SciPy (figure 18.5).

Listing 18.16 Adding a back-link between web pages

```
np.random.seed(0)
G.add_edge(0, 1)
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
```

We are now ready to expand our network by adding two more web pages: Pandas and Matplotlib. These web pages will correspond to nodes with IDs 2 and 3, respectively. We can insert the two nodes individually by calling `G.add_node(2)` and then `G.add_node(3)`. Alternatively, we can insert the nodes simultaneously using the `G.add_nodes_from` method, which takes a list of nodes to be inserted into a graph. Thus, running `G.add_nodes_from([2, 3])` will add the proper node IDs to our network. However, these new nodes will lack any web page attribute assignments. Fortunately, the

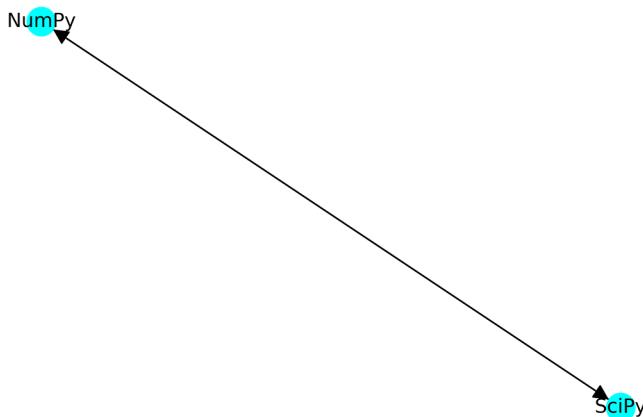


Figure 18.5 A visualized two-node directed graph. Pointed arrows are present at both ends of the edge between the nodes, indicating that the edge is bidirectional.

`G.add_nodes_from` method allows us to pass attribute values along with the node IDs. We simply need to pass `[(2, attributes_2), (3, attributes_3)]` into the method. Essentially, we must pass a list of tuples corresponding to both node IDs and attributes. The attributes are stored in a dictionary that maps attribute names with attribute values. For instance, the Pandas `attributes_2` dictionary will equal `{'webpage': 'Pandas'}`. Let's insert these nodes, together with their attributes, and output `G.nodes(data=True)` to verify that the new nodes are present.

Listing 18.17 Adding multiple nodes to a graph object

```

webpages = ['Pandas', 'Matplotlib']
new_nodes = [(i, {'webpage': webpage})
             for i, webpage in enumerate(webpages, 2)]
G.add_nodes_from(new_nodes)

print(f"We've added these nodes to our graph:\n{new_nodes}")
print('\nOur updated list of nodes is:')
print(G.nodes(data=True))

We've added these nodes to our graph:
[(2, {'webpage': 'Pandas'}), (3, {'webpage': 'Matplotlib'})]

Our updated list of nodes is:
[(0, {'webpage': 'NumPy'}), (1, {'webpage': 'SciPy'}), (2, {'webpage': 'Pandas'}), (3, {'webpage': 'Matplotlib'})]

```

We've added two more nodes. Let's visualize the updated graph (figure 18.6)



Figure 18.6 A visualized web-page directed graph. The Pandas and Matplotlib pages remain disconnected.

Listing 18.18 Plotting the updated four-node graph

```

np.random.seed(0)
labels = {i: G.nodes[i]['webpage'] for i in G.nodes}
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
  
```

Our current web link network is disconnected. We add two more web links: a link from Matplotlib (Node 3) to NumPy (Node 0), and a link from NumPy (Node 0) to Pandas (Node 2). These links can be added by calling `G.add_edge(3, 0)` and then `G.add_edge(0, 2)`. Alternatively, we can add the edges simultaneously using the `G.add_edges_from` method: this method takes as input a list of edges, where each edge is a tuple of the form `(i, j)`. Hence, running `G.add_edges_from([(0, 2), (3, 0)])` should insert the two new edges into our graph. The following code inserts these edges and regenerates our plot (figure 18.7).

Listing 18.19 Adding multiple edges to a graph object

```

np.random.seed(1)
G.add_edges_from([(0, 2), (3, 0)])
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
  
```

NOTE In our graph visualization, the nodes have been spread apart to emphasize the connectivity of their respective edges. This effect was achieved using a technique known as a *force-directed layout* visualization. A force-directed layout is based on physics. The nodes are modeled as negatively charged particles that are repelled by one another, and the edges are modeled as springs connecting the particles. As the connected nodes move apart, the springs

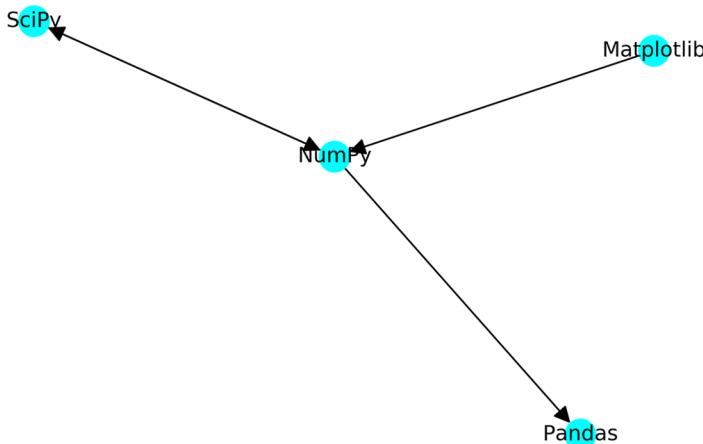


Figure 18.7 A visualized web page directed graph. Two inbound links point at the NumPy page. All the other pages have at most one inbound link.

begin to pull them back together. Modeling the physics equations in this system produces our graph visualization.

The NumPy web page appears in the center of our updated graph. Two web pages, SciPy and Matplotlib, have links that point to NumPy. All the other web pages have at most a single inbound link. More web content creators took the effort to reference the NumPy page than any other website: we can infer that NumPy is our most popular site since it has more inbound links than any other page. We've basically developed a simple metric for ranking websites on the internet. That metric equals the number of inbound edges pointing toward the site, also known as the *in-degree*. This is the opposite of the *out-degree*, which equals the number of edges pointing away from a site. By looking at our plotted graph, we can infer each website's in-degree automatically. However, we can also compute the in-degree directly from the graph's adjacency matrix. To demonstrate how, we first print our updated adjacency matrix.

Listing 18.20 Printing the updated adjacency matrix

```

adjacency_matrix = nx.to_numpy_array(G)
print(adjacency_matrix)

[[0.  1.  1.  0.]
 [1.  0.  0.  0.]
 [0.  0.  0.  0.]
 [1.  0.  0.  0.]]
  
```

As a reminder, the i th column in the matrix tracks the inbound edges of node i . The total number of inbound edges equals the number of ones in that column. Therefore,

the sum of values in the column is equal to the node's in-degree. For instance, Column 0 of our matrix equals [0, 1, 0, 1]. The sum of these values reveals an in-degree of 2, corresponding to the NumPy page. In general, executing `adjacency_matrix.sum(axis=0)` returns a vector of in-degrees. That vector's largest element corresponds to the most popular page in our internet graph.

NOTE Our simple ranking system assumes that all inbound links have equal weight, but this is not the case. An inbound link from a very popular website carries more weight since it drives more traffic to the site. In the next section, we present a more sophisticated ranking algorithm called PageRank that incorporates the popularity of traffic-directing websites.

Listing 18.21 Computing in-degrees using the adjacency matrix

```
in_degrees = adjacency_matrix.sum(axis=0)
for i, in_degree in enumerate(in_degrees):
    page = G.nodes[i]['webpage']
    print(f"{page} has an in-degree of {in_degree}")

top_page = G.nodes[in_degrees.argmax()]['webpage']
print(f"\n{top_page} is the most popular page.")

NumPy has an in-degree of 2.0
SciPy has an in-degree of 1.0
Pandas has an in-degree of 1.0
Matplotlib has an in-degree of 0.0

NumPy is the most popular page.
```

Alternatively, we can compute all in-degrees using the NetworkX `in_degree` method. Calling `G.in_degree(i)` returns the in-degree of node `i`, so we expect `G.in_degree(0)` to equal 2. Let's confirm.

Listing 18.22 Computing in-degrees using NetworkX

```
assert G.in_degree(0) == 2
```

In this code, we had to remember that `G.nodes[0]` corresponds to the NumPy page. Tracking the mapping between node IDs and page names can be slightly inconvenient, but we can bypass that inconvenience by assigning string IDs to individual nodes. For instance, given an empty graph `G2`, we insert our node IDs as strings by running `G2.add_nodes_from(['NumPy', 'SciPy', 'Matplotlib', 'Pandas'])`. Then calling `G2.in_degree('NumPy')` returns the in-degree of the NumPy page.

NOTE Storing node IDs as strings makes it more convenient to access certain nodes in the graph. However, the price of that convenience is a lack of correspondence between node IDs and indices in the adjacency matrix. As we'll learn, the adjacency matrix is indispensable for certain network tasks, so it is usually good practice to store the node IDs as indices and not strings.

Listing 18.23 Using strings as node IDs in a graph

```
G2 = nx.DiGraph()
G2.add_nodes_from(['NumPy', 'SciPy', 'Matplotlib', 'Pandas'])
G2.add_edges_from([('SciPy', 'NumPy'), ('SciPy', 'NumPy'),
                  ('NumPy', 'Pandas'), ('Matplotlib', 'NumPy')])
assert G2.in_degree('NumPy') == 2
```

Given a set of node attributes and a set of edges, we can generate the graph in just three lines of code. This pattern proves useful in many network problems. Commonly, when dealing with graph data, data scientists are provided with two files: one containing all the node attributes and another containing the linkage information. For instance, in this case study, we are provided with a table of FriendHook profiles as well as a table of existing friendships. These friendships serve as edges, which can be loaded by calling `add_edges_from`. Meanwhile, the profile information depicts attributes of each user in the friendship graph. After proper preparation, the profiles can be mapped back to the nodes by calling `add_nodes_from`. Thus, it's very straightforward to load the FriendHook graph into NetworkX for further analysis.

Introductory NetworkX graph methods

- `G = nx.DiGraph()`—Initializes a new directed graph.
- `G.add_node(i)`—Creates a new node with index `i`.
- `G.nodes[i] ['attribute'] = x`—Assigns an attribute `x` to node `i`.
- `G.add_node(i, attribute=x)`—Creates a new node `i` with attribute `x`.
- `G.add_nodes_from([(i, j)])`—Creates new nodes with indices `i` and `j`.
- `G.add_nodes_from([(i, {'a': x}), (j, {'a': y})])`—Creates new nodes with indices `i` and `j`. The attribute `a` of each new node is set to `x` and `y`, respectively.
- `G.add_edge(i, j)`—Creates an edge going from node `i` to node `j`.
- `G.add_edges_from([(i, j), (k, m)])`—Creates new edges going from `i` to `j` and from `k` to `m`.
- `nx.draw(G)`—Plots graph `G`.

So far, we've focused on directed graphs, in which traversal between nodes is limited. Each directed edge is like a one-way street that forbids travel in a certain direction. What if, instead, we treated every edge as though it were a two-way street? Our edges would be *undirected*, and we'd obtain an *undirected graph*. In an undirected graph, we can traverse connected nodes in either direction. This paradigm doesn't apply to the directed network underlying the internet, but it applies to the undirected network of roads connecting cities throughout the world. In the next subsection, we analyze road travel using undirected graphs. Later, we'll utilize these graphs to optimize the travel time between towns.

18.2 Utilizing undirected graphs to optimize the travel time between towns

In business logistics, product delivery time can impact certain critical decisions. Consider the following scenario, in which you've opened your own kombucha brewery. Your plan is to deliver batches of the delicious fermented tea to all the towns within a reasonable driving radius. More specifically, you'll only deliver to a town if it's within a two-hour driving distance of the brewery; otherwise, the gas costs won't justify the revenue from that delivery. A grocery store in a neighboring county is interested in regular deliveries. What is the fastest driving time between your brewery and that store?

Normally, you could obtain the answer by searching for directions on a smartphone, but we'll assume that existing tech solutions are not available (perhaps the area is remote and the local maps have not been scanned into an online database). In other words, you need to replicate the travel time computations carried out by existing smartphone tools. To do this, you consult a printed map of the local area. On the map, roads zigzag between towns, and some towns connect directly via a road. Conveniently, the travel times between connected towns are illustrated clearly on the map. We can model these connections using undirected graphs.

Suppose that a road connects two towns, Town 0 and Town 1. The driving time between the towns is 20 minutes. Let's record this information in an undirected graph. First, we generate the graph in NetworkX by running `nx.Graph()`. Next we add an undirected edge to that graph by executing `G.add_edge(0, 1)`. Finally, we add the drive time as an attribute of the inserted edge by running `G[0][1]['travel_time'] = 20`.

Listing 18.24 Creating a two-node undirected graph

```
G = nx.Graph()
G.add_edge(0, 1)
G[0][1]['travel_time'] = 20
```

Our travel time is an attribute of the edge `(0, 1)`. Given an attribute `k` of edge `(i, j)`, we can access that attribute by running `G[i][j][k]`, so we can access the travel time by running `G[0][1]['travel_time']`. In our undirected graph, the travel time between towns is not dependent on direction, so `G[1][0]['travel_time']` also equals 20.

Listing 18.25 Checking the edge attribute of a graph

```
for i, j in [(0, 1), (1, 0)]:
    travel_time = G[i][j]['travel_time']
    print(f"It takes {travel_time} minutes to drive from Town {i} to Town {j}.")
```

It takes 20 minutes to drive from Town 0 to Town 1.
It takes 20 minutes to drive from Town 1 to Town 0.

Towns 1 and 0 are connected on our map. However, not all towns are directly connected. Imagine an additional Town 2 that is connected to Town 1 but not Town 0. There is no road between Town 0 and Town 2, but there *is* a road between Town 1 and Town 2. The travel time on that road is 15 minutes. Let's add this new connection to our graph. We add the edge and travel time in a single line of code by executing `G.add_edge(1, 2, travel_time=15)`, and then we visualize the graph using `nx.draw`. We set the visualized node labels to equal the node IDs by passing `with_labels=True` into the `draw` function (figure 18.8).

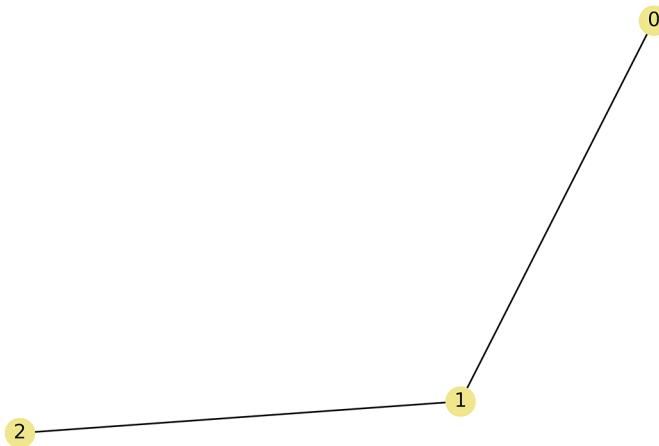


Figure 18.8 A visualized travel path from Town 0 to Town 2 by way of Town 1

Listing 18.26 Visualizing a path between multiple towns

```

np.random.seed(0)
G.add_edge(1, 2, travel_time=15)
nx.draw(G, with_labels=True, node_color='khaki')
plt.show()
  
```

Traveling from Town 0 to Town 2 requires us to first traverse Town 1. Hence, the total travel time is equal to the sum of `G[0][1]['travel_time']` and `G[1][2]['travel_time']`. Let's compute that travel time.

Listing 18.27 Computing the travel time between towns

```

travel_time = sum(G[i][1]['travel_time'] for i in [0, 2])
print(f"It takes {travel_time} minutes to drive from Town 0 to Town 2.")
  
```

It takes 35 minutes to drive from Town 0 to Town 2.

We've computed the fastest travel time between the two towns. Our computation was trivial since there is just one route between Town 0 and Town 2. However, in real life, many routes can exist between localized towns. Optimizing driving times between multiple towns is not so simple. To illustrate this point, let's build a graph containing more than a dozen towns spread across multiple counties. In our graph model, the travel time between towns will increase when towns are in different counties. We'll assume the following:

- Our towns are located in six different counties.
- Each county contains 3 to 10 towns.
- 90% of the towns in a single county are directly connected by roads. The average travel time on a county road is 20 minutes.
- 5% of the towns across different counties are directly connected by a road. The average travel time on an intra-county road is 45 minutes.

We'll now model this scenario. Then we'll devise an algorithm to compute the fastest travel time between any two towns in our complex network.

Common NetworkX methods and attribute assignments

- `G = nx.Graph()`—Initializes a new undirected graph
- `G.nodes[i] ['attribute'] = x`—Assigns an attribute `x` to node `i`
- `G[i][j] ['attribute'] = x`—Assigns an attribute `x` to edge `(i, j)`

18.2.1 Modeling a complex network of towns and counties

Let's start by modeling a single county that contains five towns. First, we insert five nodes into an empty graph. Each node is assigned a `county_id` attribute of 0, indicating that all nodes belong to the same county.

Listing 18.28 Modeling five towns in the same county

```
G = nx.Graph()
G.add_nodes_from((i, {'county_id': 0}) for i in range(5))
```

Next, we assign random roads to our five towns (figure 18.9). We iterate over each combination of node pairs and flip a biased coin. The coin lands on heads 90% of the time: whenever we see heads, we add an edge between the pair of nodes. Each edge's `travel_time` parameter is chosen at random by sampling from a normal distribution whose mean is 20.

NOTE As a reminder, the normal distribution is a bell-shaped curve commonly used to analyze random processes in probability and statistics. Refer back to section 6 for a more detailed discussion of that curve.

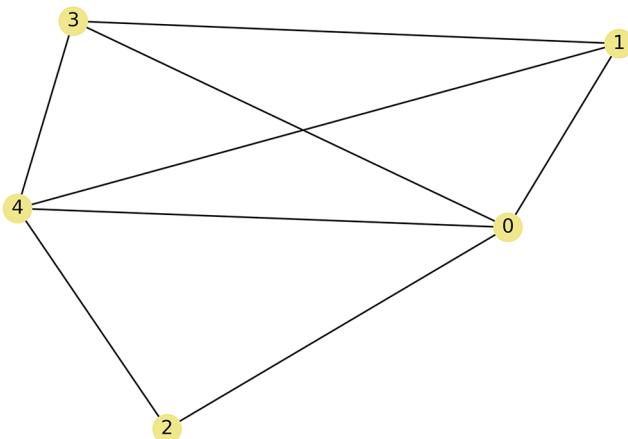


Figure 18.9 A randomly generated road network in a five-town county

Listing 18.29 Modeling random intra-county roads

```

import numpy as np
np.random.seed(0)

def add_random_edge(G, node1, node2, prob_road=0.9,
                     mean_drive_time=20):
    if np.random.binomial(1, prob_road):
        drive_time = np.random.normal(mean_drive_time)
        G.add_edge(node1, node2, travel_time=round(drive_time, 2))

nodes = list(G.nodes())
for node1 in nodes[:-1]:
    for node2 in nodes[node1 + 1:]:
        add_random_edge(G, node1, node2)

nx.draw(G, with_labels=True, node_color='khaki')
plt.show()

```

The function attempts to generate a random edge between node1 and node2 in graph G. The probability of edge insertion is equal to prob_road. If an edge is inserted, a randomized travel time attribute is assigned. The travel time is chosen from a normal distribution with a mean of mean_travel_time.

Flips a coin to determine whether an edge is inserted

Chooses the travel time from a normal distribution

We've connected most of the towns in County 0. In this same manner, we can randomly generate roads and towns for a second county: County 1. Here, we generate County 1 and store that output in a separate graph (figure 18.10). The number of towns in County 1 is a randomly chosen value between 3 and 10.

Listing 18.30 Modeling a second random county

```

np.random.seed(0)
def random_county(county_id):
    numTowns = np.random.randint(3, 10)
    G = nx.Graph()
    nodes = [(node_id, {'county_id': county_id})
              for node_id in range(numTowns)]

```

Generates a random county graph

Chooses the number of towns in the county at random from an integer range of 3 to 10

```

G.add_nodes_from(nodes)
for node1, _ in nodes[:-1]:
    for node2, _ in nodes[node1 + 1:]:
        add_random_edge(G, node1, node2)

return G

G2 = random_county(1)
nx.draw(G2, with_labels=True, node_color='khaki')
plt.show()

```

Randomly adds intra-county roads

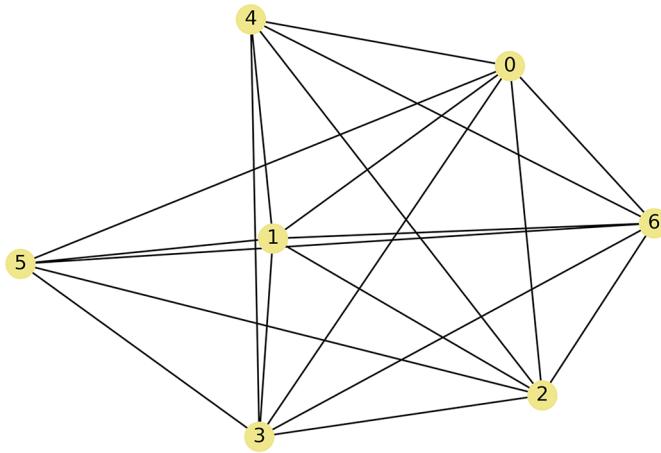


Figure 18.10 A randomly generated road network in a second county.
The number of towns in the county has also been chosen at random.

Currently, County 1 and County 2 are stored in two separate graphs: `G` and `G2`. We need to somehow combine these graphs. Merging the graphs is made more difficult by the shared IDs of nodes in `G` and `G2`. Fortunately, our task is simplified by the `nx.disjoint_union` function, which takes as input two graphs: `G` and `G2`. It then resets each node ID to a unique value between 0 and the total node count. Finally, it merges the two graphs. Here, we execute `nx.disjoint_union(G, G2)` and then plot the results (figure 18.11).

Listing 18.31 Merging two separate graphs

```

np.random.seed(0)
G = nx.disjoint_union(G, G2)
nx.draw(G, with_labels=True, node_color='khaki')
plt.show()

```

Our two counties appear in the same graph. Each town in the graph is assigned a unique ID. Now it's time to generate random roads between the counties (figure 18.12).

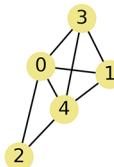
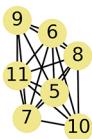


Figure 18.11 Two county networks merged together

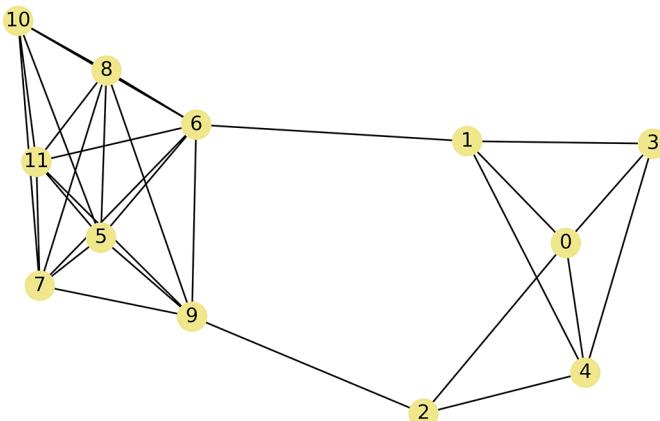


Figure 18.12 Two county networks connected by random roads

We iterate over all the pairs of inter-county nodes (in which $G[n1] ['county_id'] \neq G[n2] ['county_id']$). For each node pair, we apply `add_random_edge`. The probability of the edge is set low to 0.05, and the average travel time is set high to 90 minutes.

Listing 18.32 Adding random inter-county roads

```
np.random.seed(0)
def add_intercounty_edges(G):
    nodes = list(G.nodes(data=True))
    for node1, attributes1 in nodes[:-1]:
        county1 = attributes1['county_id']
        for node2, attributes2 in nodes[node1+1:]:
            if county1 != attributes2['county_id']:
                G.add_random_edge(node1, node2)
```

Adds random edges between nodes in graph G whose county IDs do not match

Iterates over every node and its associated attributes

Iterates over node pairs that we have not yet compared

```

        add_random_edge(G, node1, node2,
                          prob_road=0.05, mean_drive_time=45) ←
    return G

G = add_intercounty_edges(G)
np.random.seed(0)
nx.draw(G, with_labels=True, node_color='khaki')

```

Attempts to add a random inter-county edge

We've successfully simulated two interconnected counties. Now we simulate six interconnected counties (figure 18.13).

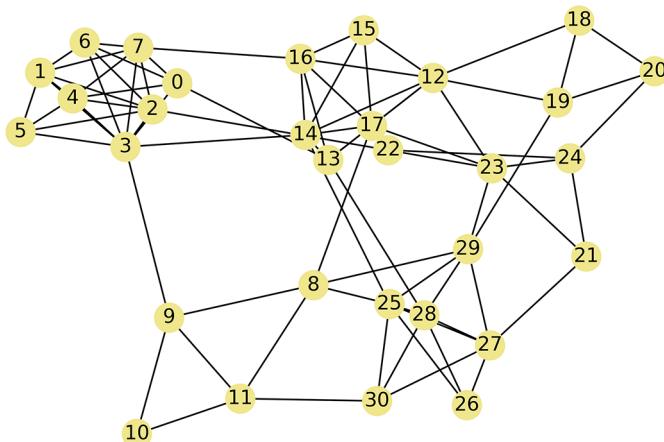


Figure 18.13 Six county networks connected by random roads

Listing 18.33 Simulating six interconnected counties

```

np.random.seed(1)
G = random_county(0)
for county_id in range(1, 6):
    G2 = random_county(county_id)
    G = nx.disjoint_union(G, G2)

G = add_intercounty_edges(G)
np.random.seed(1)
nx.draw(G, with_labels=True, node_color='khaki')
plt.show()

```

We've visualized our six-county graph, but individual counties are tricky to decipher in the visualization. Fortunately, we can improve our plot by coloring each node based on county ID. Doing so requires that we modify our input into the `node_color` parameter: rather than passing a single color string, we'll pass a list of color strings. The i th color in the list will correspond to the assigned color of the node at index i . The following code ensures that nodes in different counties receive different color assignments, while nodes that share a county are assigned the same color (figure 18.14).

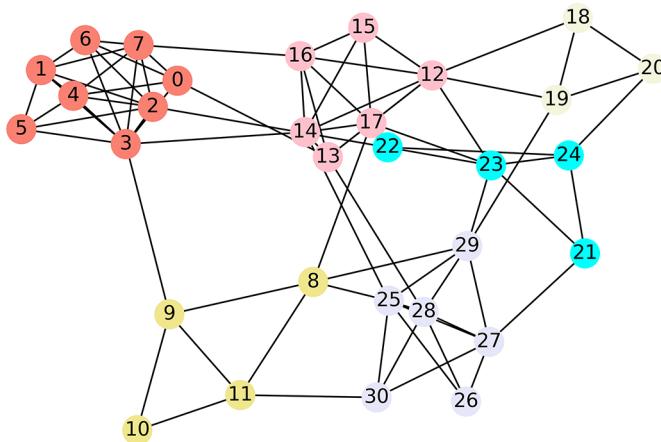


Figure 18.14 Six county networks connected by random roads. The individual towns have been colored based on county ID.

Listing 18.34 Coloring nodes by county

```
np.random.seed(1)
county_colors = ['salmon', 'khaki', 'pink', 'beige', 'cyan', 'lavender']
county_ids = [G.nodes[n]['county_id']
              for n in G.nodes]
node_colors = [county_colors[id_]
               for id_ in county_ids]
nx.draw(G, with_labels=True, node_color=node_colors)
plt.show()
```

The individual counties are now visible. Most counties form tight clumps in the network. Later, we extract these clumps automatically using network clustering. For now, we focus our attention on computing the fastest travel time between nodes.

Common NetworkX graph visualization functions

- `nx.draw(G)`—Plots graph G .
- `nx.draw(G, labels=True)`—Plots graph G with node labels. The labels equal the node IDs.
- `nx.draw(G, labels=ids_to_labels)`—Plots graph G with node labels. The nodes are labeled using a mapping between nodes IDs and labels. That mapping is specified by the `ids_to_labels` dictionary.
- `nx.draw(G, node_color=c)`—Plots graph G . All nodes are colored using color c .
- `nx.draw(G, node_color=ids_to_colors)`—Plots graph G . All nodes are colored using a mapping between node IDs and colors. That mapping is specified by the `ids_to_colors` dictionary.
- `nx.draw(G, arrowsize=20)`—Plots directed graph G while increasing the arrow size of the graph's directed edges.
- `nx.draw(G, node_size=20)`—Plots graph G while decreasing the node size from a default value of 300 to 20.

18.2.2 Computing the fastest travel time between nodes

Suppose our brewery is located in Town 0 and our potential client is located in Town 30. We want to determine the fastest travel time between Town 0 and Town 30. In the process, we need to compute the fastest travel time between Town 0 and every other town. How do we do this? Initially, all we know is the trivial travel time between Town 0 and itself: 0 minutes. Let's record this travel time in a `fastest_times` dictionary. Later, we populate this dictionary with the travel times to every town.

Listing 18.35 Tracking the fastest-known travel times

```
fastest_times = {0: 0}
```

Next, we can answer a simple question: what is the known travel distance between Town 0 and its neighboring towns? Neighbors, in this context, are towns with roads connecting to Town 0. In NetworkX, we can access the neighbors of Town 0 by executing `G.neighbors(0)`. That method call returns an iterable over the IDs of nodes connecting to node 0. Alternatively, we can access the neighbors by running `G[0]`. Here, we output the IDs of all the neighboring towns.

Listing 18.36 Accessing the neighbors of Town 0

```
neighbors = list(G.neighbors(0))
assert list(neighbors) == list(G[0])
print(f"The following towns connect directly with Town 0:\n{neighbors}")

The following towns connect directly with Town 0:
[3, 4, 6, 7, 13]
```

Now, we record the travel times between Town 0 and each of its five neighbors and use these times to update `fastest_times`. Additionally, we output the travel times in sorted order for further analysis.

Listing 18.37 Tracking the travel times to neighboring towns

```
time_to_neighbor = {n: G[0][n]['travel_time'] for n in neighbors}
fastest_times.update(time_to_neighbor)
for neighbor, travel_time in sorted(time_to_neighbor.items(),
                                     key=lambda x: x[1]):
    print(f"It takes {travel_time} minutes to drive from Town 0 to Town "
          f"{neighbor}.")
```

It takes 18.04 minutes to drive from Town 0 to Town 7.
 It takes 18.4 minutes to drive from Town 0 to Town 3.
 It takes 18.52 minutes to drive from Town 0 to Town 4.
 It takes 20.26 minutes to drive from Town 0 to Town 6.
 It takes 44.75 minutes to drive from Town 0 to Town 13.

It takes approximately 45 minutes to drive from Town 0 to Town 13. Is this the fastest travel time between these two towns? Not necessarily! A detour through another town may speed up travel, as shown in figure 18.15.

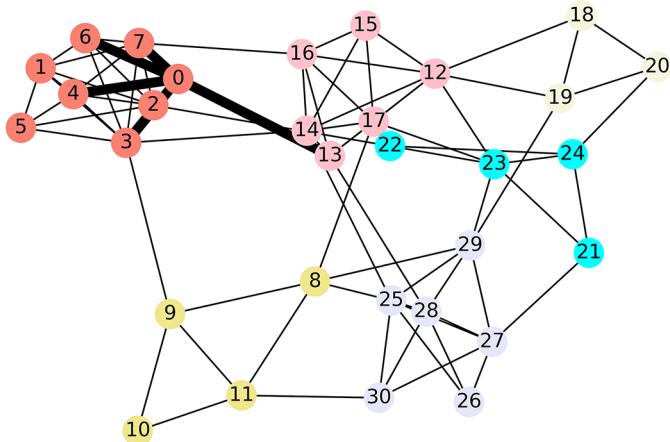


Figure 18.15 The roads connecting Town 0 to its neighbors are highlighted with thick, dark edges. The travel times across these five roads are known. It is possible that faster travel routes exist, but these routes would require additional detours.

Consider, for instance, a detour through Town 7. It's our most proximate town, with a drive time of only 18 minutes. What if there's a road between Town 7 and Town 13? If that road exists, and its travel time is under 27 minutes, then a faster route to Town 13 is possible! The same logic applies to Towns 3, 4, and 6. We can potentially shave minutes off our travel time if we examine the neighbors of Town 7. Let's carry out that examination like this:

- 1 Obtain the neighbors of Town 7.
- 2 Obtain the travel time between Town 7 and every neighboring Town N .
- 3 Add 18.04 minutes to the travel time obtained in the previous step. This represents the travel time between Town 0 and Town N when we take a detour through Town 7.
- 4 If N is present in `fastest_times`, check whether the detour is faster than `fastest_times[N]`. If an improvement is discovered, update `fastest_times` and print the faster travel time.
- 5 If N is not present in `fastest_times`, update that dictionary with the travel time computed in step 3. This represents the travel time between Town 0 and Town N when a direct road does not link the two towns.

The following code executes these steps.

Listing 18.38 Searching for faster detours through Town 7

```

The travel time between
Town 0 and town_id
def examine_detour(town_id):
    detour_found = False

    travel_time = fastest_times[town_id]
    for n in G[town_id]:
        detour_time = travel_time + G[town_id][n]['travel_time']
        if n in fastest_times:
            if detour_time < fastest_times[n]:
                detour_found = True
                print(f"A detour through Town {town_id} reduces "
                      f"travel-time to Town {n} from "
                      f"{fastest_times[n]:.2f} to "
                      f"{detour_time:.2f} minutes.")
                fastest_times[n] = detour_time
            else:
                fastest_times[n] = detour_time
        return detour_found

    if not examine_detour(7):
        print("No detours were found.")

addedTowns = len(fastest_times) - 6
print(f"We've computed travel-times to {addedTowns} additional towns.")

No detours were found.
We've computed travel-times to 3 additional towns.

```

Checks whether a detour through town_id alters the fastest known travel times from Town 0 to other towns

The detour time from Town 0 to a neighbor of town_id

Records the fastest known travel time from Town 0 to n

Checks if the detour improved the fastest known travel time from Town 0 to n

Checks how many new towns have been added to fastest_times in addition to the six towns that were initially present in the dictionary

We've uncovered travel times to three additional towns, but we have not found any faster detours for travel to neighbors of Town 0. Still, those detours might exist. Let's choose another viable detour candidate. We'll select a town that's proximate to Town 0, whose neighbors we have not examined. Doing so requires that we do the following:

- 1 Combine the neighbors of Town 0 and Town 7 into a pool of detour candidates. Note that both Town 0 and Town 7 will be present in that pool, thus necessitating the next step.
- 2 Remove Town 0 and Town 7 from the pool of candidates, leaving behind a set of unexamined towns.
- 3 Select an unexamined town with the fastest known travel time to Town 0.

Let's run these steps to choose our next detour candidate, using the logic visualized in figure 18.16.

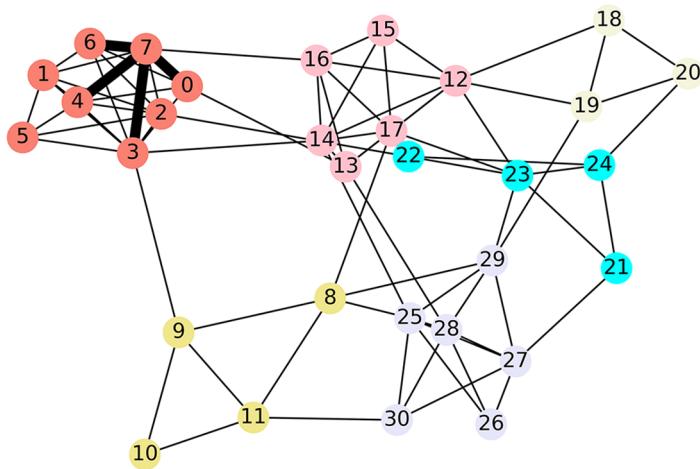


Figure 18.16 Direct detours taken through Town 7 to reach the neighbors of Town 0 are highlighted with thick, dark edges. The detours did not improve our driving times. Perhaps an additional detour through Town 3 will yield improved results.

Listing 18.39 Selecting an alternative detour candidate

Removes all previously examined towns from the candidate set

```
candidate_pool = set(G[0]) | set(G[7]) ←
examinedTowns = {0, 7}
unexaminedTowns = candidate_pool - examinedTowns
detour_candidate = min(unexaminedTowns,
key=lambda x: fastest_times[x]) ←
travel_time = fastest_times[detour_candidate]
print(f"Our next detour candidate is Town {detour_candidate}, "
      f"which is located {travel_time} minutes from Town 0.")
```

The pool of detour candidates combines the neighbors of Town 0 and Town 7. Note that both of these towns are neighbors of each other. Thus, they need to be removed from the candidate set.

Selects a detour candidate with the fastest known travel time to Town 0

Our next detour candidate is Town 3, which is located 18.4 minutes from Town 0.

Our next detour candidate is Town 3. We check Town 3 for detours: examining this town's neighbors may reveal new, unexamined towns. We insert all such towns into unexaminedTowns, which will allow us to track the remaining detour candidates for further analysis. Note that candidate tracking requires us to shift Town 3 from unexaminedTowns to examinedTowns after examination.

Listing 18.40 Searching for faster detours through Town 3

```
if not examine_detour(detour_candidate):
    print("No detours were found.") ←
Examines Town 3 for possible detours
```

```

def new_neighbors(town_id):
    return set(G[town_id]) - examinedTowns

def shift_to_examined(town_id):
    unexaminedTowns.remove(town_id)
    examinedTowns.add(town_id)

unexaminedTowns.update(new_neighbors(detour_candidate))
shift_to_examined(detour_candidate)
num_candidates = len(unexaminedTowns)
print(f"{num_candidates} detour candidates remain.")

No detours were found.
9 detour candidates remain.

```

Once again, no detours were discovered. However, nine detour candidates remain in our unexaminedTowns set. Let's examine the remaining candidates. Listing 18.41 iteratively does the following:

- 1 Select an unexamined town with the fastest known travel time to Town 0.
- 2 Check that town for detours using examine_detour.
- 3 Shift the town's ID from unexaminedTowns to examinedTowns.
- 4 Repeat step 1 if any unexamined towns remain. Terminate otherwise.

Listing 18.41 Examining every town for faster detours

```

The iteration continues until every possible town has been examined.
while unexaminedTowns:
    detour_candidate = min(unexaminedTowns,
                           key=lambda x: fastest_times[x])
    examine_detour(detour_candidate)
    shift_to_examined(detour_candidate)
    unexaminedTowns.update(new_neighbors(detour_candidate))

Selects a new detour candidate based on the fastest travel time to Town 0
Examines the candidate for detours
Adds previously unseen neighbors of the candidate to unexaminedTowns

```

Removes the candidate from unexamined-Towns

A detour through Town 14 reduces travel-time to Town 15 from 83.25 to 82.27 minutes.
A detour through Town 22 reduces travel-time to Town 23 from 111.21 to 102.38 minutes.
A detour through Town 28 reduces travel-time to Town 29 from 127.60 to 108.46 minutes.
A detour through Town 28 reduces travel-time to Town 30 from 126.46 to 109.61 minutes.
A detour through Town 19 reduces travel-time to Town 20 from 148.03 to 131.23 minutes.

We've examined the travel time to every single town and discovered five possible detours. Two of the detours are directed through Town 28: they reduce the travel times to Towns 29 and 30 from 2.1 hours to 1.8 hours, so both towns fall in a viable driving range for our kombucha brewery.

How many other towns are within two hours of Town 0? Let's find out.

Listing 18.42 Counting all the towns within a two-hour driving range

```
closeTowns = {town for town, drive_time in fastest_times.items()
              if drive_time <= 2 * 60}

num_closeTowns = len(closeTowns)
totalTowns = len(G.nodes)
print(f"{num_closeTowns} of our {totalTowns} towns are within two "
      "hours of our brewery.")

29 of our 31 towns are within two hours of our brewery.
```

All but two of our towns are within two hours of the brewery. We've figured this out by solving the *shortest path length problem*. The problem applies to graphs whose edges contain numeric attributes, which are called *edge weights*. Additionally, a sequence of node transitions in the graph is called a *path*. Each path traverses a sequence of edges. The sum of edge weights in that sequence is called the *path length*. The problem asks us to compute the shortest path lengths between a node N and every node in the graph. If all the edge weights are positive, we can compute these path lengths like this:

- 1 Create a dictionary of shortest path lengths. Initially, that dictionary equals $\{N: 0\}$.
- 2 Create a set of examined nodes. Initially, it is empty.
- 3 Create a set of nodes we wish to examine. Initially, it contains just N .
- 4 Remove an unexamined node U from our set of unexamined nodes. We pick a U whose path length to N is minimized.
- 5 Obtain all neighbors of U .
- 6 Compute the path length between each neighbor and N . Update the dictionary of shortest path lengths accordingly.
- 7 Add every neighbor that has not yet been examined to our set of unexamined nodes.
- 8 Add U to our set of examined nodes.
- 9 Repeat step 4 if any unexamined nodes remain. Terminate otherwise.

This shortest path length algorithm is included in NetworkX. Given graph G with an edge weight attribute of `weight`, we can compute all shortest path lengths from node N by running `nx.shortest_path_length(G, weight='weight', source=N)`. Here, we utilize the `shortest_path_length` function to compute `fastest_times` in a single line of code.

Listing 18.43 Computing shortest path lengths with NetworkX

```
shortest_lengths = nx.shortest_path_length(G, weight='travel_time',
                                             source=0)
for town, path_length in shortest_lengths.items():
    assert fastest_times[town] == path_length
```

Our shortest path length algorithm doesn't actually return the shortest path. However, in real-world circumstances, we want to know the path that minimizes the distance between nodes. For instance, simply knowing the fastest travel time between Town 0 and Town 30 is insufficient: we also need the driving directions that will get us there in under two hours. Fortunately, the shortest path length algorithm can be easily modified to track the shortest path. All we need to do is add a dictionary structure that tracks the transition between nodes. The actual sequence of traversed nodes can be represented by a list. For brevity's sake, we forgo defining a shortest path tracker from scratch; however, you are encouraged to code a shortest path function and compare its outputs to the built-in NetworkX shortest_path function. Calling `nx.shortest_path(G, weight='weight', source=N)` computes all shortest paths from node N to every node in G. Hence, executing `nx.shortest_path(G, weight='travel_time', source=0)` [30] should return the fastest travel route between Town 0 and Town 30. We'll now print that route, which is also displayed in figure 18.17.

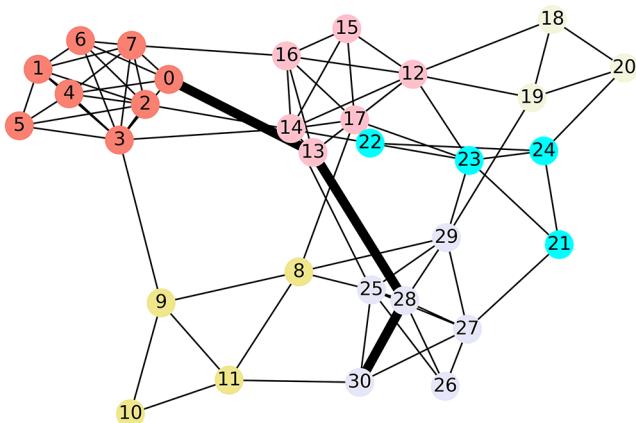


Figure 18.17 The shortest path between Town 0 and Town 30 is highlighted on the graph with thick, dark edges. The path goes from Town 0 to Town 13, followed by Towns 28 and 30. It's worth noting that alternative paths exist in the graph: for example, we can travel from Town 13 to Town 25 and then to Town 30. However, our highlighted path is guaranteed to have the shortest possible path length.

Listing 18.44 Computing shortest paths with NetworkX

```
shortest_path = nx.shortest_path(G, weight='travel_time', source=0) [30]
print(shortest_path)

[0, 13, 28, 30]
```

Driving time is minimized if we travel from Town 0 to Town 13 to Town 28 and finally to Town 30. We expect that travel time to equal `fastest_times[30]`. Let's confirm.

Listing 18.45 Verifying the length of a shortest path

```
travel_time = 0
for i, town_a in enumerate(shortest_path[:-1]):
    town_b = shortest_path[i + 1]
    travel_time += G[town_a][town_b]['travel_time']
```

```
print("The fastest travel time between Town 0 and Town 30 is "
      f"{travel_time} minutes.")
assert travel_time == fastest_times[30]
```

The fastest travel time between Town 0 and Town 30 is 109.61 minutes.

Basic network theory allows us to optimize the travel paths between geolocations. In the following section, we build on that theory to develop more advanced techniques. More precisely, we simulate the traffic flow across the network of towns. The simulation will allow us to uncover the most central towns in the graph. Later, we use these traffic simulations to cluster the towns into distinct counties and illustrate how this clustering technique can be used to identify friend groups in a social graph.

Common NetworkX path-related techniques

- `G.neighbors(i)`—Returns all neighbors of node *i*.
- `G[i]`—Returns all neighbors of node *i*.
- `G[i] [j] ['weight']`—Returns the length of a single transition path between neighboring nodes *i* and *j*.
- `nx.shortest_path_length(G, weight='weight', source=N)`—Returns the dictionary of shortest path lengths from node *N* to all accessible nodes in the graph. The `weight` attribute is used to measure the path length.
- `nx.shortest_path(G, weight='weight', source=N)`—Returns the dictionary of shortest paths from node *N* to all accessible nodes in the graph.

Summary

- *Network theory* is the study of connections between objects. A collection of objects and their dispersed connections is called either a *network* or a *graph*. The objects are called *nodes*, and the connections are called *edges*.
- If an edge has a specific direction, it is called a *directed edge*. Graphs with directed edges are called *directed graphs*. If a graph is not directed, it is called an *undirected graph*.
- We can represent a graph as a binary matrix *M*, where $M[i][j] = 1$ if an edge exists between node *i* and node *j*. This matrix representation of a graph is known as an *adjacency matrix*.
- In a directed graph, we can count the inbound and outbound edges for each node. The number of inbound edges is called the *in-degree*, and the number of outbound edges is called the *out-degree*. In certain graphs, the in-degree serves as a measure of a node's popularity. We can compute the in-degree by summing over the rows in the adjacency matrix.
- We can use graph theory to optimize travel between nodes. A sequence of node transitions is called a *path*. A length can be associated with that path if each

edge has a numeric attribute assigned. That numeric attribute is called an *edge weight*. The sum of edge weights across the node sequence in a path is called the *path length*. The *shortest path length* problem attempts to minimize the path lengths from node N to all other nodes in the graph. If the edge weights are positive, the path lengths can be minimized algorithmically.

Dynamic graph theory techniques for node ranking and social network analysis

This section covers

- Finding the most central network locations
- Clustering the connections in a network
- Understanding social graph analysis

In the previous section, we investigated several types of graphs. We examined web pages connected by directed links and also a network of roads spanning multiple counties. In our analysis, we've mostly treated the network as frozen, static objects—we've counted neighboring nodes as though they were frozen clouds in a photograph. In real life, clouds are constantly in motion, and so are many networks. Most networks worth studying are perpetually buzzing with dynamic activity. Cars race across networks of roads, causing traffic congestion near popular towns. In that same vein, web traffic flows across the internet as billions of users explore the many web links. Our social networks are also flowing with activity as gossip, rumors, and cultural memes spread across tight circles of close friends. Understanding this dynamic flow can help uncover friend groups in an automated manner. Understanding the flow can also help us identify the most heavily trafficked web pages on the internet. Such modeling of dynamic network activity is critical to

the function of many large tech organizations. In fact, one of the modeling methods presented in this section led to the founding of a trillion-dollar company.

The dynamic flow (of people, cars, etc.) across a graph is an inherently random process, so it can be studied using random simulations similar to those presented in section 3. In the early part of this section, we utilize random simulations to study the traffic flow of cars. Then we attempt to compute the traffic probabilities more efficiently using matrix multiplication. Later, we use our matrix analysis to uncover clusters of communities with heavy traffic. We then apply our clustering technique to uncover groups of friends in social networks.

Let's get started. We begin with the straightforward problem of uncovering heavily trafficked towns based on traffic simulations.

19.1 **Uncovering central nodes based on expected traffic in a network**

In the previous section, we simulated a road network connecting 31 towns in 6 different counties (figure 19.1). We stored that network in a graph called G . Our goal was to optimize business delivery travel times across the 31 towns. Let's explore this scenario further.

Suppose our business is growing at an impressive rate. We wish to expand our customer base by putting up a billboard advertisement in one of the local towns represented by $G.nodes$. To maximize billboard views, we'll choose the town with the heaviest traffic. Intuitively, traffic is determined by the number of cars that pass through town every day. Can we rank the 31 towns in $G.nodes$ based on the expected daily traffic? Yes, we can! Using simple modeling, we can predict traffic flow from the network of roads between the towns. Later, we'll expand on these traffic-flow techniques to identify local counties in an automated manner.

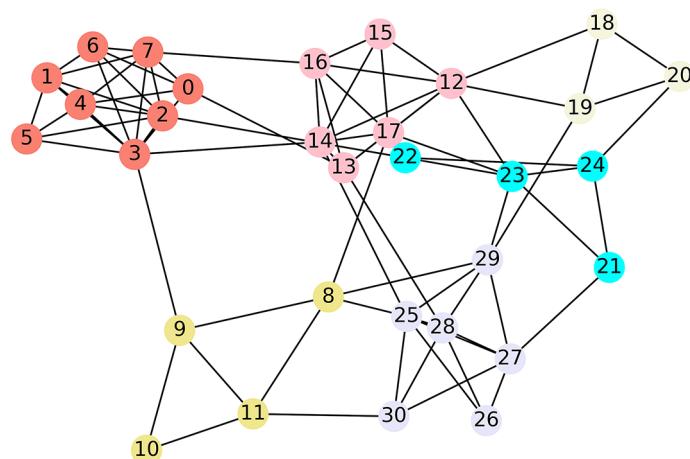


Figure 19.1 A simulated node network from section 18, which is stored in graph G . The roads connect 31 towns that are spread across six counties. Each town is colored based on its county ID.

We need a way of ranking the towns based on expected traffic. Naively, we could simply count the inbound roads into each town: a town with five roads can receive traffic from five different directions, while a town with just one road is more limited in its traffic flow. The road count is analogous to our in-degree ranking of websites introduced in section 18. As a reminder, the in-degree of a node is the number of directed edges pointing at a node. However, unlike the website graph, our network of roads is undirected: there's no distinction between inbound edges and outbound edges. Thus, there's no distinction between a node's in-degree and out-degree; both values are equal, so the edge count of a node in an undirected graph is simply called the node's *degree*. We can compute the degree of any node i by summing over the i th column of the graph's adjacency matrix, or we can measure the degree by running `len(G.nodes[i])`. Alternatively, we can utilize the NetworkX `degree` method by calling `G.degree(i)`. Here, we use all these techniques to count the roads passing through Town 0.

Listing 19.1 Computing the degree of a single node

```
adjacency_matrix = nx.to_numpy_array(G)
degree_town_0 = adjacency_matrix[:, 0].sum()
assert degree_town_0 == len(G[0])
assert degree_town_0 == G.degree(0)
print(f"Town 0 is connected by {degree_town_0} roads.")

Town 0 is connected by 5 roads.
```

Using their degrees, we rank our nodes based on importance. In graph theory, any measure of a node's importance is commonly called *node centrality*, and ranked importance based on a node's degree is called the *degree of centrality*. We now select the node with the highest degree of centrality in G : this central node will serve as our initial choice for the billboard's location (figure 19.2).

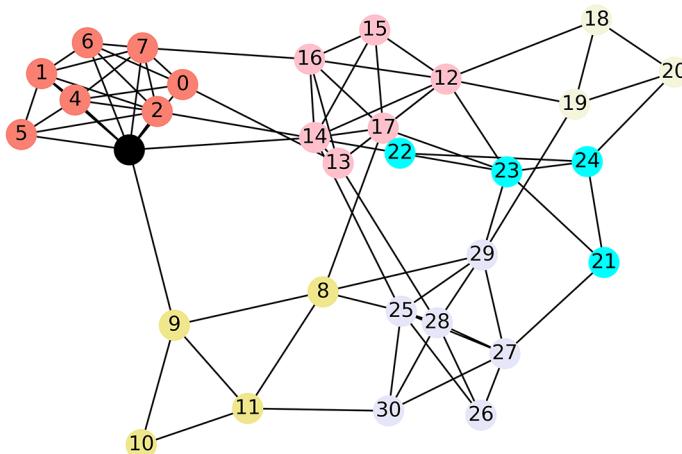


Figure 19.2 A network of roads between different towns. Town 3 has the highest degree of centrality and is colored black.

Listing 19.2 Selecting a central node using degree of centrality

```
np.random.seed(1)
central_town = adjacency_matrix.sum(axis=0).argmax()
degree = G.degree(central_town)
print(f"Town {central_town} is our most central town. It has {degree} "
      "connecting roads.")
node_colors[central_town] = 'k'
nx.draw(G, with_labels=True, node_color=node_colors)
plt.show()
```

Town 3 is our most central town. It has 9 connecting roads.

Town 3 is our most central town. Roads connect it to nine different towns and three different counties. How does Town 3 compare with the second-most-central town? We'll quickly check by outputting the second-highest degree in G.

Listing 19.3 Selecting a node with the second-highest degree of centrality

```
second_town = sorted(G.nodes, key=lambda x: G.degree(x), reverse=True)[1]
second_degree = G.degree(second_town)
print(f"Town {second_town} has {second_degree} connecting roads.")
```

Town 12 has 8 connecting roads.

Town 12 has eight connecting roads—it lags behind Town 3 by just one road. What would we do if these two towns had equal degrees? Let's challenge ourselves to find out. In figure 19.2, we see a road connecting Town 3 and Town 9. Suppose that road is closed due to disrepair. That closure necessitates the removal of an edge in G. Running G.remove(3, 9) removes the edge between nodes 3 and 9, so the degree of Town 3 shifts to equal the degree of Town 12. There are also other important structural changes to the network. Here, we visualize these changes (figure 19.3).

Listing 19.4 Removing an edge from the most central node

```
np.random.seed(1)
G.remove_edge(3, 9)
assert G.degree(3) == G.degree(12)
nx.draw(G, with_labels=True, node_color=node_colors)
plt.show()
```

After edge deletion, Towns 3
and 12 share the same
degree of centrality.

Removal of the road has partially isolated Town 3 as well as its neighboring towns. Town 3 is in County 0, which encompasses Towns 0 through 7. Previously, a single road passing through Town 3 linked County 0 to County 1; now that road has been eliminated, so Town 3 is less accessible than it was before. This is in contrast to Town 12, which continues to be the neighbor of multiple different counties.

Town 3 is now less central than Town 12, but the degrees of both towns are equal. We've exposed a significant flaw of the degree of centrality: connecting roads don't matter if they don't lead anywhere important.

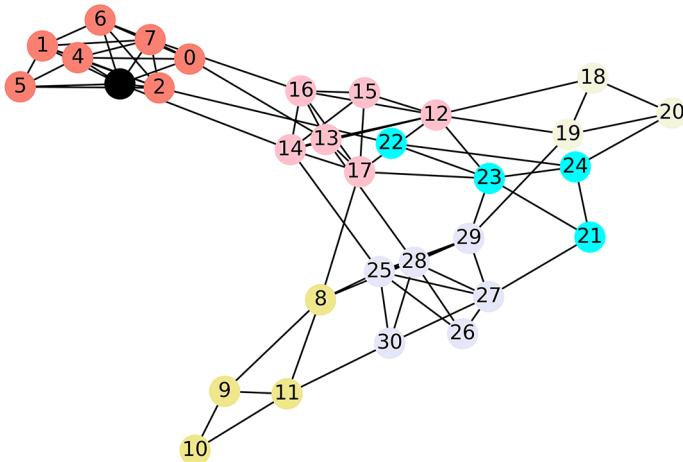


Figure 19.3 A network of roads between different towns after a road closure. Towns 3 and 12 now share the same degree of centrality. Despite their shared degree, Town 12 appears to be in a more central position in the graph. Its county borders multiple other counties. Meanwhile, the closed road has partially isolated Town 3 from the outside world.

Imagine if a town has 1,000 roads, all of which lead to dead ends. Now imagine a town with just four roads, but each road leads to a large metropolitan city. We would expect the second town to have heavier traffic than the first, despite the extreme difference in degrees. Similarly, we expect Town 12 to garner more traffic than Town 3, even though their degrees are equal. In fact, we can quantitate these differences using random simulations. In the next subsection, we measure town centrality by simulating traffic flows between towns.

19.1.1 Measuring centrality using traffic simulations

We'll shortly simulate traffic in our network by having 20,000 simulated cars drive randomly around our 31 towns. However, first we need to simulate the random path of a single car. The car will start its journey in a random town i . Then the driver will randomly select one of the $G.\text{degree}(i)$ roads that cut through town and pay a visit to a random neighboring town of i . Next, another random road will be selected. The process will repeat until the car has driven through 10 towns. Let's define a `random_drive` function to run this simulation on graph G ; the function returns the final location of the car.

NOTE In graph theory, this type of random traversal between nodes is called a *random walk*.

Listing 19.5 Simulating the random route of a single car

```

np.random.seed(0)
def random_drive(num_stops=10):
    town = np.random.choice(G.nodes)
    for _ in range(num_stops):
        town = np.random.choice(G[town])
    return town

destination = random_drive()
print(f"After driving randomly, the car has reached Town {destination}.")

```

After driving randomly, the car has reached Town 24.

Listing 19.6 repeats this simulation with 20,000 cars and counts the number of cars in each of the 31 towns. That car count represents the traffic in each town. We print the traffic in the most heavily visited town. We also time our 20,000 iterations to get a sense of the running-time costs associated with traffic simulations.

NOTE Our simulation is greatly oversimplified. In real life, people don't drive randomly from town to town: there is a lot of traffic in certain areas because they are between places people must regularly go, where they find lots of housing, employment opportunities, retailers, etc. But our simplification isn't detrimental; it's beneficial! Our model generalizes beyond just car traffic. It can be applied to web traffic and also to the flow of social interactions. Shortly, we'll expand our analysis to these other categories of graphs—and that expansion would not have been possible if our model was less simple and more concrete.

Listing 19.6 Simulating traffic using 20,000 cars

```

import time
np.random.seed(0)
car_counts = np.zeros(len(G.nodes))      ← Stores the traffic counts in an array rather than a dictionary to more easily vectorize these counts in subsequent code
num_cars = 20000

start_time = time.time()
for _ in range(num_cars):
    destination = random_drive()
    car_counts[destination] += 1

central_town = car_counts.argmax()
traffic = car_counts[central_town]
running_time = time.time() - start_time
print(f"We ran a {running_time:.2f} second simulation.")
print(f"Town {central_town} has the most traffic.")
print(f"There are {traffic:.0f} cars in that town.")

```

We ran a 3.47 second simulation.
 Town 12 has the most traffic.
 There are 1015 cars in that town.

Town 12 has the most traffic, with over 1,000 cars. This is not surprising, given that Town 12 and Town 3 share the highest degree of centrality. Based on our previous discussion, we also expect Town 12 to have heavier traffic than Town 3. Let's confirm.

Listing 19.7 Checking the traffic in Town 3

```
print(f"There are {car_counts[3]:.0f} cars in Town 3.")

There are 934 cars in Town 3.
```

Our expectations are verified. Town 3 has fewer than 1000 cars. We should note that car counts can be cumbersome to compare, especially when `num_cars` is large. Hence, it's preferable to replace these direct counts with probabilities through division by the simulation count. If we execute `car_counts / num_cars`, we obtain a probability array: each *i*th probability equals the likelihood that a randomly traveling car winds up in Town *i*. Let's print these probabilities for Towns 12 and 3.

Listing 19.8 Converting traffic counts to probabilities

```
probabilities = car_counts / num_cars
for i in [12, 3]:
    prob = probabilities[i]
    print(f"The probability of winding up in Town {i} is {prob:.3f}.")
```

The probability of winding up in Town 12 is 0.051.
 The probability of winding up in Town 3 is 0.047.

According to our random simulation, we'll wind up in Town 12 5.1% of the time and Town 3 just 4.7% of the time. Hence, we've shown that Town 12 is more central than Town 3. Unfortunately, our simulation process is slow and doesn't scale well to larger graphs.

NOTE Our simulations took 3.47 seconds to run. This seems like a reasonable running time, but larger graphs will require more simulations to estimate travel probabilities. This is due to the law of large numbers, which we introduced in section 4. A graph with 1,000 times more nodes would require 1,000 times more simulations, which would increase our running time to approximately one hour.

Can we compute these probabilities directly without simulating the flow of 20,000 cars? Yes! In the next section, we show how to compute the traffic probabilities using straightforward matrix multiplication.

19.2 Computing travel probabilities using matrix multiplication

Our traffic simulation can be modeled mathematically using matrices and vectors. We'll break this process into simple, manageable parts. Consider, for instance, a car that is about to leave Town 0 for one of the neighboring towns. There are $G.\text{degree}(0)$ neighboring towns to choose from, so the probability of traveling from Town 0 to any neighboring town is $1 / G.\text{degree}(0)$. Let's compute this probability.

Listing 19.9 Computing the probability of travel to a neighboring town

```
num_neighbors = G.degree(0)
prob_travel = 1 / num_neighbors
print("The probability of traveling from Town 0 to one of its "
      f"{G.degree(0)} neighboring towns is {prob_travel}")
```

The probability of traveling from Town 0 to one of its 5 neighboring towns is 0.2

If we're in Town 0 and Town i is a neighboring town, there's a 20% chance of us traveling from Town 0 to Town i . Of course, if Town i is not a neighboring town, the probability drops to 0.0. We can track the probabilities for every possible i using a vector v . The value of $v[i]$ will equal 0.2 if i is in $G[0]$ and 0 otherwise. Vector v is called a *transition vector* since it tracks the probability of transitioning from Town 0 to other towns. There are multiple ways to compute the transition vector:

- Run `np.array([0.2 if i in G[0] else 0 for i in G.nodes])`. Each i th element will equal 0.2 if i is in $G[0]$ and 0 otherwise.
- Run `np.array([1 if i in G[0] else 0 for i in G.nodes]) * 0.2`. Here, we are simply multiplying 0.2 by the binary vector that tracks the presence or absence of edges linking to $G[0]$.
- Run $M[:, 0] * 0.2$, where M is the adjacency matrix. Each adjacency matrix column tracks the binary presence or absence of edges between nodes, so column 0 of M will equal the array in the previous example.

The third computation is the simplest. Of course, 0.2 is equal to $1 / G.\text{degree}(0)$. As we discussed at the beginning of this section, the degree can also be computed by summing across an adjacency matrix column. Thus, we can also compute the transitional vector by running `M[:, 0] / M[:, 0].sum()`. Listing 19.10 computes the transition vector using all the listed methodologies.

NOTE Currently, an adjacency matrix M is stored with an `adjacency_matrix` variable. However, that matrix does not take into account the deleted edge between Town 3 and Town 9, so we recompute the matrix by running `adjacency_matrix = nx.to_numpy_array(G)`.

Listing 19.10 Computing a transition vector

```

transition_vector = np.array([0.2 if i in G[0] else 0 for i in G.nodes])

adjacency_matrix = nx.to_numpy_array(G)
v2 = np.array([1 if i in G[0] else 0 for i in G.nodes]) * 0.2
v3 = adjacency_matrix[:,0] * 0.2
v4 = adjacency_matrix[:,0] / adjacency_matrix[:,0].sum() ← Recomputes the
                                                               adjacency matrix to
                                                               take into account
                                                               our earlier edge
                                                               deletion

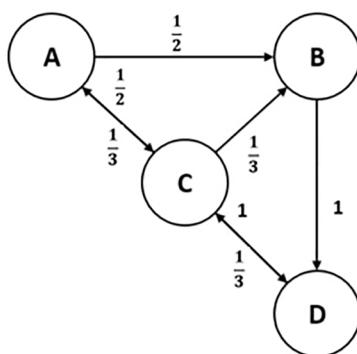
for v in [v2, v3, v4]:
    assert np.array_equal(transition_vector, v) ← Computes the transition
                                                   vector directly from the
                                                   adjacency matrix

print(transition_vector) ← All four computed versions of the
                           transition vector are identical.

[0.  0.  0.  0.2 0.2 0.  0.2 0.2 0.  0.  0.  0.  0.  0.2 0.  0.  0.  0.
 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

```

We can compute the transition vector for any Town i by running $M[:,i] / M[:,i].sum()$, where M is the adjacency matrix. Furthermore, we can compute these vectors all at once by running $M / M.sum(axis=0)$. The operation divides each column of the adjacency matrix by the associated degree. The end result is a matrix whose columns correspond to transition vectors. This matrix, which is illustrated in figure 19.4, is referred to as a *transition matrix*. It is also commonly called the *Markov matrix*, named after Andrey Markov, a Russian mathematician who studied random processes. We now compute the transition matrix: based on our expectation, the output's column 0 should equal Town 0's `transition_vector`.



$$\begin{matrix} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \end{matrix} & \left[\begin{matrix} 0 & 0 & \frac{1}{3} & 0 \\ \frac{1}{2} & 0 & \frac{1}{3} & 0 \\ \frac{1}{2} & 0 & 0 & 1 \\ 0 & 1 & \frac{1}{3} & 0 \end{matrix} \right] \end{matrix}$$

Figure 19.4 If M is the adjacency matrix, then $M / M.sum(axis=0)$ equals the transition matrix, even if the adjacencies are directed. This figure shows a directed graph. Edges are marked with the transition probabilities. These probabilities are also displayed in a matrix equal to $M / M.sum(axis=0)$. Each column in the matrix is a transition vector whose probabilities sum to 1.0. According to the matrix, the probability of travel from A to C is $1/2$, and the probability of travel from C to A is $1/3$.

Listing 19.11 Computing a transition matrix

```
transition_matrix = adjacency_matrix / adjacency_matrix.sum(axis=0)
assert np.array_equal(transition_vector, transition_matrix[:, 0])
```

Our transition matrix has a fascinating property: it allows us to compute the traveling probability to every town in just a few lines of code! If we want to know the probability of winding up in Town i after 10 stops, then we simply need to do the following:

- 1 Initialize a vector v , where v equals `np.ones(31) / 31`.
- 2 Update v to equal `transition_matrix @ v` over 10 iterations.
- 3 Return $v[i]$.

Later, we derive this amazing property from scratch. For now, let's prove our claims by computing the travel probabilities to Towns 12 and 3 using matrix multiplication. We expect these probabilities to equal 0.051 and 0.047, based on our previous observations.

Listing 19.12 Computing travel probabilities using the transition matrix

```
v = np.ones(31) / 31
for _ in range(10):
    v = transition_matrix @ v

for i in [12, 3]:
    print(f"The probability of winding up in Town {i} is {v[i]:.3f}.")
```

The probability of winding up in Town 12 is 0.051.
The probability of winding up in Town 3 is 0.047.

Our expectations are confirmed.

We can model traffic flow using a series of matrix multiplications. These multiplications serve as the basis for *PageRank centrality*, which is the most profitable node-importance measure in history. PageRank centrality was invented by the founders of Google; they used it to rank web pages by modeling a user's online journey as a series of random clicks through the internet's graph. These page clicks are analogous to a car that drives through randomly chosen towns. More popular web pages have a higher likelihood of visits. This insight allowed Google to uncover relevant websites in a purely automated manner. Google was thus able to outperform its competition and become a trillion-dollar company. Sometimes, data science can pay off nicely.

PageRank centrality is easy to compute but not so easy to derive. Nonetheless, with basic probability theory, we can demonstrate why repeated `transition_matrix` multiplications directly yield the travel probabilities.

NOTE If you're not interested in the PageRank centrality derivation, skip ahead to the next subsection. It describes PageRank usage in NetworkX.

19.2.1 Deriving PageRank centrality from probability theory

We know that `transition_matrix[i][j]` equals the probability of traveling from Town j directly to Town i , but this assumes that our car is actually located in Town j . What if the car's location is not certain? For instance, what if there is just a 50% chance that the car is located in Town j ? Under such circumstances, the travel probability equals $0.5 * \text{transition_matrix}[i][j]$. Generally, if the probability of our current location is p , then the probability of travel from the current location j to new location i equals $p * \text{transition_matrix}[i][j]$.

Suppose a car begins its journey in a random town and travels one town over. What is the probability that the car will travel from Town 3 to Town 0? Well, the car can start the journey in any of 31 different towns, so the probability of starting in Town 3 is $1 / 31$. Thus the probability of traveling from Town 3 to Town 0 is `transition_matrix[0][3] / 31`.

Listing 19.13 Computing a travel likelihood from a random starting location

```
prob = transition_matrix[0][3] / 31
print("Probability of starting in Town 3 and driving to Town 0 is "
      f"{prob:.2}")

Probability of starting in Town 3 and driving to Town 0 is 0.004
```

There are multiple ways of reaching Town 0 directly from a random starting location. Let's print all nonzero instances of `transition_matrix[0][i] / 31` for every possible Town i .

Listing 19.14 Computing travel likelihoods of random routes leading to Town 0

```
for i in range(31):
    prob = transition_matrix[0][i] / 31
    if not prob:
        continue

    print(f"Probability of starting in Town {i} and driving to Town 0 is "
          f"{prob:.2}")

print("\nAll remaining transition probabilities are 0.0")

Probability of starting in Town 3 and driving to Town 0 is 0.004
Probability of starting in Town 4 and driving to Town 0 is 0.0054
Probability of starting in Town 6 and driving to Town 0 is 0.0065
Probability of starting in Town 7 and driving to Town 0 is 0.0046
Probability of starting in Town 13 and driving to Town 0 is 0.0054

All remaining transition probabilities are 0.0
```

Five different routes take us to Town 0. Each route has a different probability, and the sum of these probabilities equals the likelihood of starting at any random town and traveling directly to Town 0 (figure 19.5). We'll now compute that likelihood. Furthermore,

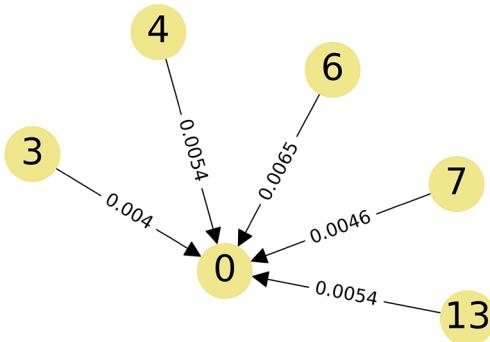


Figure 19.5 Five different routes take us from a random initial town directly to Town 0. Each route has a small probability assigned. Summing these values gives the probability of starting at a random town and traveling directly to Town 0.

we'll compare the likelihood to the result of random simulations. We run the simulations by executing `random_drive(num_stops=1)` 50,000 times, which yields the frequency with which Town 0 appears as the first stop on a randomized journey. We expect that frequency to approximate our probability sum.

Listing 19.15 Computing the probability that the first stop is Town 0

```

np.random.seed(0)
prob = sum(transition_matrix[0][i] / 31 for i in range(31))
frequency = np.mean([random_drive(num_stops=1) == 0
                     for _ in range(50000)])
print(f"Probability of making our first stop in Town 0: {prob:.3f}")
print(f"Frequency with which our first stop is Town 0: {frequency:.3f}")

Probability of making our first stop in Town 0: 0.026
Frequency with which our first stop is Town 0: 0.026
  
```

Our computed probability is consistent with the observed frequency: we will make the first stop of our journey in Town 0 approximately 2.6% of the time. It's worth noting that the probability can be computed more concisely as a vector dot-product operation—we just need to run `transition_matrix[0] @ v`, where `v` is a 31-element vector whose elements all equal `1 / 31`. Let's execute this computation shortcut.

Listing 19.16 Computing a travel probability using a vector dot product

```

v = np.ones(31) / 31
assert transition_matrix[0] @ v == prob
  
```

Executing `transition_matrix[i] @ v` returns the likelihood of making our first stop in Town `i`. We can compute this likelihood for every town by running `[transition_matrix[i] @ v for i in range(31)]`. Of course, this operation is equivalent to the matrix product between `transition_matrix` and `v`, so `transition_matrix @ v` returns all first-stop probabilities. Listing 19.17 computes this `stop_1_probabilities` array

and prints the probability of making our first stop in Town 12. That probability should approximate the frequency computed through random simulations.

Listing 19.17 Computing all first stop probabilities

```

np.random.seed(0)
stop_1_probabilities = transition_matrix @ v
prob = stop_1_probabilities[12]
frequency = np.mean([random_drive(num_stops=1) == 12
                     for _ in range(50000)])

print('First stop probabilities:')
print(np.round(stop_1_probabilities, 3))
print(f"\nProbability of making our first stop in Town 12: {prob:.3f}")
print(f"Frequency with which our first stop is Town 12: {frequency:.3f}")

First stop probabilities:
[0.026 0.033 0.045 0.046 0.033 0.019 0.025 0.038 0.033 0.031 0.019 0.041
 0.052 0.03 0.036 0.019 0.031 0.039 0.023 0.031 0.027 0.019 0.018 0.044
 0.038 0.046 0.015 0.045 0.04 0.035 0.023]

Probability of making our first stop in Town 12: 0.052
Frequency with which our first stop is Town 12: 0.052

```

We've established that `transition_matrix @ v` returns a vector of first-stop probabilities. Now we need to prove that iteratively repeating this operation will eventually yield a vector of tenth-stop probabilities. However, first let's answer a simpler question: what is the probability of making our second stop in Town i ? Based on our previous discussions, we know the following:

- The probability of making our first stop in Town j equals `stop_1_probabilities[j]`.
- If the probability of our current location is p , then the probability of travel from the current location j to new location i equals $p * transition_matrix[i][j]$.
- Hence, the probability of first stopping in Town j and then travelling to Town i is $p * transition_matrix[i][j]$, where $p = stop_1_probabilities[j]$.
- We can compute this travel probability for every possible Town j .
- The sum of these probabilities equals the likelihood of making our first stop at a random town and then traveling directly to Town i . The sum equals `sum(p * transition_matrix[i][j] for j, p in enumerate(stop_1_probabilities))`.
- We can compute this likelihood more concisely as a vector dot-product operation. That operation equals `transition_matrix[i] @ stop_1_probabilities`.

The probability of making our second stop in Town i equals `transition_matrix[i] @ stop_1_probabilities`. We can compute this likelihood for every town using a matrix-vector product. Thus, `transition_matrix @ stop_1_probabilities` returns all second-stop probabilities. However, `stop_1_probabilities` is equal to `transition_matrix @ v`, so the second-stop probabilities are also equal to `transition_matrix @ transition_matrix @ v`.

Let's confirm our calculations by obtaining the second-stop probabilities. Then we print the probability of making our second stop in Town 12, which should approximate the frequency computed through random simulations.

Listing 19.18 Computing all second-stop probabilities

```
np.random.seed(0)
stop_2_probabilities = transition_matrix @ transition_matrix @ v
prob = stop_2_probabilities[12]
frequency = np.mean([random_drive(num_stops=2) == 12
                     for _ in range(50000)])

print('Second stop probabilities:')
print(np.round(stop_2_probabilities, 3))
print(f"\nProbability of making our second stop in Town 12: {prob:.3f}")
print(f"Frequency with which our second stop is Town 12: {frequency:.3f}")

Second stop probabilities:
[0.027 0.033 0.038 0.043 0.033 0.023 0.028 0.039 0.039 0.026 0.021 0.032
 0.048 0.034 0.039 0.023 0.032 0.041 0.023 0.029 0.025 0.024 0.023 0.04
 0.029 0.043 0.021 0.036 0.036 0.042 0.031]

Probability of making our second stop in Town 12: 0.048
Frequency with which our second stop is Town 12: 0.048
```

We were able to derive our second-stop probabilities directly from our first-stop probabilities. In a similar vein, we can derive the third-stop probabilities. If we repeat our derivation, we can easily show that `stop_3_probabilities` equals `transition_matrix @ stop_2_probabilities`. Of course, this vector also equals $M @ M @ M @ v$, where M is the transition matrix.

We can repeat this process to compute the fourth-stop probabilities, then the fifth-stop probabilities, and eventually the N th-stop probabilities. To compute the N th-stop probabilities, we simply need to execute $M @ v$ across N iterations. Let's define a function that computes all N th-stop probabilities directly from transition matrix M .

NOTE We're dealing with a random process composed of N distinct steps, in which N th-step probabilities can be computed directly from the $N - 1$ step. Such processes are called *Markov chains*, after the mathematician Andrey Markov.

Listing 19.19 Computing the N th-stop probabilities

```
def compute_stop_likelihoods(M, num_stops):
    v = np.ones(M.shape[0]) / M.shape[0]
    for _ in range(num_stops):
        v = M @ v

    return v

stop_10_probabilities = compute_stop_likelihoods(transition_matrix, 10)
prob = stop_10_probabilities[12]
```

```

print('Tenth stop probabilities:')
print(np.round(stop_10_probabilities, 3))
print(f"\nProbability of making our tenth stop in Town 12: {prob:.3f}")

Tenth stop probabilities:
[0.029 0.035 0.041 0.047 0.035 0.023 0.029 0.041 0.034 0.021 0.014 0.028
 0.051 0.038 0.044 0.025 0.037 0.045 0.02 0.026 0.02 0.02 0.019 0.039
 0.026 0.047 0.02 0.04 0.04 0.04 0.027]

Probability of making our tenth stop in Town 12: 0.051

```

As we've discussed, our iterative matrix multiplications form the basis for PageRank centrality. In the next subsection, we compare our outputs to the NetworkX PageRank implementation. This comparison will give us deeper insights into the PageRank algorithm.

19.2.2 Computing PageRank centrality using NetworkX

A function to compute PageRank centrality is included in NetworkX. Calling `nx.pagerank(G)` returns a dictionary mapping between the node IDs and their centrality values. Let's print the PageRank centrality of Town 12. Will it equal 0.051?

Listing 19.20 Computing PageRank centrality using NetworkX

```

centrality = nx.pagerank(G)[12]
print(f"The PageRank centrality of Town 12 is {centrality:.3f}.")

The PageRank centrality of Town 12 is 0.048.

```

The printed PageRank value is 0.048, which is slightly lower than expected. The difference is due to a slight tweak that ensures PageRank works on all possible networks. As a reminder: PageRank was initially intended to model random clicks through a web link graph. A web link graph has directed edges, which means certain web pages might not have any outbound links. Thus, an internet user might get stuck on a dead-end page if they rely on outbound links to traverse the web (figure 19.6). To counter this, the PageRank designers assumed that a user would eventually get tired of clicking web links and would reboot their journey by going to a totally random web page—in other words, they'd teleport to one of the `len(G.nodes)` nodes in the internet graph. The PageRank designers programmed teleportation to occur in 15% of transversal instances. Teleportation ensures that a user will never get stranded on a node with no outbound links.

In our road network example, teleportation is analogous to calling a helicopter service. Imagine that in 15% of our town visits, we get bored with the local area. We then call for a helicopter, which swoops in and takes us to a totally random town. Once in the air, our probability of flying to any town equals $1 / 31$. After we land, we rent a car and continue our journey using the existing network of roads. Hence, 15% of the time, we fly from Town i to Town j with a probability of $1 / 31$. In the remaining

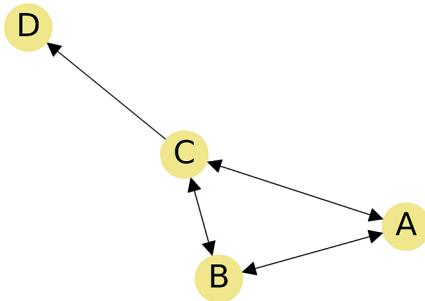


Figure 19.6 A directed graph containing four nodes. We can freely travel between interconnected nodes A, B, and C, but node D has no outbound edges. Sooner or later, a random traversal will take us from C to D. We will then be stuck forever at node D. Teleportation prevents this from happening. In 15% of our traversals, we'll teleport to a randomly chosen node. Even if we travel to node D, we can still teleport back to nodes A, B, and C.

85% of instances, we drive from Town i to Town j with a probability of `transition_matrix[j][i]`. Consequently, the actual travel probability equals the weighted mean of `transition_matrix[j][i]` and $1 / 31$. The respective weights are 0.85 and 0.15. As discussed in section 5, we can compute the weighted average using the `np.average` function. We can also compute that mean directly by running `0.85 * transition_matrix[j][i] + 0.15 / 31`.

Taking the weighted mean across all elements of the transition matrix will produce an entirely new transition matrix. Let's input that new matrix into our `compute_stop_likelihoods` function and print Town 12's new travel probability. We expect that probability to drop from 0.051 to 0.048.

Listing 19.21 Incorporating randomized teleportation into our model

```

new_matrix = 0.85 * transition_matrix + 0.15 / 31           ◀
stop_10_probabilities = compute_stop_likelihoods(new_matrix, 10)

prob = stop_10_probabilities[12]
print(f"The probability of winding up in Town 12 is {prob:.3f}.")

The probability of winding up in Town 12 is 0.048.

```

Multiples `transition_matrix` by 0.85 and then adds 0.15 / 31 to every element. See section 13 for a more in-depth discussion of arithmetic operations on 2D NumPy arrays.

Our new output is consistent with the NetworkX result. Will that output remain consistent if we increase the number of stops from 10 to 1,000? Let's find out. We'll input 1,000 stops into `compute_stop_likelihoods` and check whether Town 12's PageRank is still equal to 0.048.

Listing 19.22 Computing the probability after 1,000 stops

```

prob = compute_stop_likelihoods(new_matrix, 1000)[12]
print(f"The probability of winding up in Town 12 is {prob:.3f}.")

The probability of winding up in Town 12 is 0.048.

```

The centrality is still 0.048. Ten iterations were sufficient for convergence to a stable value. Why is this the case? Well, our PageRank computation is nothing more than the repeated multiplication of a matrix and a vector. The elements of the multiplied vector are all values between 0 and 1. Perhaps this sounds familiar: our PageRank computation is nearly identical to the power iteration algorithm that we presented in section 14. Power iteration repeatedly takes the product of a matrix and a vector; eventually, the product converges to an eigenvector of the matrix. As a reminder, the eigenvector v of a matrix M is a special vector where $\text{norm}(v) == \text{norm}(M @ v)$. Usually, 10 iterations are sufficient to achieve convergence. Hence, our PageRank values converge because we're running power iteration! This proves that our centrality vector is an eigenvector of the transition matrix. Thus, PageRank centrality is inexplicably linked to the beautiful mathematics behind dimensional reduction.

Given any graph G , we compute its PageRank centralities using the following series of steps:

- 1 Obtain the graph's adjacency matrix M .
- 2 Convert the adjacency matrix into the transition matrix by running $M = M / M.\text{sum(axis=0)}$.
- 3 Update M to allow for random teleportation. This is done by taking the weighted mean of M and $1 / n$, where n equals the number of nodes in the graph. The weights are usually set to 0.85 and 0.15, so the weighted mean equals $0.85 * M + 0.15 / n$.
- 4 Return the largest (and only) eigenvector of M . We can compute the eigenvector by running $v = M @ v$ across approximately 10 iterations. Initially, vector v is set to $\text{np.ones}(n) / n$.

Markov matrices tie graph theory together with probability theory and matrix theory. They can also be used to cluster network data using a procedure called *Markov clustering*. In the next subsection, we utilize Markov matrices to cluster communities in graphs.

Common NetworkX centrality computations

- `G.in_degree(i)`—Returns the in-degree of node i in a directed graph
- `G.degree(i)`—Returns the degree of node i in an undirected graph
- `nx.pagerank(G)`—Returns a dictionary mapping between node IDs and their PageRank centralities

19.3 Community detection using Markov clustering

Graph G represents a network of towns, some of which fall into localized counties. Currently, we know the county IDs, but what if we didn't? How would we identify the counties? Let's ponder this question by visualizing G without any sort of color mapping (figure 19.7).

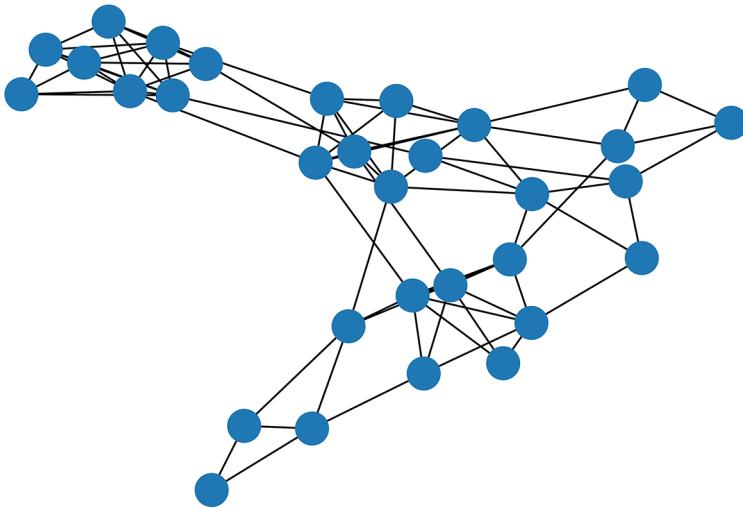


Figure 19.7 A network of roads between different towns. The towns have not been colored based on their counties. But we can still spot certain counties in our network: they appear as spatially clustered clumps.

Listing 19.23 Plotting G without county-based coloring

```
np.random.seed(1)
nx.draw(G)
plt.show()
```

Our plotted graph has neither colors nor labels. Still, we can spot potential counties: they look like tightly connected node clusters in the network. In graph theory, such clusters are formally referred to as *communities*. Graphs with clearly visible communities contain a *community structure*. Many types of graphs contain a community structure, including graphs of towns and graphs of social media friends.

NOTE Some common graphs do not contain a community structure. For instance, the internet lacks tightly clustered communities of web pages.

The process of uncovering graph communities is called *community detection* or *graph clustering*. Multiple graph-clustering algorithms exist, some of which depend on simulations of traffic flow.

How can we use traffic to uncover clusters of counties in our network? Well, we know that towns in the same country are more likely to share a road than towns in different counties, so if we drive to a neighboring town, we are likely to remain in the same county. In community-structured graphs, this logic holds even if we drive two towns over. Suppose, for instance, that we drive from Town i to Town j and then to Town k . Based on our network structure, Towns i and k are more likely to be in the same county. We will confirm this statement shortly; however, first we need to compute

the probability of a transition from Town i to Town k after two stops. This probability is called the *stochastic flow*, or *flow* for short. Flow is closely related to the transition probability; but unlike the transition probability, flow covers towns that aren't directly connected. We need to calculate the flow between each pair of towns and store that output in a *flow matrix*. Later, we show that the average flow is higher in towns that share the same community.

NOTE Generally, in network theory, flow is a very loosely defined concept. But in Markov clustering, that definition is constrained to the probability of eventual travel between nodes.

How do we calculate a matrix of flow values? One strategy is to simulate a two-stop journey between random towns. The simulated frequencies can then be converted into probabilities. However, it's far easier to compute these probabilities directly. With a bit of math, we can show that the flow matrix is equal to `transition_matrix @ transition_matrix`.

NOTE We can prove this statement as follows. Previously, we showed that the second-stop probabilities equal `transition_matrix @ transition_matrix @ v`. Furthermore, `transition_matrix @ transition_matrix` produces a new matrix, M . So, the second-stop probabilities equal $M @ v$. Essentially, M serves the same purpose as the `transition_matrix`, but it tracks two stops and not one; so M fits our definition of the flow matrix.

Basically, the random simulation approximates the product of the transition matrix with itself. Let's quickly verify before proceeding.

Listing 19.24 Comparing computed flow to random simulations

```
np.random.seed(0)
flow_matrix = transition_matrix @ transition_matrix

simulated_flow_matrix = np.zeros((31, 31))
num_simulations = 10000
for town_i in range(31):
    for _ in range(num_simulations):
        town_j = np.random.choice(G[town_i])
        town_k = np.random.choice(G[town_j])
        simulated_flow_matrix[town_k][town_i] += 1
    simulated_flow_matrix /= num_simulations
assert np.allclose(flow_matrix, simulated_flow_matrix, atol=1e-2)
```

Tracks the frequency with which we travel from Town i to Town k after two stops

Ensures that our simulated frequencies closely resemble the directly computed flow

Our `flow_matrix` is consistent with random simulations. Now, let's test our theory that flow is higher between towns in the same county. As a reminder, each town in `G.nodes` has been assigned a county ID. We believe that the average flow between Towns i and j is higher if `G.nodes[i]['county_id']` equals `G.nodes[j]['county_id']`. We can confirm by separating all flows into two lists: `county_flows` and

`between_county_flows`. The two lists track intra-county flows and inter-county flows, respectively. We'll plot a histogram for each of the lists and compare their mean flow values (figure 19.8). If we are correct, then `np.mean(county_flows)` should be noticeably higher than the mean flow of the second list. We'll also check whether any inter-county flows are explicitly less than `np.min(county_flows)`.

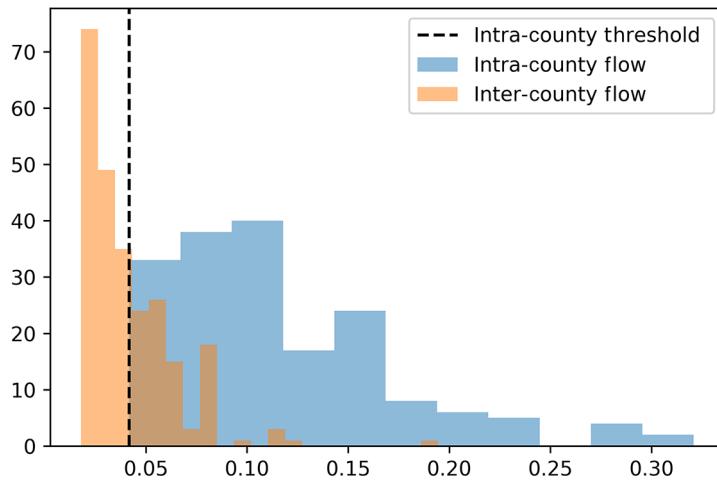


Figure 19.8 Two histograms representing all nonzero inter-county flows and intra-county flows. A separation between flow types is clearly visible. The inter-county flows skew strongly to the left: a threshold of approximately 0.042 is sufficient to separate 132 inter-county flows from the intra-county flow distribution.

Note that for a fair comparison, we should consider only the nonzero flows. So, we must skip over `flow_matrix[j][i]` if it has a zero value. A zero value implies that it's impossible to travel from i to j in just two stops (the probability of this occurring is zero). At least three stops are required, which indicates that the towns are far away from each other. This practically guarantees that they are in different counties. Hence, the inclusion of zero flows would unfairly skew our distribution of inter-county values toward zero. Let's challenge ourselves to examine the flows between only those towns that are in close proximity.

Listing 19.25 Comparing intra- and inter-county flow distributions

```
def compare_flow_distributions():
    county_flows = []           Tracks nonzero intra-county flows
    between_county_flows = []   Tracks nonzero inter-county flows
    for i in range(31):
        county = G.nodes[i]['county_id']
        nonzero_indices = np.nonzero(flow_matrix[:, i])[0]
        <span style="float: right; margin-right: 20px;">We only iterate over nonzero rows in column i.
```

```

for j in nonzero_indices:
    flow = flow_matrix[j][i]

    if county == G.nodes[j]['county_id']:
        county_flows.append(flow)
    else:
        between_county_flows.append(flow)

mean_intra_flow = np.mean(county_flows)
mean_inter_flow = np.mean(between_county_flows)
print(f"Mean flow within a county: {mean_intra_flow:.3f}")
print(f"Mean flow between different counties: {mean_inter_flow:.3f}")

threshold = min(county_flows)
num_below = len([flow for flow in between_county_flows
                 if flow < threshold])
print(f"The minimum intra-county flow is approximately {threshold:.3f}")
print(f"{num_below} inter-county flows fall below that threshold.")

plt.hist(county_flows, bins='auto', alpha=0.5,
         label='Intra-County Flow')
plt.hist(between_county_flows, bins='auto', alpha=0.5,
         label='Inter-County Flow')
plt.axvline(threshold, linestyle='--', color='k',
            label='Intra-County Threshold')
plt.legend()
plt.show()

compare_flow_distributions()

Mean flow within a county: 0.116
Mean flow between different counties: 0.042
The minimum intra-county flow is approximately 0.042
132 inter-county flows fall below that threshold.

```

Tracks all inter-county flows that are below the minimum intra-county flows

A histogram plot of intra-county flows

Checks if two towns are in the same county

A histogram plot of inter-county flows

The mean flow between counties is three times lower than the mean flow between towns in different counties. This difference is clearly visible in the plotted distribution: flows below a threshold of approximately 0.04 are guaranteed to represent inter-county values. Thus, we can isolate inter-county towns using an explicit threshold cutoff. Of course, we're only able to observe this threshold due to our advance knowledge of county identities. In a real-world scenario, the actual county IDs would not be known, so the separation cutoff would be impossible to explicitly determine. We'd be forced to assume that the cutoff is a low value, like 0.01. Suppose we made that assumption with our data. How many nonzero inter-county flows are less than 0.01? Let's find out.

Listing 19.26 Decreasing the separation threshold

```

num_below = np.count_nonzero((flow_matrix > 0.0) & (flow_matrix < 0.01))
print(f"{num_below} inter-county flows fall below a threshold of 0.01")

0 inter-county flows fall below a threshold of 0.01

```

None of the flow values fall below the stringent threshold of 0.01. What should we do? One option is to manipulate the flow distribution to exaggerate the difference between large and small values. Ideally, we'll force small values to fall below 0.01 while ensuring that larger flows do not drop in value. This manipulation can be carried out with a simple process called *inflation*. Inflation is intended to influence the values of a vector while keeping its mean constant. Values below the mean drop, while the remaining values increase. We'll demonstrate inflation with a simple example. Suppose we're inflating some vector v , which is equal to $[0.7, 0.3]$. The mean of v is 0.5. We want to increase $v[0]$ while decreasing $v[1]$. A partial solution is to square each element of v by running $v ** 2$. Doing so decreases $v[1]$ from 0.3 to 0.09. Unfortunately, it also decreases $v[0]$ from 0.7 to 0.49, so $v[0]$ drops below the original vector mean. We can alleviate the drop by dividing the squared vector by its sum to produce an inflated vector $v2$, whose sum is 1. It follows that $v2.mean()$ equals $v.mean()$. Furthermore, $v2[0]$ is greater than $v[0]$, and $v2[1]$ is less than $v[1]$. Let's confirm.

Listing 19.27 Exaggerating value differences through vector inflation

```
v = np.array([0.7, 0.3])
v2 = v ** 2
v2 /= v2.sum()
assert v.mean() == round(v2.mean(), 10)
assert v2[0] > v[0]
assert v2[1] < v[1]
```

Like vector v , the columns of our flow matrix are vectors whose elements sum to 1. We can inflate each column by squaring its elements and then dividing by the subsequent column sum. Let's define an *inflate* function for this purpose. Then we'll inflate the flow matrix and rerun `compare_flow_distributions()` to check whether our inter-county threshold has decreased (figure 19.9).

Listing 19.28 Exaggerating flow differences through vector inflation

```
def inflate(matrix):
    matrix = matrix ** 2
    return matrix / matrix.sum(axis=0)

flow_matrix = inflate(flow_matrix)
compare_flow_distributions()

Mean flow within a county: 0.146
Mean flow between different counties: 0.020
The minimum intra-county flow is approximately 0.012
118 inter-county flows fall below that threshold.
```

After inflation, our threshold has decreased from 0.042 to 0.012, but it still remains above 0.01. How do we further exaggerate the difference between inter-county and intra-county edges? The answer is surprisingly simple, although its reasoning is not immediately obvious: all we need to do is take the product of `flow_matrix` with itself

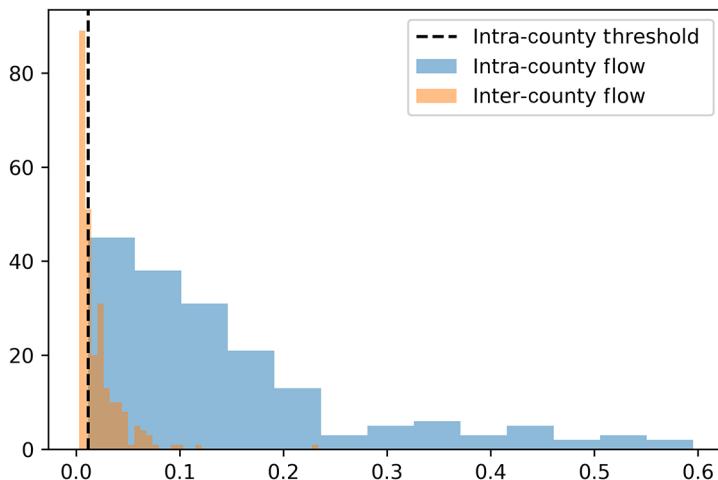


Figure 19.9 Two histograms representing all nonzero inter-county flows and intra-county flows after inflation. The separation between flows has become more visible: inflation has decreased the separation threshold from 0.042 to 0.012.

and then inflate the results. In other words, setting the flow matrix to equal `inflate (flow_matrix @ flow_matrix)` will cause the threshold to drastically decrease. Let's verify this claim before discussing the intuitive reasons behind the threshold drop (figure 19.10).

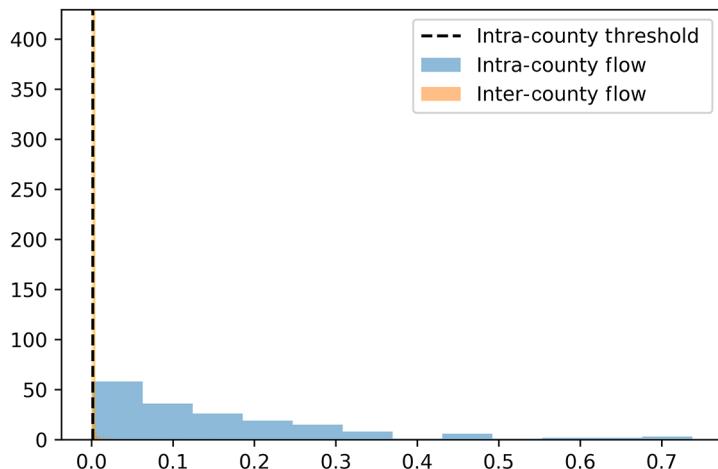


Figure 19.10 Two histograms representing all nonzero inter-county flows and intra-county flows after inflating `flow_matrix @ flow_matrix`. Most of the inter-county flows now fall below the very small separation threshold of 0.001.

Listing 19.29 Inflating the product of flow_matrix with itself

```
flow_matrix = inflate(flow_matrix @ flow_matrix) ←
compare_flow_distributions()

Mean flow within a county: 0.159
Mean flow between different counties: 0.004
The minimum intra-county flow is approximately 0.001
541 inter-county flows fall below that threshold.
```

Before this step,
flow_matrix was equal to
inflate(transition_matrix
@ transition_matrix). We
are essentially repeating a
matrix product that is then
coupled with inflation.

The threshold decreased to 0.001. More than 500 inter-county roads fall below that threshold. Why was our strategy successful? We can answer with a straightforward analogy. Suppose that we can build new roads between the towns, but all built roads require some annual maintenance. A poorly maintained road will develop cracks and fissures. Drivers will be more reluctant to go down a damaged road, so periodic repairs are very important. However, in our analogy, there isn't enough money to build new roads while repairing all existing roads in G . A local transportation bureau is given the difficult task of deciding

- Which new roads are built
- Which existing roads are maintained
- Which existing roads are ignored

The bureau makes the following assumption: pairs of towns with heavy flows require a better transportation infrastructure. Hence, a road between Towns i and j will be maintained only if $\text{flow_matrix}[i][j]$ or $\text{flow_matrix}[j][i]$ is high. If $\text{flow_matrix}[i][j]$ is high but there is no road between i and j , then resources will be allocated to connect Towns i and j directly.

NOTE A pair of non-neighboring towns will still have a high flow if multiple short detours exist between them. Building a road between that pair of towns makes sense since doing so will alleviate traffic along the detours.

Regrettably, not all existing roads will be maintained. Less-traveled inter-county roads will have a lower flow and will not receive attention from the bureau. Therefore, these roads will partially decay, and drivers will be less likely to travel between counties. Instead, the drivers will prefer to take the well-maintained intra-county roads, as well as newly built roads between the towns.

NOTE As a reminder, we're assuming that drivers travel randomly, without a particular destination in mind. Their aimless cruising is determined solely by the quality of road conditions.

Road construction, maintenance, and decay will inevitably alter our transition matrix. Transition probabilities between decaying low-flow roads will drop. Meanwhile, the transition probabilities between well-maintained high-flow roads will increase. We need to somehow model the alteration to our matrix while ensuring that matrix columns still sum to 1. How? With inflation, of course! Our inflation function exaggerates

the differences between values in the matrix while maintaining a column sum of 1. Thus, we'll model the consequences of the bureau's decision making by updating our transition matrix M to equal inflation (`flow_matrix`).

But the story isn't over. By changing the transition matrix, we also change the flow within the graph. Flow is equal to $M @ M$, where M is the flow matrix after inflation. Of course, the change will alter local resource allocation: after a new round of road construction and decay, the transition probabilities will come to equal inflate ($M @ M$). We can model the impact of the iterative road work as $M = \text{inflate}(M @ M)$. Note that in the current version of our code, M is set to `flow_matrix`. Thus, running `flow_matrix = inflate(flow_matrix @ flow_matrix)` will reinforce well-traveled roads even as less popular roads wither away (figure 19.11).

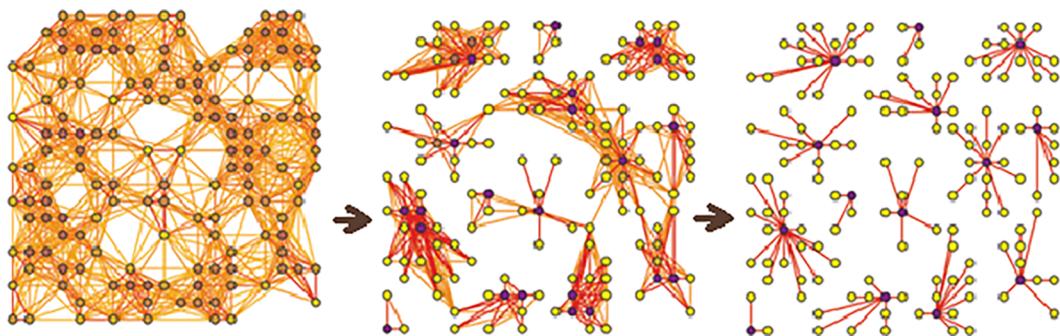


Figure 19.11 Modeling changes to a road graph using inflation. Roads between tightly connected towns are reinforced. Meanwhile, resources are diverted from less trafficked roads, which causes these roads to decay. Eventually, only the roads within the graph's communities remain.

This iterative feedback loop has unexpected ramifications: every year, the inter-county roads get worse and worse. As a result, more drivers stay within the boundaries of their county. More resources are allocated to the internal county roads, and the inter-county roads get less support and crumble further. It's a vicious cycle—eventually, the inter-county roads will crumble to dust, and it will no longer be possible to travel from county to county. Each separate county will become like an isolated island that is completely cut off from its neighbors. This isolation makes for terrible transportation policy, but it greatly simplifies the process of community detection. An isolated town cluster is easy to detect since it lacks boundaries with any other cluster. Consequently, our model of road build-up and decay serves as a basis for a network clustering algorithm: the *Markov Cluster Algorithm* (MCL), also referred to as *Markov clustering*.

MCL is executed by running `inflate(flow_matrix @ flow_matrix)` over many repeating iterations. With each iteration, the inter-county flows get smaller and smaller; eventually they drop to zero. Meanwhile, the intra-county flows maintain their positive values. This binary difference allows us to identify tightly connected county

clusters. Listing 19.30 attempts to execute MCL by running `flow_matrix = inflate(flow_matrix @ flow_matrix)` across 20 iterations.

Listing 19.30 Inflating the product of `flow_matrix` repeatedly with itself

```
for _ in range(20):
    flow_matrix = inflate(flow_matrix @ flow_matrix)
```

Based on our discussion, certain edges in graph `G` should now have a flow of zero. We expect these edges to connect diverging counties. Let's isolate the suspected inter-county edges. We iterate over every edge (i, j) by calling the `G.edges()` method. Then we track each edge (i, j) for which the flow is nonexistent and sort all the tracked edges in a `suspected_inter_county` edges list.

Listing 19.31 Selecting suspected inter-county edges

```
suspected_inter_county = [(i, j) for (i, j) in G.edges()
                           if not (flow_matrix[i][j] or flow_matrix[j][i])]
num_suspected = len(suspected_inter_county)
print(f"We suspect {num_suspected} edges of appearing between counties.")
```

We suspect 57 edges of appearing between counties.

57 edges lack any flow. We suspect that these edges connect towns between diverging counties. Deleting the suspected edges from our graph should sever all cross-county connections, so only clustered counties should remain if we visualize the graph after edge deletion. Let's verify by deleting the suspected edges from a copy of our graph (figure 19.12). We utilize the NetworkX `remove_edge_from` method to delete all edges in the `suspected_inter_county` list.

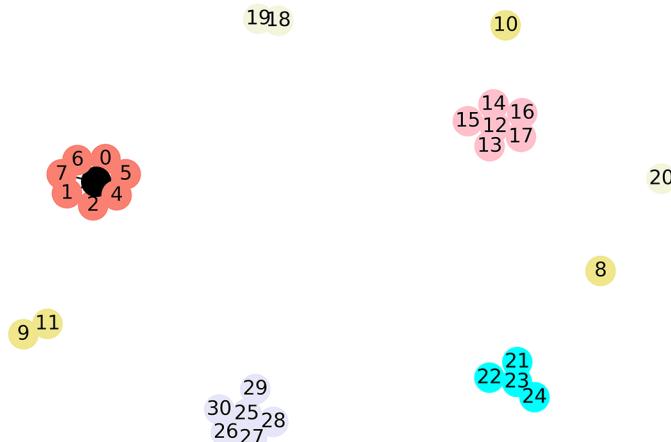


Figure 19.12 A network of towns after the deletion of all suspected inter-county edges. All counties have been fully isolated from each other. Four of the six counties have been fully preserved, but the remaining two counties are no longer fully connected.

Listing 19.32 Deleting suspected inter-county edges

```
np.random.seed(1)
G_copy = G.copy()    ← Running G.copy() returns a copied version of graph G. We can delete
G_copy.remove_edges_from(suspected_inter_county)
nx.draw(G_copy, with_labels=True, node_color=node_colors)
plt.show()
```

All inter-county edges have been eliminated. Unfortunately, a few key intra-county edges have also been deleted. Towns 8, 10, and 20 are no longer connected to any other towns. Our algorithm acted too aggressively. Why is this the case? The problem is due to a minor error in our model: it assumes that travelers can drive to neighboring towns, but it does not allow travelers to remain in their current location. This has unexpected consequences.

We'll illustrate with a simple two-node network. Imagine that a single road connects Towns A and B. In our current model, a driver in Town A has no choice except to travel to Town B. But the driver cannot stay: they must turn around and go back to Town A. A two-stop path does not exist between the towns, even though they are connected. Consequently, the flow between the towns will equal zero, and their connecting road will be eliminated. Of course, this situation is ridiculous—we should give the driver an option of remaining in Town B. How? One solution is to add an edge from Town B to itself. That edge is like a looping road, which takes you back to your current destination (figure 19.13). In other words, the edge is a self-loop. Adding self-loops to a graph will limit unexpected model behavior. Listing 19.33 illustrates the impact of self-loops in a simple two-node adjacency matrix.

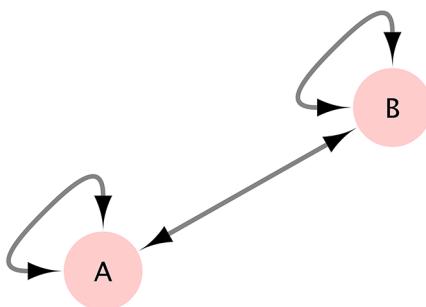


Figure 19.13 A graph indicating possible travel paths between Town A and Town B. Circular self-loops in each of the nodes allow a traveler to remain in place rather than journeying to a neighboring town. Without these loops, the traveler is forced to journey nonstop from A to B and back. If this happens, the flow between the towns will equal zero.

Listing 19.33 Improving flow by adding self-loops

```
def compute_flow(adjacency_matrix):
    transaction_matrix = adjacency_matrix / adjacency_matrix.sum(axis=0)
    return (transaction_matrix @ transaction_matrix) [1] [0]

M1 = np.array([[0, 1], [1, 0]])
M2 = np.array([[1, 1], [1, 1]])
```

```

flow1, flow2 = [compute_flow(M) for M in [M1, M2]]
print(f"The flow from A to B without self-loops is {flow1}")
print(f"The flow from A to B with self-loops is {flow2}")

The flow from A to B without self-loops is 0.0
The flow from A to B with self-loops is 0.5

```

Adding self-loops to graph G should limit inappropriate edge deletions. We can add the loops by running $G.add_edge(i, i)$ for every i in $G.nodes$. With this in mind, let's now define a `run_mcl` function that runs MCL on an inputted graph by executing the following steps:

- 1** Add a self-loop to each node in the graph.
- 2** Compute the graph's transition matrix by dividing the adjacency matrix by its column sums.
- 3** Calculate the flow matrix from `transition_matrix @ transition_matrix`.
- 4** Set `flow_matrix` to equal `inflate(flow_matrix @ flow_matrix)` over the course of 20 iterations.
- 5** Delete all edges in the graph that lack a flow.

After defining `run_mcl`, we execute the function on a copy of graph G . The plotted output should retain all relevant intra-county edges while also deleting all edges between the communities (figure 19.14).

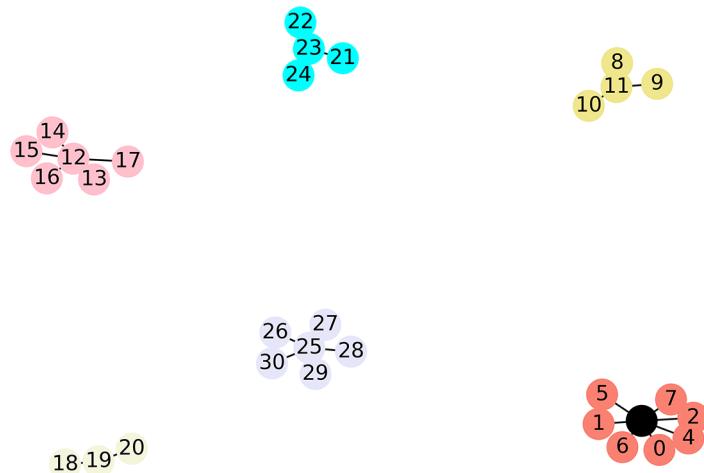


Figure 19.14 A network of towns after MCL was used to delete all inter-county edges. All counties have been fully isolated from each other. The internal connections within each county have also been fully preserved.

Listing 19.34 Defining an MCL function

```

def run_mcl(G):
    for i in G.nodes:
        G.add_edge(i, i)           ↪ Adds self-loops to each
                                    node in the graph

    adjacency_matrix = nx.to_numpy_array(G)
    transition_matrix = adjacency_matrix / adjacency_matrix.sum(axis=0)
    flow_matrix = inflate(transition_matrix @ transition_matrix)

    for _ in range(20):
        flow_matrix = inflate(flow_matrix @ flow_matrix)

    G.remove_edges_from([(i, j) for i, j in G.edges()
                        if not (flow_matrix[i][j] or flow_matrix[j][i])])

G_copy = G.copy()
run_mcl(G_copy)
nx.draw(G_copy, with_labels=True, node_color=node_colors)
plt.show()

```

Our graph has clustered perfectly into six secluded counties. The towns in each country are accessible to each other while remaining isolated from the outside world. In graph theory, such isolated clusters are referred to as *connected components*: two nodes are in the same connected component if a path exists between them. Otherwise, the nodes exist in different components (and thus in different communities). To compute the full component of a node, it is sufficient to run `nx.shortest_path_length` on that node. The shortest path length algorithm returns only those nodes that are accessible within a clustered community. The following code uses `nx.shortest_path_length` to compute all towns that remain accessible from Town 0 and confirm that all these towns share the same county ID.

Listing 19.35 Using path lengths to uncover a county cluster

```

component = nx.shortest_path_length(G_copy, source=0).keys()
county_id = G.nodes[0]['county_id']
for i in component:
    assert G.nodes[i]['county_id'] == county_id

print(f"The following towns are found in County {county_id}:")
print(sorted(component))

The following towns are found in County 0:
[0, 1, 2, 3, 4, 5, 6, 7]

```

With minor modifications to the shortest path length algorithm, we can extract a graph's connected components. For brevity's sake, we will not discuss these modifications, but you're encouraged to try to work them out for yourself. This modified component algorithm is incorporated into NetworkX: calling `nx.connected_components(G)`

returns an iterable over all connected components in G . Each connected component is stored as a set of node IDs. Let's utilize this function to output all the county clusters.

Listing 19.36 Extracting all the clustered connected components

```
for component in nx.connected_components(G_copy):
    county_id = G.nodes[list(component)[0]]['county_id']
    print(f"\nThe following towns are found in County {county_id}:")
    print(component)

The following towns are found in County 0:
{0, 1, 2, 3, 4, 5, 6, 7}

The following towns are found in County 1:
{8, 9, 10, 11}

The following towns are found in County 2:
{12, 13, 14, 15, 16, 17}

The following towns are found in County 3:
{18, 19, 20}

The following towns are found in County 4:
{24, 21, 22, 23}

The following towns are found in County 5:
{25, 26, 27, 28, 29, 30}
```

Common network matrix computations

- `adjacency_matrix = nx.to_numpy_array(G)`—Returns the graph's adjacency matrix.
- `degrees = adjacency_matrix.sum(axis=0)`—Computes the degree vector using the adjacency matrix.
- `transition_matrix = adjacency_matrix / degrees`—Computes the graph's transition matrix.
- `stop_1_probabilities = transition_matrix @ v`—Computes the probabilities of making a first stop at each node. Here, we assume that v is a vector of equally likely starting probabilities.
- `stop_2_probabilities = transition_matrix @ stop_1_probabilities`—Computes the probabilities of making a second stop at each node.
- `transition_matrix @ stop_n_probabilities`—Returns the probabilities of making an $N + 1$ stop at each node.
- `flow_matrix = transition_matrix @ transition_matrix`—Computes the probability matrix of transitioning between i and j in two stops.
- `(flow_matrix ** 2) / (flow_matrix ** 2).sum(axis=0)`—Inflates the flows in the flow matrix.

We've successfully uncovered the communities in our graph using very little code. Unfortunately, our MCL implementation will not scale to very large networks. Further optimizations are required for successful scaling; these optimizations have been integrated into the external Markov clustering library. Let's install the library and import two functions from the installed `markov_clustering` module: `get_clusters` and `run_mcl`.

NOTE Call `pip install markov_clustering` from the command line terminal to install the Markov clustering library.

Listing 19.37 Importing from the Markov clustering library

```
from markov_clustering import get_clusters, run_mcl
```

Given an adjacency matrix `M`, we can efficiently execute Markov clustering by running `get_clusters(run_mcl(M))`. The nested function call returns a `clusters` list. Each element in `clusters` equals a tuple of nodes that form a clustered community. Let's carry out this clustering on our original graph `G`. The outputted clusters should remain consistent with the connected components in `G_copy`.

Listing 19.38 Clustering with the Markov clustering library

```
adjacency_matrix = nx.to_numpy_array(G)
clusters = get_clusters(run_mcl(adjacency_matrix))

for cluster in clusters:
    county_id = G.nodes[cluster[0]]['county_id']
    print(f"\nThe following towns are found in County {county_id}:")
    print(cluster)

The following towns are found in County 0:
(0, 1, 2, 3, 4, 5, 6, 7)

The following towns are found in County 1:
(8, 9, 10, 11)

The following towns are found in County 2:
(12, 13, 14, 15, 16, 17)

The following towns are found in County 3:
(18, 19, 20)

The following towns are found in County 4:
(21, 22, 23, 24)

The following towns are found in County 5:
(25, 26, 27, 28, 29, 30)
```

With Markov clustering, we can detect communities in community-structured graphs. This will prove useful when we search for groups of friends in social networks.

19.4 Uncovering friend groups in social networks

We can represent many processes as networks, including relationships between people. In these *social networks*, nodes represent individual people. An edge exists between two people if they somehow socially interact. For instance, we can connect two people by an edge if they are friends.

Many different types of social networks are possible. Some networks are digital: for example, FriendHook's service is structured around online connections. However, social networks were studied for many decades before the rise of social media. One of the most-studied social networks originated in the 1970s: Zachery's *Karate Club*, based on the social structure of a university karate club, recorded by a scientist named Wayne Zachery. Over the course of three years, Zachery tracked friendships between the 34 members of the club. Edges were assigned to track friends who frequently met up outside the club. After three years, something unexpected happened: a karate instructor named Mr. Hi left to start a new club of his own, and half of the karate club went with him. Much to Zachery's surprise, most of the departing members could be identified solely from network structure.

We'll now repeat Zachery's experiment. First we'll load his famous karate network, which is available through NetworkX. Calling `nx.karate_club_graph()` returns that graph. The following code prints the graph nodes along with their attributes. As a reminder, we can output nodes with attributes by calling `G.nodes(data=True)`.

Listing 19.39 Loading the karate club graph

```
G_karate = nx.karate_club_graph()
print(G_karate.nodes(data=True))

[(0, {'club': 'Mr. Hi'}), (1, {'club': 'Mr. Hi'}), (2, {'club': 'Mr. Hi'}),
(3, {'club': 'Mr. Hi'}), (4, {'club': 'Mr. Hi'}), (5, {'club': 'Mr. Hi'}),
(6, {'club': 'Mr. Hi'}), (7, {'club': 'Mr. Hi'}), (8, {'club': 'Mr. Hi'}),
(9, {'club': 'Officer'}), (10, {'club': 'Mr. Hi'}), (11, {'club':
'Mr. Hi'}), (12, {'club': 'Mr. Hi'}), (13, {'club': 'Mr. Hi'}), (14,
{'club': 'Officer'}), (15, {'club': 'Officer'}), (16, {'club': 'Mr. Hi'}),
(17, {'club': 'Mr. Hi'}), (18, {'club': 'Officer'}), (19, {'club':
'Mr. Hi'}), (20, {'club': 'Officer'}), (21, {'club': 'Mr. Hi'}), (22,
{'club': 'Officer'}), (23, {'club': 'Officer'}), (24, {'club':
'Officer'}), (25, {'club': 'Officer'}), (26, {'club': 'Officer'}), (27,
{'club': 'Officer'}), (28, {'club': 'Officer'}), (29, {'club': 'Officer'}),
(30, {'club': 'Officer'}), (31, {'club': 'Officer'}), (32, {'club':
'Officer'}), (33, {'club': 'Officer'})]
```

Our nodes track 34 people. Each node has a `club` attribute set to `Mr. Hi` if the person joined Mr. Hi's new club or `Officer` otherwise. Let's visualize the network: we color each node based on `club` attribute type (figure 19.15).

Listing 19.40 Visualizing the karate club graph

```
np.random.seed(2)
club_to_color = {'Mr. Hi': 'k', 'Officer': 'b'}
```

```

node_colors = [club_to_color[G_karate.nodes[i]['club']]
               for i in G_karate]

nx.draw(G_karate, node_color=node_colors)
plt.show()

```

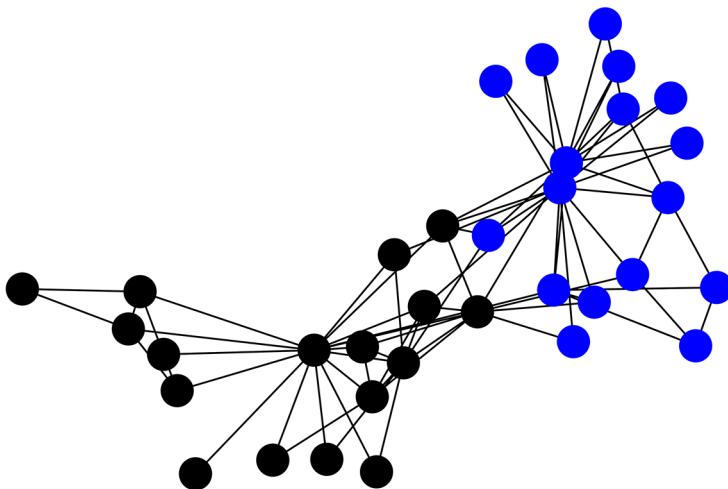


Figure 19.15 The visualized karate club graph. The node colors correspond to the splitting of the club. These colors overlap with the graph's community structure.

The karate club graph has a clear community structure. This is not surprising; many social networks contain detectable communities. In this case, the communities correspond to the splitting of the club: the black-colored cluster on the left side of the plot represents the club members who left to join with Mr. Hi, and the right-side cluster represents students who stayed behind. These clusters represent friend groups that formed over multiple years. When the split happened, most members simply went along with their preferred group of friends.

Can we extract these friend clusters automatically? We can try, using MCL. First we run the algorithm on the graph's adjacency matrix and print all the resulting clusters.

Listing 19.41 Clustering the karate club graph

```

adjacency_matrix = nx.to_numpy_array(G_karate)
clusters = get_clusters(run_mcl(adjacency_matrix))
for i, cluster in enumerate(clusters):
    print(f"Cluster {i}:\n{cluster}\n")

```

Cluster 0:
(0, 1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21)

```
Cluster 1:
(2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33)
```

Two clusters have been outputted, as expected. We now replot the graph while coloring each node based on cluster ID (figure 19.16).

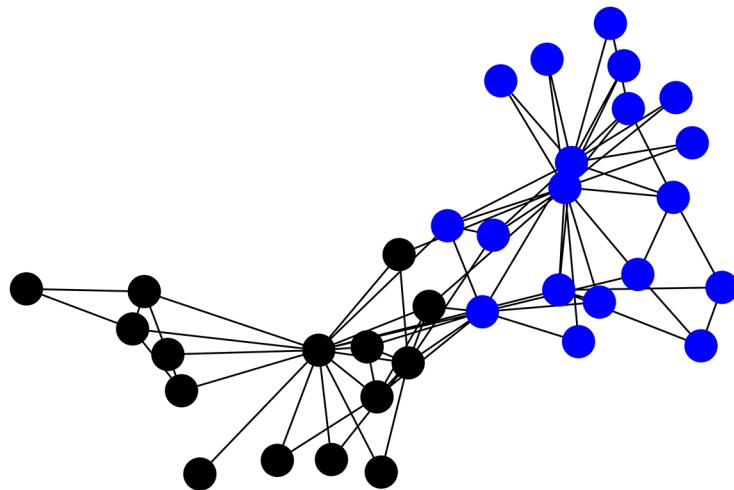


Figure 19.16 The visualized karate club graph. The node colors correspond to the community clusters. These colors overlap with the eventual splitting of the club.

Listing 19.42 Coloring the plotted graph based on cluster

```
np.random.seed(2)
cluster_0, cluster_1 = clusters
node_colors = ['k' if i in cluster_0 else 'b'
               for i in G_karate.nodes]

nx.draw(G_karate, node_color=node_colors)
plt.show()
```

Our clusters are nearly identical to the two splintered clubs. MCL has capably extracted the friend groups in the social network, so the algorithm should serve us well as we pursue our case study solution. In our case study, we're asked to analyze a digital social network. Extracting existing friend groups could prove invaluable to that analysis. Of course, in a large network, the number of groups will be greater than two—we can expect to encounter a dozen (or perhaps a few dozen) friend clusters. We'll also probably want to visualize these clusters in the graph. Manually assigning colors to a dozen clusters is a tedious task, so we'll want to generate the cluster colors automatically. In NetworkX, we can automate color assignment as follows:

- 1 Create a mapping between each node and its cluster ID by adding a `cluster_id` attribute to each node.
- 2 Set each element of `node_colors` to equal a cluster ID, rather than the color. This can be done by running `[G.nodes[n]['cluster_id'] for n in G.nodes]`, where `G` is the clustered social graph.
- 3 Pass `cmap=plt.cm.tab20` into `nx.draw`, along with the numeric `node_colors` list. The `cmap` parameter assigns a color mapping to each cluster ID. `plt.cm.tab20` represents the color palette used to generate that mapping; we've previously used color palette mappings to generate heatmap plots (see section 8 for details).

Let's execute these steps to color our clusters automatically (figure 19.17).

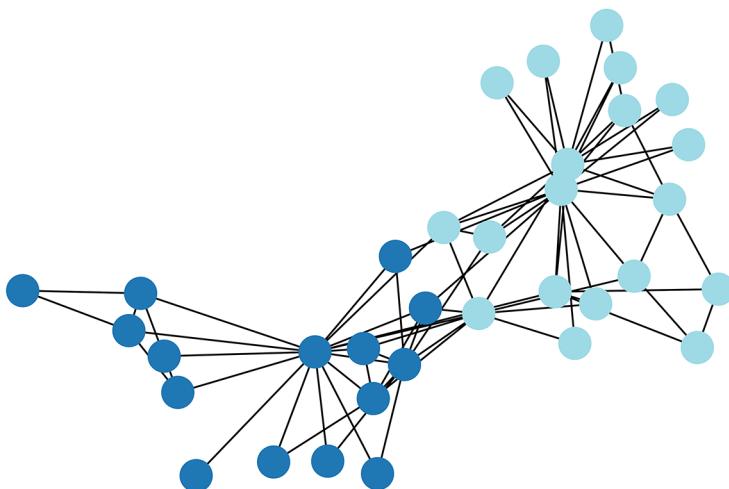


Figure 19.17 The visualized karate club graph. The node colors correspond to the community clusters. These colors were generated in an automated manner.

Listing 19.43 Coloring social graph clusters automatically

```
np.random.seed(2)
for cluster_id, node_indices in enumerate(clusters):
    for i in node_indices:
        G_karate.nodes[i]['cluster_id'] = cluster_id
```

↳ Assigns a cluster ID to every node

```
    node_colors = [G_karate.nodes[n]['cluster_id'] for n in G_karate.nodes]
    nx.draw(G_karate, node_color=node_colors, cmap=plt.cm.tab20)
    plt.show()
```

↳ Maps node colors to numeric cluster IDs

↳ Uses the `plt.cm.tab20` color palette to assign a color mapping to each cluster ID

We've completed our deep dive into graph theory. In the next section, we use our newfound knowledge to derive a simple, graph-based prediction algorithm.

Summary

- The edge count of a node in an undirected graph is simply called the node's *degree*. We can compute the degree of every node by summing over the columns of the graph's adjacency matrix.
- In graph theory, any measure of a node's importance is commonly called *node centrality*. Ranked importance based on a node's degree is called the *degree of centrality*.
- Sometimes, the degree of centrality is an inadequate measure of node importance. We can better derive centrality by simulating random traffic in the network. The traffic can be converted into a probability of randomly winding up at a particular node.
- Traffic probability can be computed directly from the graph's *transition matrix*. The transition matrix tracks the likelihood of randomly traveling from node i to node j . Repeatedly taking the product of the transition matrix and a probability vector produces a vector of final end-point likelihoods. Higher likelihoods correspond to more central nodes. This measure of centrality is known as *PageRank centrality*; mathematically, it is equal to the eigenvector of the transition matrix.
- Certain graphs, when visualized, show tightly connected clusters. These clusters of nodes are called *communities*. Graphs with clearly visible communities are said to contain a *community structure*. The process of uncovering communities in graphs is called *community detection*.
- We can detect communities using the *Markov Cluster Algorithm* (MCL). This algorithm requires us to compute a *stochastic flow*, which is a multistop transition probability. Taking the product of the transition matrix with itself produces a flow matrix. Lower flow values are more likely to correspond with inter-community edges. This difference between low and high flow values can be further amplified via *inflation*. Iteratively repeating matrix multiplication and inflation causes inter-community flows to drop to zero. Then, deleting zero-flow edges completely isolates the graph's communities. These isolated components can be identified with a variant of the shortest path length algorithm.
- In *social networks*, edges represent relationships between people. Social networks commonly contain a community structure, so we can use MCL to detect clusters of friends in these networks.



Network-driven supervised machine learning

This section covers

- Using classifiers in supervised machine learning
- Making simple predictions based on similarity
- Metrics for evaluating the quality of predictions
- Common supervised learning methods in scikit-learn

People can learn from real-world observations. In some respects, machines can do the same. Teaching computers to metaphorically understand the world through curated experience is referred to as *supervised machine learning*. In recent years, supervised machine learning has been all over the news: computers have been taught to predict stock prices, diagnose diseases, and even drive cars. These advancements have been rightly heralded as cutting-edge innovations. Yet, in some ways, the algorithms behind these innovations are not that novel. Variants of existing machine learning techniques have been around for many decades; but due to limited computing power, these techniques could not be adequately applied. Only now has our computing strength caught up. Hence, ideas planted many years ago are finally bearing the fruits of significant technological advancements.

In this section, we explore one of the oldest and simplest supervised learning techniques. This algorithm, called K-nearest neighbors, was first developed by the US Air Force in 1951. It is rooted in network theory and can be traced back to discoveries made by the medieval scholar Alhazen. Despite its age, the algorithm's usage has much in common with more modern techniques. Thus, the insights we'll gain will be transferable to the broader field of supervised machine learning.

20.1 The basics of supervised machine learning

Supervised machine learning is used to automate certain tasks that would otherwise be done by human beings. The machine observes a human carrying out a task and then learns to replicate the observed behavior. We'll illustrate this learning using the flower dataset introduced in section 14. As a reminder, the dataset represents three different species of the iris flower, which are displayed in figure 20.1. Visually, the three species look alike; botanists use subtle variations along the lengths and widths of the leaves to distinguish the species. That type of expert knowledge must be learned—no human or machine can distinguish between the species without training.



Iris Setosa

Iris Versicolor

Iris Virginica

Figure 20.1 Three species of iris flowers: *Setosa*, *Versicolor*, and *Virginica*. The species all look alike. Subtle differences in their leaves can be utilized to tell the species apart, but training is required to appropriately identify the different species.

Suppose a botany professor conducts an ecological analysis of a local pasture. Hundreds of iris plants are growing in the pasture, and the professor wishes to know the distribution of iris species among these plants. However, the professor is busy writing grants and doesn't have time to examine all the flowers personally. The professor thus hires an assistant to examine the flowers in the field. Unfortunately, the assistant is not a botanist and lacks the skills to tell the species apart. Instead, the assistant chooses to meticulously measure the lengths and widths of the leaves for every flower. Can these measurements be used to automatically identify all species? This question lies at the heart of supervised learning.

Essentially, we want to construct a model that maps inputted measurements to one of three species categories. In machine learning, these inputted measurements are

called *features*, and the outputted categories are called *classes*. The goal of supervised learning is to construct a model that can identify classes based on features. Such a model is called a *classifier*.

NOTE By definition, classes are discrete, categorical variables such as species of flower or type of car. Alternatively, there are models called *regressors* that predict numeric variables, such as the price of a house or the speed of a car.

There are many different types of machine learning classifiers. Whole books are dedicated to demarcating the various classifier categories. But despite their diversity, most classifiers require the same common steps for construction and implementation. To implement a classifier, we need to do the following:

- 1 Compute the features for each data point. In our botany example, all data points are flowers, so we need to measure the leaf lengths for each flower.
- 2 A domain expert must assign labels to a subset of the data points. Our botanist has no choice but to manually identify the species in a subset of the flowers. Without the professor's supervision, the classifier cannot be constructed properly. The term *supervised learning* is derived from this supervised labeling phase. Labeling the subset of flowers takes time, but that effort will pay off once the classifier can make automated predictions.
- 3 Show the classifier the combination of features and manually labeled classes. It then attempts to learn the association between the features and the classes. This learning phase varies from classifier to classifier.
- 4 Show the classifier a set of features that it has not previously encountered. It then attempts to predict the associated classes based on its exposure to the labeled data.

To construct a classifier, our botanist needs a set of features for a collection of identified flowers. Each flower is assigned the following four features (previously discussed in section 14):

- The length of a colorful petal
- The width of the colorful petal
- The length of a green leaf supporting the petal
- The width of the green leaf supporting the petal

We can store these features in a feature matrix. Each matrix column corresponds to one of the four features, and each matrix row corresponds to a labeled flower. The class labels are stored in a NumPy array. Such arrays are intended to hold numbers and not text; so, in machine learning, class labels are represented as integers that range from 0 to $N - 1$ (where N is the total number of classes). In our iris example, we are dealing with three species, so class labels range from 0 to 2.

As seen in section 14, we can load known iris features and class labels using scikit-learn's `load_iris` function. Let's do that now. Per existing scikit-learn convention, a

feature matrix is usually assigned to a variable called `x`, and the class-label array is assigned to a variable called `y`. Following this convention, the following code loads the iris `x` and `y` by passing `return X y=True` into `load_iris`.

Listing 20.1 Loading iris features and class labels

```
from sklearn.datasets import load_iris
X, y = load_iris(return_X_y=True)
num_classes = len(set(y))
print(f"We have {y.size} labeled examples across the following "
      f"{num_classes} classes:\n{set(y)}\n")
print(f"First four feature rows:\n{X[:4]}")
print(f"\nFirst four labels:\n{y[:4]}")

We have 150 labeled examples across the following 3 classes:
[0, 1, 2]

First four feature rows:
[[5.1 3.5 1.4 0.2]
 [4.9 3. 1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]]

First four labels:
[0 0 0 0]
```

All 150 flower measurements have been labeled as belonging to one of three flower species. Such labeling is hard work. Imagine that our botanist only has time to label one-fourth of the flowers. The professor then constructs a classifier to predict the classes of the remaining flowers. Let's simulate this scenario. We start by choosing the first one-fourth of data points in \mathbf{X} and y . This data slice is referred to as $\mathbf{X}_{\text{train}}$ and y_{train} since it is used for training purposes; such datasets are called *training sets*. After sampling our training set, we investigate the contents of y_{train} .

Listing 20.2 Creating a training set

```
sampling_size = int(y.size / 4)
X_train, y_train = X[:sampling_size], y[:sampling_size]
print(f"Training set labels:\n{y_train}")
```

Our training set contains just the labeled examples with Species 0; the remaining two flower species are not represented. To increase representation, we should sample at random from x and y . Random sampling can be achieved using scikit-learn's `train_test_split` function, which takes as input X and y and returns four randomly generated outputs. The first two outputs are x_{train} and y_{train} , corresponding to our training set. The next two outputs cover the features and classes outside of our

training set. These outputs can be utilized to test the classifier after training, so that data is commonly called the *test set*. We'll refer to the test features and classes as `x_test` and `y_test`, respectively. Later in this section, we use the test set to evaluate our trained model.

Listing 20.3 calls the `train_test_split` function and passes it an optional `train_size=0.25` parameter. The `train_size` parameter ensures that 25% of our total data winds up in the training set. Finally, we print `y_train` to ensure that all three labels are properly represented.

Listing 20.3 Creating a training set through random sampling

```
from sklearn.model_selection import train_test_split
import numpy as np
np.random.seed(0)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.25)
print(f"Training set labels:\n{y_train}")
```

```
Training set labels:
[0 2 1 2 1 0 2 0 2 0 0 2 0 2 1 1 1 2 2 1 1 0 1 2 2 0 1 1 1 1 0 0 0 2 1 2 0]
```

All three label classes are present in the training data. How can we utilize both `X_train` and `y_train` to predict the classes of the remaining flowers in the test set? One simple strategy involves geometric proximity. As we saw in section 14, the features in the iris dataset can be plotted in multidimensional space. This plotted data forms spatial clusters: elements in `X_test` are more likely to share their class with the `X_train` points found in the adjacent cluster.

Let's illustrate this intuition by plotting both `X_train` and `X_test` in 2D space (figure 20.2). We use principal component analysis to shrink our data to two dimensions, and then we plot the reduced features in our training set while coloring each plotted point based on its labeled class. We also plot the elements of our test set using a

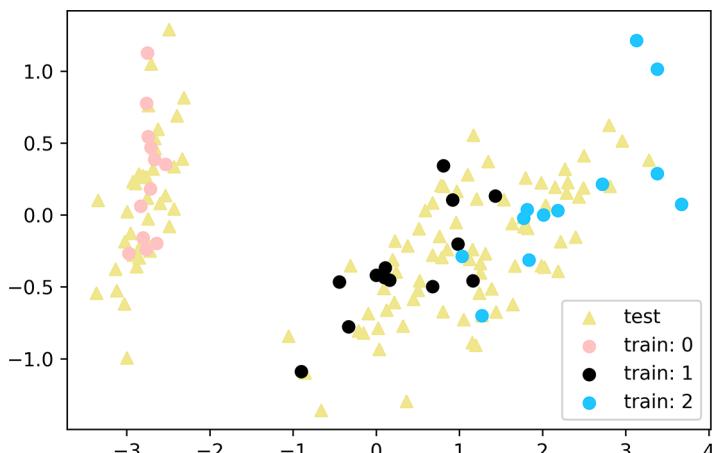


Figure 20.2 Flower data points plotted in 2D. Each labeled flower is colored based on its species class. Unlabeled flowers are also present in the plot. Visually, we can guess the identity of the unlabeled flowers based on their proximity to labeled points.

triangular marker to indicate the lack of a label. We then guess the identity of the unlabeled points based on their proximity to labeled data.

Listing 20.4 Plotting the training and test sets

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca_model = PCA()
transformed_data_2D = pca_model.fit_transform(X_train)

unlabeled_data = pca_model.transform(X_test)
plt.scatter(unlabeled_data[:,0], unlabeled_data[:,1],
            color='khaki', marker='^', label='test')

for label in range(3):
    data_subset = transformed_data_2D[y_train == label]
    plt.scatter(data_subset[:,0], data_subset[:,1],
                color=['r', 'k', 'b'][label], label=f'train: {label}')

plt.legend()
plt.show()
```

In the left-hand section of our plot, many unlabeled points cluster around Species 0. There is no ambiguity here: these unlabeled flowers clearly belong to the same species. Elsewhere in the plot, certain unlabeled flowers are proximate to both Species 1 and Species 2. For each such point, we need to quantify which labeled species are closer. Doing so requires us to track the Euclidean distance between each feature in `X_test` and each feature in `X_train`. Essentially, we need a distance matrix `M` where `M[i][j]` equals the Euclidean distance between `X_test[i]` and `X_test[j]`. Such a matrix can easily be generated using scikit-learn's `euclidean_distances` function. We simply need to execute `euclidean_distances(X_test, X_train)` to return the distance matrix.

Listing 20.5 Computing Euclidean distances between points

```
from sklearn.metrics.pairwise import euclidean_distances
distance_matrix = euclidean_distances(X_test, X_train)

f_train, f_test = X_test[0], X[0]
distance = distance_matrix[0][0]
print(f"Our first test set feature is {f_train}")
print(f"Our first training set feature is {f_test}")
print(f"The Euclidean distance between the features is {distance:.2f}")

Our first test set feature is [5.8 2.8 5.1 2.4]
Our first training set feature is [5.1 3.5 1.4 0.2]
The Euclidean distance between the features is 4.18
```

Given any unlabeled point in `X_test`, we can assign a class using the following strategy:

- 1 Sort all data points in the training set based on their distance to the unlabeled points.
- 2 Select the top K -nearest neighbors of the point. For now, we'll arbitrarily set K to equal 3.
- 3 Pick the most frequently occurring class across the K neighboring points.

Essentially, we're assuming that each unlabeled point shares a class that is common to its neighbors. This strategy forms the basis for the *K -nearest neighbors* (KNN) algorithm. Let's try this strategy on a randomly chosen point.

Listing 20.6 Labeling a point based on its nearest neighbors

```
from collections import Counter
np.random.seed(6)
random_index = np.random.randint(y_test.size)
labeled_distances = distance_matrix[random_index]
labeled_neighbors = np.argsort(labeled_distances)[:3]
labels = y_train[labeled_neighbors]

top_label, count = Counter(labels).most_common()[0]
print(f"The 3 nearest neighbors of Point {random_index} have the "
      f"following labels:\n{labels}")
print(f"\nThe most common class label is {top_label}. It occurs {count} "
      "times.")

The 3 nearest neighbors of Point 10 have the following labels:
[2 1 2]

The most common class label is 2. It occurs 2 times.
```

The most common class label among the neighbors of Point 10 is Label 2. How does this compare to the actual class of the flower?

Listing 20.7 Checking the true class of a predicted label

```
true_label = y_test[random_index]
print(f"The true class of Point {random_index} is {true_label}.")
```

The true class of Point 10 is 2.

KNN successfully identified the flower class of Point 10. All we needed to do was to check the labeled neighbors and count the most common label among them. Interestingly, this process can be reformulated as a graph theory problem. We can treat each point as a node and its label as a node attribute and then choose an unlabeled point and extend edges to its K closest labeled neighbors. Visualizing the neighbor graph allows us to identify the point.

NOTE This type of graph structure is called a *K -nearest neighbor graph* (k-NNG). Such graphs are used in a variety of fields, including transportation planning,

image compression, and robotics. Additionally, these graphs can be used to improve the DBSCAN clustering algorithm.

Let's demonstrate the network formulation of the problem by plotting the neighbor graph of Point 10 (figure 20.3). We utilize NetworkX for the purpose of this visualization.

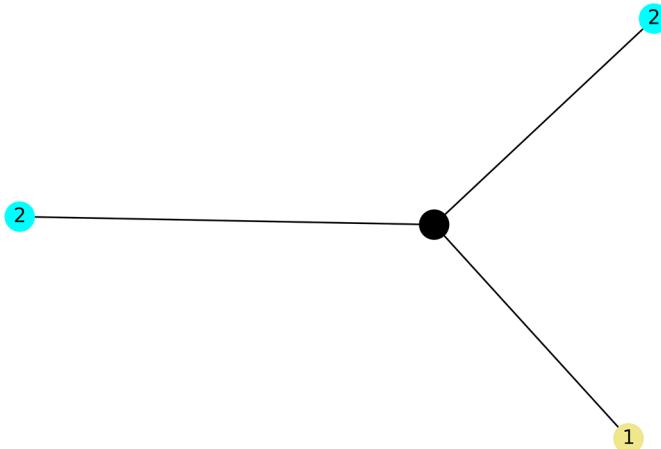


Figure 20.3 A NetworkX graph representing an unlabeled point and its three nearest labeled neighbors. Two of three neighbors are labeled Class 2. Thus, we can hypothesize that the unlabeled point also belongs to that majority class.

Listing 20.8 Visualizing nearest neighbors with NetworkX

```
import networkx as nx
np.random.seed(0)

def generate_neighbor_graph(unlabeled_index, labeled_neighbors):
    G = nx.Graph()
    nodes = [(i, {'label': y_train[i]}) for i in labeled_neighbors]
    nodes.append((unlabeled_index, {'label': 'U'}))
    G.add_nodes_from(nodes)
    G.add_edges_from([(i, unlabeled_index) for i in labeled_neighbors])
    labels = y_train[labeled_neighbors]
    label_colors = ['pink', 'khaki', 'cyan']
    colors = [label_colors[y_train[i]] for i in labeled_neighbors] + ['k']
    labels = {i: G.nodes[i]['label'] for i in G.nodes}
    nx.draw(G, node_color=colors, labels=labels, with_labels=True)
    plt.show()
    return G

G = generate_neighbor_graph(random_index, labeled_neighbors)
```

Plots and returns a NetworkX graph containing connections between an unlabeled data point and the labeled nearest neighbors of that point

Obtains the labels of the nearest neighbors

Colors the labeled neighbors based on their labels

KNN works when there are just three neighbors. What happens if we increase the neighbor count to four? Let's find out (figure 20.4).

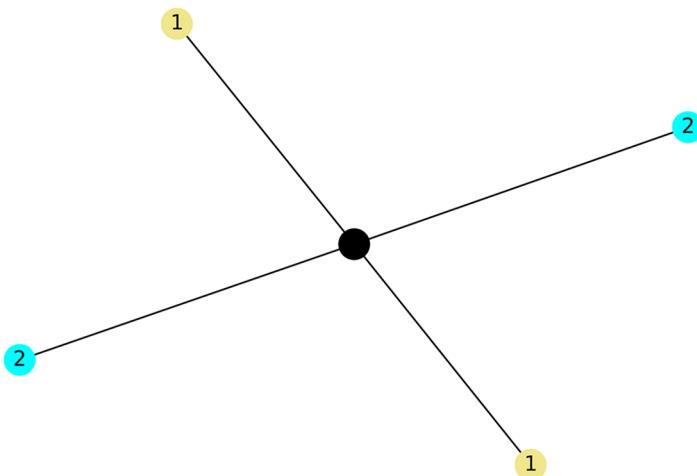


Figure 20.4 A NetworkX graph representing an unlabeled point and its four nearest labeled neighbors. Two of four neighbors are labeled Class 2, and the remaining two neighbors belong to Class 1. No majority class is present. Thus, we are unable to identify the unlabeled point

Listing 20.9 Increasing the number of nearest neighbors

```
np.random.seed(0)
labeled_neighbors = np.argsort(labeled_distances) [:4]
G = generate_neighbor_graph(random_index, labeled_neighbors)
```

There is a tie! No label dominates the majority. We can't make a decision. What should we do? One option is to break the tie at random. A better approach is to factor in the distances to the labeled points. Labeled points that are closer to Point 10 are more likely to share the correct class. Hence, we should give more weight to more proximate points, but how?

Well, in our initial KNN implementation, each labeled point received an equal vote, like in a fair democracy. Now we want to weigh each vote based on distance. One simple weighing scheme is to give each labeled point $1 / \text{distance}$ votes: a point that's one unit away will receive one vote, a point that's 0.5 units away will receive two votes, and a point that's two units away will receive just half a vote. This doesn't make for fair politics, but it could improve the output of our algorithm.

The following code assigns each labeled point a vote amount equal to its inverse distance from Point 10. Then we let the labeled points vote based on their class. We utilize the tallied votes to choose an elected class for our Point 10.

Listing 20.10 Weighing votes of neighbors based on distance

```
from collections import defaultdict
class_to_votes = defaultdict(int)
```

```

for node in G.neighbors(random_index):
    label = G.nodes[node]['label']
    distance = distance_matrix[random_index][node]
    num_votes = 1 / distance
    print(f"A data point with a label of {label} is {distance:.2f} units "
          f"away. It receives {num_votes:.2f} votes.")
    class_to_votes[label] += num_votes

print()
for class_label, votes in class_to_votes.items():
    print(f"We counted {votes:.2f} votes for class {class_label}.")

top_class = max(class_to_votes.items(), key=lambda x: x[1])[0]
print(f"Class {top_class} has received the plurality of the votes.")

A data point with a label of 2 is 0.54 units away. It receives 1.86 votes.
A data point with a label of 1 is 0.74 units away. It receives 1.35 votes.
A data point with a label of 2 is 0.77 units away. It receives 1.29 votes.
A data point with a label of 1 is 0.98 units away. It receives 1.02 votes.

We counted 3.15 votes for class 2.
We counted 2.36 votes for class 1.
Class 2 has received the plurality of the votes.

```

Once again, we've correctly chosen Class 2 as the true class of Point 10. The optional weighted voting can potentially improve our final prediction. Of course, this improvement is by no means guaranteed; occasionally, weighted voting can worsen the outputted results. Depending on the preset value of our K , weighted voting can either improve or worsen our predictions. We won't know for sure until we test prediction performance across a range of parameters. Such testing will require us to develop a robust metric for measuring performance accuracy.

20.2 Measuring predicted label accuracy

Thus far, we've examined class prediction for a single, randomly chosen point. Now we want to analyze predictions across all the points in `X_test`. We define a `predict` function for this purpose, which takes as input the index of an unlabeled point and a value of K , which we preset to 1.

NOTE We're purposefully inputting a low value of K to generate a multitude of errors worth improving. Later, we measure the error across multiple K -values to optimize performance.

The final parameter is a `weighted_voting` Boolean, which we set to `False`. That Boolean determines whether votes should be distributed according to distance.

Listing 20.11 Parameterizing KNN predictions

Predicts the label of a point using its row index in the distance matrix based on its K-nearest neighbors. The `weighted_voting` Boolean determines whether voting is weighted by neighbor distance.

```
def predict(index, K=1, weighted_voting=False):
    labeled_distances = distance_matrix[index]
```

```

labeled_neighbors = np.argsort(labeled_distances) [:K] ←
class_to_votes = defaultdict(int)
for neighbor in labeled_neighbors:
    label = y_train[neighbor]
    distance = distance_matrix[index] [neighbor]
    num_votes = 1 / max(distance, 1e-10) if weighted_voting else 1 ←
    class_to_votes[label] += num_votes
    ↳ return max(class_to_votes, key=lambda x: class_to_votes[x]) ←

assert predict(random_index, K=3) == 2
assert predict(random_index, K=4, weighted_voting=True) == 2

```

Returns the class label with the most votes

Obtains the K-nearest neighbors

Weighs votes equally if weighted_voting is False and by inverse distance otherwise. We take precautions when computing the inverse to avoid dividing by zero.

Let's execute predict across all unlabeled indices. Following a common naming convention, we store the predicted classes in an array named `y_pred`.

Listing 20.12 Predicting all unlabeled flower classes

```
y_pred = np.array([predict(i) for i in range(y_test.size)])
```

We want to compare the predicted classes with the actual classes in `y_test`. Let's start by printing out both the `y_pred` and the `y_test` arrays.

Listing 20.13 Comparing the predicted and actual classes

```

print(f"Predicted Classes:\n{y_pred}")
print(f"\nActual Classes:\n{y_test}")

Predicted Classes:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 2 0 2 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
2 1 1 2 0 2 0 0 1 2 2 1 2 1 2 1 1 1 1 1 2 1 0 2 1 1 1 2 2 0 0 2 1 0 0 2 1 0 0
1 0 2 1 0 1 2 1 0 2 2 2 2 0 0 2 2 0 2 0 2 2 0 0 2 0 0 0 1 2 2 0 0 0 1 1 0
0 1]

Actual Classes:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
1 1 1 2 0 2 0 0 1 2 2 2 2 1 2 1 1 2 2 2 2 1 2 1 0 2 1 1 1 1 2 0 0 2 1 0 0 2 1 0 0
1 0 2 1 0 1 2 1 0 2 2 2 2 0 0 2 2 0 2 0 2 2 0 0 2 0 0 0 1 2 2 0 0 0 1 1 0
0 1]

```

It's difficult to compare the two printed arrays. We can run an easier comparison if we aggregate the arrays into a single matrix `M` that contains three rows and three columns, corresponding to the number of classes. The rows track predicted classes, and the columns track the true class identities. Each element `M[i][j]` counts the co-occurrences between predicted Class i and actual Class j , as is illustrated in figure 20.5.

This type of matrix representation is known as a *confusion matrix* or an *error matrix*. As we shall see shortly, the confusion matrix can help quantify prediction errors. We

		Actual		
		Setosa	Versicolor	Virginica
Predicted	Setosa	14	1	1
	Versicolor	1	11	3
	Virginica	1	3	10

Figure 20.5 A hypothetical matrix representation of the predicted and actual classes. The rows correspond to predicted classes, and the columns correspond to the actual classes. Each element $M[i][j]$ counts the co-occurrences between predicted Class i and actual Class j . Hence, the matrix diagonal is counting all the accurate predictions.

now compute the confusion matrix using `y_pred` and `y_test` and visualize the matrix as a heatmap (figure 20.6).

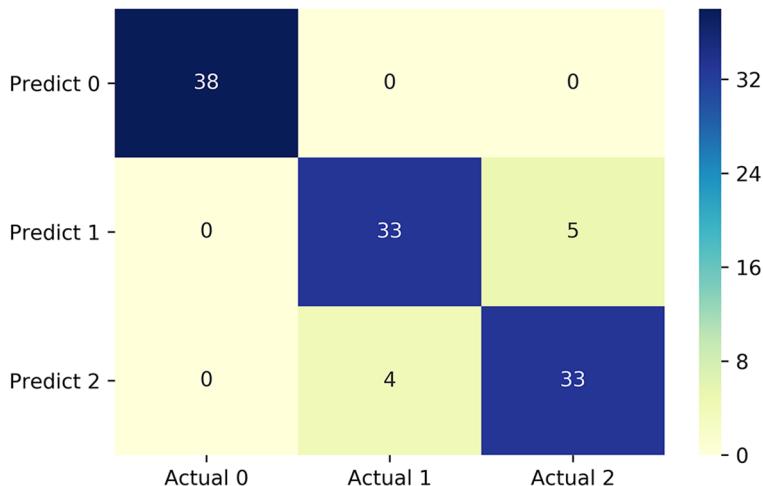


Figure 20.6 A confusion matrix comparing the predicted results to the actual results. The rows correspond to predicted classes, and the columns correspond to the actual classes. The matrix elements count all corresponding instances between the predicted and actual classes. The diagonal of the matrix counts all accurate predictions. Most of our counts lie along the matrix diagonal, indicating that our model is highly accurate.

Listing 20.14 Computing the confusion matrix

```
Checks for the total number of classes. This value
defines the number of matrix rows and columns.

import seaborn as sns
def compute_confusion_matrix(y_pred, y_test):
    num_classes = len(set(y_pred) | set(y_test)) ←
    Computes the
    confusion matrix
    between y_pred
    and y_test
```

```

confusion_matrix = np.zeros((num_classes, num_classes))
for prediction, actual in zip(y_pred, y_test):
    confusion_matrix[prediction][actual] += 1 ←

return confusion_matrix

M = compute_confusion_matrix(y_pred, y_test)
sns.heatmap(M, annot=True, cmap='YlGnBu',
            yticklabels=[f"Predict {i}" for i in range(3)],
            xticklabels = [f"Actual {i}" for i in range(3)])
plt.yticks(rotation=0)
plt.show()

```

Each predicted class Prediction corresponds to an actual class Actual. For every such pair, we add a 1 to the row Prediction and column Actual of our matrix. Note that if Prediction == Actual, then the added value appears on the diagonal of the matrix.

Most of the values in the matrix lie along its diagonal. Each diagonal element $M[i][i]$ tracks the number of accurately predicted instances of Class i . Such accurate predictions are commonly called *true positives*. Based on our displayed diagonal values, we know that our true positive count is very high. Let's print the total true positive count by summing across $M.diagonal()$.

Listing 20.15 Counting the number of accurate predictions

```

num_accurate_predictions = M.diagonal().sum()
print(f"Our results contain {int(num_accurate_predictions)} accurate "
      "predictions.")

```

Our results contain 104 accurate predictions.

The results include 104 accurate predictions: our accuracy is high. Of course, not all the predictions are accurate. Occasionally, our classifier gets confused and predicts the wrong class label: out of 113 total predictions, 9 predictions in the matrix lie outside the diagonal. The fraction of total accurate predictions is referred to as the *accuracy* score. Accuracy can be computed by dividing the diagonal sum across the total sum of matrix elements: in our case, dividing 104 by 113 produces a high accuracy value.

Listing 20.16 Measuring the accuracy score

```

accuracy = M.diagonal().sum() / M.sum()
assert accuracy == 104 / (104 + 9)
print(f"Our predictions are {100 * accuracy:.0f}% accurate.")

```

Our predictions are 92% accurate.

Our predictions are quite accurate, but they are not perfect. Errors are present in the output. These errors are not equally distributed: for instance, by examining the matrix, we can see that our Class 0 predictions are always right. The model never confuses Class 0 with any other class or vice versa; all 38 predictions for that class lie along

the diagonal. This is not the case for the other two classes: the model periodically confuses instances of Classes 1 and 2.

Let's try to quantify the observed confusion. Consider the elements in matrix Row 1, which tracks our predictions of Class 1. Summing across this row yields the total count of elements that we've predicted as belonging to Class 1.

Listing 20.17 Counting the predicted Class 1 elements

```
row1_sum = M[1].sum()
print(f"We've predicted that {int(row1_sum)} elements belong to Class 1.")

We've predicted that 38 elements belong to Class 1.
```

We predicted that Class 1 has 38 elements. How many of these predictions are correct? Well, 33 predictions lie along the `M[1][1]` diagonal. Thus, we've correctly identified 33 true positives of Class 1. Meanwhile, the remaining 5 predictions lie in Column 2. These 5 *false positives* represent Class 2 elements that we've misidentified as belonging to Class 1; they make our Class 1 predictions less reliable. Just because our model returns a Class 1 label does not mean the prediction is correct. In fact, our Class 1 label is correct in just 33 of 38 total instances. The ratio 33 / 38 produces a metric called *precision*: the true positive count divided by the sum of true positives and false positives. The precision of Class *i* is also equal to `M[i][i]` divided by the sum across Row *i*. A low precision indicates that a predicted class label is not very reliable. Let's output the precision of Class 1.

Listing 20.18 Computing the precision of Class 1

```
precision = M[1][1] / M[1].sum()
assert precision == 33 / 38
print(f"Precision of Class 1 is {precision:.2f}")

Precision of Class 1 is 0.87
```

The precision of Class 1 is 0.87, so a Class 1 label is reliable only 87% percent of the time. In the remaining 13% of instances, the prediction is a false positive. These false positives are a cause of error, but they're not the only one: additional errors can be detected across the confusion matrix columns. Consider, for example, Column 1, which tracks all elements in `y_test`, whose true label is equal to Class 1. Summing over Column 1 yields the total count of Class 1 elements.

Listing 20.19 Counting the total Class 1 elements

```
coll_sum = M[:, 1].sum()
assert coll_sum == y_test[y_test == 1].size
print(f"{int(coll_sum)} elements in our test set belong to Class 1.")

37 elements in our test set belong to Class 1.
```

37 elements in our test set belong to Class 1. 33 of these elements lie along the $M[1][1]$ diagonal: these true positive elements have been identified correctly. The remaining four elements lie in Row 2; these *false negatives* represent Class 1 elements that we've misidentified as belonging to Class 2. Hence, our identification of Class 1 elements is incomplete. Of the 37 possible class instances, only 33 have been identified correctly. The ratio $33 / 37$ produces a metric called *recall*: the true positive count divided by the sum of true positives and false negatives. The recall of Class i is also equal to $M[i][i]$ divided by the sum across Column i . A low recall indicates that our predictor commonly misses valid instances of a class. Let's output the recall of Class 1.

Listing 20.20 Computing the recall of Class 1

```
recall = M[1][1] / M[:,1].sum()
assert recall == 33 / 37
print(f"Recall of Class 1 is {recall:.2f}")
```

```
Recall of Class 1 is 0.89
```

The recall of Class 1 is 0.89, so we're able to detect 89% of valid Class 1 instances. The remaining 11% of instances are misidentified. The recall measures the fraction of identified Class 1 flowers in the pasture. By contrast, the precision measures the likelihood that a Class 1 prediction is correct.

It's worth noting that a maximum recall of 1.0 is trivial to achieve: we simply need to label each incoming data point as belonging to Class 1. We will detect all valid instances of Class 1, but this high recall will come at a cost. Precision will drop drastically, because all instances of Class 0 and Class 2 will be misidentified as belonging to Class 1. This low precision score equals $M[1][1] / M.sum()$.

Listing 20.21 Checking precision at a recall of 1.0

```
low_precision = M[1][1] / M.sum()
print(f"Precision at a trivially maximized recall is {low_precision:.2f}")
```

```
Precision at a trivially maximized recall is 0.29
```

In this same manner, a maximized precision is worthless if the recall is low. Imagine if the Class 1 precision equaled 1.0. We'd thus have 100% confidence that all Class 1 predictions are correct. However, if the corresponding recall is too low, most Class 1 instances will be misidentified as belonging to another class. Hence, high-level confidence is of little use if the classifier ignores most true instances.

A good predictive model should yield both high precision and high recall. We therefore should combine precision and recall into a single score. How can we combine the two distinct measures? One obvious solution is to take their average by running $(precision + recall) / 2$. Unfortunately, this solution has an unexpected drawback. Both precision and recall are fractions: $M[1][1] / M[1].sum()$ and $M[1][1] / M[:,1].sum()$, respectively. They share the same numerator but have different

denominators. This is problematic; fractions should only be added if their denominators are equal. Thus, the summation required to take the average is ill advised. What should we do? Well, we can take the inverse of both precision and recall. The inversions will swap each numerator with the denominators, so $1 / \text{precision}$ and $1 / \text{recall}$ will share an equal denominator of $M[1][1]$. These inverted fractions can then be summed. Let's see what happens when we take the average of the inverted metrics.

Listing 20.22 Taking the mean of the inverted metrics

```
inverse_average = (1 / precision + 1 / recall) / 2
print(f"The average of the inverted metrics is {inverse_average:.2f}")
```

The average of the inverted metrics is 1.14

The average of the inverses is greater than 1.0, but both precision and recall have a maximum ceiling of 1.0. Thus, their aggregation should fall below 1.0. We can guarantee this by inverting the computed average.

Listing 20.23 Taking the inverse of the inverted mean

```
result = 1 / inverse_average
print(f"The inverse of the average is {result:.2f}")
```

The inverse of the average is 0.88

Our final aggregated score is 0.88, which lies between the precision of 0.87 and the recall of 0.89. Hence, this aggregation is a perfect balance of precision and recall. This aggregated metric is called the *f1-measure*, *f1-score*, or, commonly, simply the *f-measure*. The f-measure can be computed more directly by running $2 * \text{precision} * \text{recall} / (\text{precision} + \text{recall})$.

NOTE This inversion of the arithmetic mean of inverse values is called the *harmonic mean*. The harmonic mean is intended to measure the central tendency of rates, such as velocities. Suppose, for instance, that an athlete runs three laps around a one-mile lake. The first lap takes 10 minutes, the next lap takes 16 minutes, and the final lap takes 20 minutes, so the athlete's velocities in miles per minute are $1 / 10$ (0.1), $1 / 16$ (0.0625), and $1 / 20$ (0.05). The arithmetic mean is $(0.1 + 0.0625 + 0.05) / 3$: approximately 0.071. However, this value is erroneous since diverging denominators are summed. Instead, we should compute the harmonic mean, $3 / (10 + 16 + 20)$, which is approximately 0.065 miles per minute. By definition, the f-measure equals the harmonic mean of precision and recall.

Listing 20.24 Computing the f-measure of Class 1

```
f_measure = 2 * precision * recall / (precision + recall)
print(f"The f-measure of Class 1 is {f_measure:.2f}")
```

The f-measure of Class 1 is 0.88

We should note that although in this instance, the f-measure is equal to the average of the precision and recall, this is not always the case. Consider a prediction that has one true positive, one false positive, and zero false negatives. What are the precision and the recall? How does their average compare to the f-measure? Let's check.

Listing 20.25 Comparing the f-measure to the average

```
tp, fp, fn = 1, 1, 0
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f_measure = 2 * precision * recall / (precision + recall)
average = (precision + recall) / 2
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"Average: {average}")
print(f"F-measure: {f_measure:.2f}")

Precision: 0.5
Recall: 1.0
Average: 0.75
F-measure: 0.67
```

In this theoretical example, the precision is low: 50%. Meanwhile, the recall is a perfect 100%. The average value between these two measures is a tolerable 75%. However, the f-measure is much lower than the average because the high recall cannot be justified by the exceptionally low precision value.

The f-measure provides us with a robust evaluation for an individual class. With this in mind, we'll now compute the f-measure for each class in our dataset.

Listing 20.26 Computing the f-measure for each class

```
def compute_f_measures(M):
    precisions = M.diagonal() / M.sum(axis=0)
    recalls = M.diagonal() / M.sum(axis=1)
    return 2 * precisions * recalls / (precisions + recalls)

f_measures = compute_f_measures(M)
for class_label, f_measure in enumerate(f_measures):
    print(f"The f-measure for Class {class_label} is {f_measure:.2f}")

The f-measure for Class 0 is 1.00
The f-measure for Class 1 is 0.88
The f-measure for Class 2 is 0.88
```

The f-measure for Class 0 is 1.0: that distinct class can be identified with perfect precision and perfect recall. Meanwhile, Class 1 and Class 2 share an f-measure of 0.88. The distinction between these classes is not perfect, and one is commonly mistaken for the other. These mistakes degrade the precision and recall of each class. Nonetheless, the final f-measure of 0.88 is wholly acceptable.

NOTE There's no official standard for an acceptable f-measure. Appropriate values can vary from problem to problem. But it's common to treat f-measures like exam grades: an f-measure of 0.9 to 1.0 is treated like an A; the model performs exceptionally well. An f-measure of 0.8 to 0.89 is treated like a B; there's room for improvement even though the model is acceptable. An f-measure of 0.7 to 0.79 is treated like a C; the model performs adequately but is not very impressive. An f-measure of 0.6 to 0.69 is treated like a D; unacceptable but still better than random. F-measure values below 0.6 are usually treated as totally unreliable.

We computed three f-measures across three different classes. These f-measures can be combined into a single score by taking their mean. Listing 20.27 outputs that unified f-measure score.

NOTE Our three f-measures are fractions with potentially different denominators. As we've discussed, it's best to combine fractions only when the denominators are equal. Unfortunately, unlike with precision and recall, there's no existing method for achieving denominator equality among f-measure outputs. Hence, we have no choice but to compute their average if we wish to obtain a unified score.

Listing 20.27 Computing a unified f-measure for all classes

```
avg_f = f_measures.mean()
print(f"Our unified f-measure equals {avg_f:.2f}")

Our unified f-measure equals 0.92
```

The f-measure of 0.92 is identical to our accuracy. This is not surprising since both f-measure and accuracy are intended to measure model performance. However, we must emphasize that f-measure and accuracy are not guaranteed to be the same. The difference between the metrics is especially noticeable when the classes are *imbalanced*. In an imbalanced dataset, there are far more instances of some Class A than of some Class B. Let's consider an example where we have 100 instances of Class A and just 1 instance of Class B. Furthermore, let's suppose our Class B predictions have a recall of 100% and a precision of 50%. We can represent this scenario with a two-by-two confusion matrix of the form `[[99, 0], [1, 1]]`. Let's compare the accuracy with the unified f-measure for this imbalanced result.

Listing 20.28 Comparing performance metrics across imbalanced data

```
M_imbalanced = np.array([[99, 0], [1, 1]])
accuracy_imb = M_imbalanced.diagonal().sum() / M_imbalanced.sum()
f_measure_imb = compute_f_measures(M_imbalanced).mean()
print(f"The accuracy for our imbalanced dataset is {accuracy_imb:.2f}")
print(f"The f-measure for our imbalanced dataset is {f_measure_imb:.2f}")

The accuracy for our imbalanced dataset is 0.99
The f-measure for our imbalanced dataset is 0.83
```

Our accuracy is nearly 100%. That accuracy is misleading—it doesn't truly represent the terrible precision with which the model predicts the second class. Meanwhile, the lower f-measure better reflects the balance between the different class predictions. Generally, the f-measure is considered a superior prediction metric due to its sensitivity to imbalance. Going forward, we rely on the f-measure to evaluate our classifiers.

20.2.1 Scikit-learn's prediction measurement functions

All the prediction metrics that we've discussed thus far are available in scikit-learn. They can be imported from the `sklearn.metrics` module. Each metric function takes as input `y_pred` and `y_test` and returns the metric criteria of our choice. For instance, we can compute the confusion matrix by importing and running `confusion_matrix`.

Listing 20.29 Computing the confusion matrix using scikit-learn

```
from sklearn.metrics import confusion_matrix
new_M = confusion_matrix(y_pred, y_test)
assert np.array_equal(new_M, M)
print(new_M)

[[38  0  0]
 [ 0 33  5]
 [ 0  4 33]]
```

In that same manner, we can compute the accuracy by importing and running `accuracy_score`.

Listing 20.30 Computing the accuracy using scikit-learn

```
from sklearn.metrics import accuracy_score
assert accuracy_score(y_pred, y_test) == accuracy
```

Also, the f-measure can be computed with the `f1_score` function. Using this function is a bit more nuanced since the f-measure can be returned as a vector or unified mean. Passing `average=None` into the function returns a vector of individual f-measures for each class.

Listing 20.31 Computing all f-measures using scikit-learn

```
from sklearn.metrics import f1_score
new_f_measures = f1_score(y_pred, y_test, average=None)
assert np.array_equal(new_f_measures, f_measures)
print(new_f_measures)

[1. 0.88 0.88]
```

Meanwhile, passing `average='macro'` returns a single average score.

NOTE Passing `average='micro'` computes the mean precision and mean recall across all classes. Then, these mean values are used to compute a single

f-measure score. Generally, this approach does not significantly impact the final unified f-measure result.

Listing 20.32 Computing a unified f-measure using scikit-learn

```
new_f_measure = f1_score(y_pred, y_test, average='macro')
assert new_f_measure == new_f_measures.mean()
assert new_f_measure == avg_f
```

Using the `f1_score` function, we can readily optimize our KNN classifier across its input parameters.

Common scikit-learn classifier evaluation functions

- `M = confusion_matrix(y_pred, y_test)`—Returns the confusion matrix `M` based on predicted classes in `y_pred` and the actual classes in `y_test`. Each matrix element `M[i][j]` counts the number of times that `y_pred[index] == i` while `y_test[index] == j` across every possible index.
- `accuracy_score(y_pred, y_test)`—Returns the accuracy score based on predicted classes in `y_pred` and the actual classes in `y_test`. Given the confusion matrix `M`, the accuracy score is equal to `M.diagonal().sum() / M.sum()`.
- `f_measure_vector = f1_score(y_pred, y_test, average=None)`—Returns a vector of f-measures for all possible `f_measure_vector.size` classes. The f-measure of Class `i` is equal to `f_measure_vector[i]`. This equals the harmonic mean of the precision and recall of Class `i`. Both precision and recall can be computed from the confusion matrix `M`. The precision of Class `i` equals `M[i][i] / M[i].sum()`, and the recall of Class `i` equals `M[i][i] / M[:,i].sum()`. The final f-measure value `f_measure_vector[i]` equals `2 * precision * recall / (precision + recall)`.
- `f1_score(y_pred, y_test, average='macro')`—Returns the average f-measure, equal to `f_measure_vector.mean()`.

20.3 Optimizing KNN performance

Currently, our `predict` function takes two input parameters: `K` and `weighted_voting`. These parameters must be set before training and influence the classifier's performance. Data scientists refer to such parameters as *hyperparameters*. All machine learning models have some hyperparameters that can be tweaked to enhance predictive power. Let's try to optimize our classifier's hyperparameters by iterating over all possible combinations of `K` and `weighted_voting`. Our `K` values range from 1 to `y_train.size`, and our Boolean `weighted_voting` parameter is set to `True` or `False`. For each hyperparameter combination, we train on `y_train` and compute `y_pred`. We then obtain the f-measure based on our predictions. All f-measures are plotted relative to the input `K`. We plot two separate curves: one for `weighted_voting = True` and another for `weighted_voting = False` (figure 20.7). Finally, we find the maximum f-measure in the plot and return its optimized parameters.

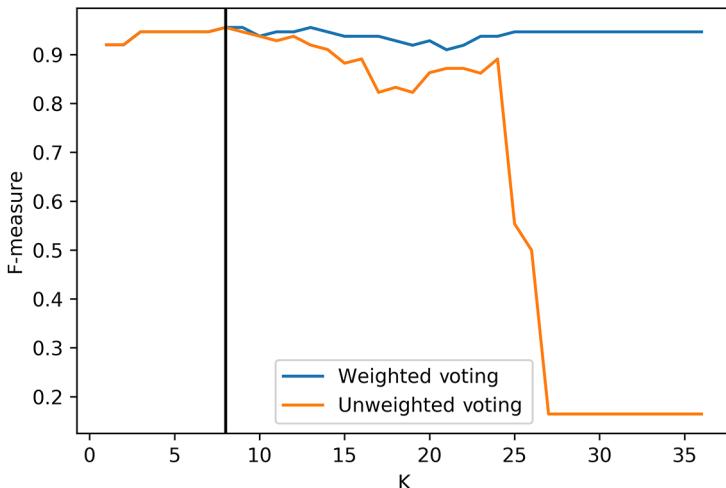


Figure 20.7 A plot of KNN weighted and unweighted voting performance measures across a range of input K values. F-measure is maximized when K is set to 8. There's no significant difference for weighted and unweighted voting for low values of K . However, unweighted performance starts to degrade when K is larger than 10.

Listing 20.33 Optimizing KNN hyperparameters

```

k_values = range(1, y_train.size)
weighted_voting_bools = [True, False]
f_scores = [[], []]

params_to_f = {} ← Tracks the mapping
for k in k_values: ← between each parameter
    for i, weighted_voting in enumerate(weighted_voting_bools): ← combination and the
        y_pred = np.array([predict(i, K=k, ← f-measure
                               weighted_voting=weighted_voting) ←
                           for i in range(y_test.size)]) ←
        f_measure = f1_score(y_pred, y_test, average='macro')
        f_scores[i].append(f_measure)
        params_to_f[(k, weighted_voting)] = f_measure ← Computes a KNN
                                                        prediction for
                                                        each parameter
                                                        combination

Computes the ←
f-measure for ←
each parameter ←
combination ←
    best_k, best_f = max(params_to_f.items(), ←
                          key=lambda x: x[1]) ←
    plt.plot(k_values, f_scores[0], label='Weighted Voting') ←
    plt.plot(k_values, f_scores[1], label='Unweighted Voting') ←
    plt.axvline(best_k, c='k') ←
    plt.xlabel('K') ←
    plt.ylabel('F-measure') ←
    plt.legend() ←
    plt.show() ←

    print(f"The maximum f-measure of {best_f:.2f} is achieved when K={best_k} "
          f"and weighted_voting={best_weighted}")

The maximum f-measure of 0.96 is achieved when K=8 and weighted_voting=True

```

Performance is maximized when K is set to 8 and weighted voting is activated. However, there's no significant difference between the weighted and unweighted voting output for that value of K . Interestingly, as K continues to increase, the unweighted f-measure drops rapidly. Meanwhile, the weighted f-measure continues to hover at above 90%. Thus, weighted KNN appears to be more stable than the unweighted variant.

We gained these insights by exhaustively iterating over all the possible input parameters. This exhaustive approach is called a *parameter sweep* or *grid search*. A grid search is a simple but effective way to optimize hyperparameters. Though it suffers from computational complexity when the parameter count is high, a grid search is very easy to parallelize. With enough computing power, a grid search can effectively optimize many common machine learning algorithms. Generally, a grid search is conducted like this:

- 1 Select our hyperparameters of interest.
- 2 Assign a range of values to each hyperparameter.
- 3 Split our input data into a training set and a validation set. The validation set is used to measure the prediction quality. This approach is called *cross-validation*. Note that it is possible to split the data further into multiple training and validation sets; that way, multiple prediction metrics can be averaged out into a single score.
- 4 Iterate over all possible hyperparameter combinations.
- 5 At each iteration, train a classifier on the training data using the specified hyperparameters.
- 6 Measure the classifier's performance using the validation set.
- 7 Once all iterations are completed, return the hyperparameter combination with the highest metric output.

Scikit-learn allows us to execute a grid search on all its built-in machine learning algorithms. Let's utilize scikit-learn to run a grid search on KNN.

20.4 Running a grid search using scikit-learn

Scikit-learn has built-in logic for running KNN classification. We utilize this logic by importing the `KNeighborsClassifier` class.

Listing 20.34 Importing scikit-learn's KNN class

```
from sklearn.neighbors import KNeighborsClassifier
```

Initializing the class creates a KNN classifier object. Per common convention, we store this object in a `clf` variable.

NOTE The KNN algorithm can be extended beyond mere classification: it can be modified to predict continuous values. Imagine that we wish to predict the sale price of a house. We can do this by averaging the known sales prices for similar houses in the neighborhood. In that same way, we construct a

KNN regressor that predicts a data point's continuous value by averaging known values of its neighbors. Scikit-learn includes a `KNeighborsRegressor` class that is designed for this specific purpose.

Listing 20.35 Initializing scikit-learn's KNN classifier

```
clf = KNeighborsClassifier()
```

The initialized `clf` object has preset specifications for K and weighted voting. The K value is stored in the `clf.n_neighbors` attribute, and the weighted voting specifications are stored in the `clf.weights` attribute. Let's print and examine both these attributes.

Listing 20.36 Printing the preset KNN parameters

```
K = clf.n_neighbors
weighted_voting = clf.weights
print(f"K is set to {K}.")
print(f"Weighted voting is set to '{weighted_voting}'.")

K is set to 5.
Weighted voting is set to 'uniform'.
```

Our K is set to 5, and weighted voting is set to `uniform`, indicating that all votes are weighted equally. Passing `weights='distance'` into the initialization function ensures that votes are weighted by distance. Additionally, passing `n_neighbors=4` sets K to 4. Let's reinitialize `clf` with these parameters.

Listing 20.37 Setting scikit-learn's KNN parameters

```
clf = KNeighborsClassifier(n_neighbors=4, weights='distance')
assert clf.n_neighbors == 4
assert clf.weights == 'distance'
```

Now we want to train our KNN model. Any scikit-learn `clf` classifier can be trained using the `fit` method. We simply need to execute `clf.fit(X, y)`, where `X` is a feature matrix and `y` is a class-label array. Let's train the classifier using the training set defined by `X_train` and `y_train`.

Listing 20.38 Training scikit-learn's KNN classifier

```
clf.fit(X_train, y_train)
```

After training, `clf` can predict the classes of any input `X_test` matrix (whose dimensions match `X_train`). Predictions are carried out with the `clf.predict` method. Running `clf.predict(X_test)` returns a `y_pred` prediction array. Subsequently, `y_pred` together with `y_test` can be used to calculate the f-measure.

Listing 20.39 Predicting classes with a trained KNN classifier

```
y_pred = clf.predict(X_test)
f_measure = f1_score(y_pred, y_test, average='macro')
print(f"The predicted classes are:\n{y_pred}")
print(f"\nThe f-measure equals {f_measure:.2f}.")

The predicted classes are:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0
2 1 1 2 0 2 0 0 1 2 2 1 2 1 2 1 1 1 1 1 2 1 0 2 1 1 1 2 0 0 2 1 0 0
1 0 2 1 0 1 2 1 0 2 2 2 0 0 2 2 0 2 0 2 2 0 0 2 0 0 1 2 2 0 0 0 1 1 0
0 1]
```

The f-measure equals 0.95.

clf also allows us to extract more nuanced prediction outputs. For instance, we can generate the fraction of the votes received by each class for an inputted sample in X_test. To obtain this voting distribution, we need to run clf.predict_proba(X_test). The predict_proba method returns a matrix whose columns correspond to vote ratios. Here we print the first four rows of this matrix, which correspond to X_test[:5].

Listing 20.40 Outputting vote ratios for each class

```
vote_ratios = clf.predict_proba(X_test)
print(vote_ratios[:4])

array([[0.          , 0.21419074, 0.78580926],
       [0.          , 1.          , 0.          ],
       [1.          , 0.          , 0.          ],
       [0.          , 0.          , 1.          ]])
```

As we can see, the data point at X_test[0] received 78.5% of votes for Class 2. The rest of the votes were given to Class 1. Meanwhile, X_test[4] received a full 100% of votes for Class 2. Even though both data points are assigned a class label of 2, the second point is assigned that label with a higher degree of confidence.

It's worth noting that all scikit-learn classifiers include their own version of predict_proba. The method returns an estimated probability distribution of data points belonging to some class. The column index with the highest probability is equal to the class label in y_pred.

Relevant scikit-learn classifier methods

- `clf = KNeighborsClassifier()`—Initializes a KNN classifier where $K = 5$ and voting is uniform across the five nearest neighbors.
- `clf = KNeighborsClassifier(n_neighbors=x)`—Initializes a KNN classifier where $K = x$ and voting is uniform across the x neighbors.

(continued)

- `clf = KNeighborsClassifier(n_neighbors=x, weights='distance')`—Initializes a KNN classifier where $K = x$ and voting is weighted by distance to each of the x neighbors.
- `clf.fit(X_train, y_train)`—Fits any classifier `clf` to predict classes `y` from features `X` based on training features `X_train` and training labeled classes `y_train`.
- `y = clf.predict(x)`—Predicts an array of classes associated with the feature matrix `x`. Each predicted class `y[i]` maps the matrix feature row `X[i]`.
- `M = clf.predict_proba(X)`—Returns a matrix `M` of probability distributions. Each row `M[i]` represents the probability distribution of data point i belonging to some class. The class prediction of that data point equals the distribution's maximum value. More concisely, `M[i].argmax() == clf.predict(X)[i]`.

Now, let's turn our attention to running a grid search across `KNeighborsClassifier`. First we need to specify a dictionary mapping between our hyperparameters and their value ranges. The dictionary keys equal our input parameters `n_neighbors` and `weights`. The dictionary values equal the respective iterables, `range(1, 40)`, and `['uniform', 'distance']`. Let's create this `hyperparams` dictionary.

Listing 20.41 Defining a hyperparameter dictionary

```
hyperparams = {'n_neighbors': range(1, 40),           ←
               'weights': ['uniform', 'distance']}
```

In our manual grid search, the neighbor count ranged from 1 to `y_train.size`, where `y_train.size` equaled 37. However, that parameter range can be set to any arbitrary value. Here, we set the range cutoff to 40, which is a nice round number.

Next, we need to import scikit-learn's `GridSearchCV` class, which we'll use to execute the grid search.

Listing 20.42 Importing scikit-learn's grid search class

```
from sklearn.model_selection import GridSearchCV
```

It's time to initialize the `GridSearchCV` class. We input three parameters into the initializing method. The first parameter is `KNeighborsClassifier()`: an initialized scikit-learn object whose hyperparameters we wish to optimize. Our second input is the `hyperparams` dictionary. Our final input is `scoring='f1_macro'`, which sets the evaluation metric to the averaged f-measure value.

The following code executes `GridSearchCV(KNeighborsClassifier(), hyperparams, scoring='f1_macro')`. The initialized object can perform classification, so we assign it to the variable `clf_grid`.

Listing 20.43 Initializing scikit-learn's grid search class

```
clf_grid = GridSearchCV(KNeighborsClassifier(), hyperparams,
                        scoring='f1_macro')
```

We're ready to run grid search on our fully labeled dataset X , y . Running `clf_grid.fit(X, y)` executes this parameter sweep. Scikit-learn's internal methods automatically split X and y during the validation process.

Listing 20.44 Running a grid search using scikit-learn

```
clf_grid.fit(X, y)
```

We've executed the grid search. The optimized hyperparameters are stored in the `clf_grid.best_params_` attribute, and the f-measure associated with these parameters is stored in `clf_grid.best_score_`. Let's output these results.

Listing 20.45 Checking the optimized grid search results

```
best_f = clf_grid.best_score_
best_params = clf_grid.best_params_
print(f"A maximum f-measure of {best_f:.2f} is achieved with the "
      f"following hyperparameters:\n{best_params}")

A maximum f-measure of 0.99 is achieved with the following hyperparameters:
{'n_neighbors': 10, 'weights': 'distance'}
```

Scikit-learn's grid search achieved an f-measure of 0.99. This value is higher than our custom grid search output of 0.96. Why is it higher? Well, scikit-learn has carried out a more sophisticated version of cross-validation. Rather than splitting the dataset into two parts, it split the data into five equal parts. Each individual data partition served as a training set, and the data outside each partition was used for testing. The five f-scores across the five training sets were computed and averaged. The final mean value of 0.99 represents a more accurate estimation of classifier performance.

NOTE Splitting the data into five parts for evaluation purposes is called *5-fold cross-validation*. Generally, we can split the data into K equal parts. In `GridSearchCV`, the splitting is controlled by the `cv` parameter. Passing `cv = 2` splits the data into two parts, and the final f-measure resembles our original value of 0.96.

Maximized performance is achieved when `n_neighbors` is set to 10 and weighted voting is activated. The actual KNN classifier containing these parameters is stored in the `clf_grid.best_estimator_` attribute.

NOTE Multiple hyperparameter combinations lead to an f-measure of 0.99. The chosen combination may vary across different machines. Thus, your parameter outputs may be slightly different even though the optimized f-measure will remain the same.

Listing 20.46 Accessing the optimized classifier

```
clf_best = clf_grid.best_estimator_
assert clf_best.n_neighbors == best_params['n_neighbors']
assert clf_best.weights == best_params['weights']
```

By using `clf_best`, we can carry out predictions on new data. Alternatively, we can carry out predictions directly with our optimized `clf_grid` object by running `clf_grid.predict`. Both objects return identical results.

Listing 20.47 Generating predictions with `clf_grid`

```
assert np.array_equal(clf_grid.predict(X), clf_best.predict(X))
```

Relevant scikit-learn grid search methods and attributes

- `clf_grid = GridSearchCV(ClassifierClass(), hyperparams, scoring = scoring_metric)`—Creates a grid search object intended to optimize classifier prediction across all possible hyperparameters based on a scoring metric specified by `scoring`. If `ClassifierClass()` is equal to `KNeighborsClassifier()`, then `clf_grid` serves to optimize KNN. If the `scoring_metric` is equal to `f1_macro`, the average f-measure is utilized for optimization.
- `clf_grid.fit(X, y)`—Executes a grid search to optimize classifier performance across all possible combinations of hyperparameter values.
- `clf_grid.best_score_`—Returns the optimal measure of classifier performance after a grid search has been executed.
- `clf_grid.best_params_`—Returns the combination of hyperparameters that leads to optimal performance based on the grid search.
- `clf_best = clf_grid.best_estimator_`—Returns a scikit-learn classifier object that shows optimal performance based on a grid search.
- `clf_grid.predict(X)`—A shortcut to execute `clf_grid.best_estimator_.predict(X)`.

20.5 Limitations of the KNN algorithm

KNN is the simplest of all supervised learning algorithms. That simplicity leads to certain flaws. Unlike other algorithms, KNN is not interpretable: we can predict the class and inputted data point, but we cannot comprehend why that data point belongs to that class. Suppose we train a KNN model that predicts whether a high school student belongs to 1 of 10 possible social cliques. Even if the model is accurate, we still can't understand why the student is classified as a jock and not as a member of the glee club. Later, we'll encounter other algorithms that can be used to better understand how data features relate to class identity.

Additionally, KNN only works well when the feature count is low. As the number of features increases, potentially redundant information begins to creep into the data.

Hence, the distance measures become less reliable, and the prediction quality suffers. Fortunately, feature redundancy can partially be alleviated by the dimension-reduction techniques introduced in section 14. But even with the proper application of these techniques, large feature sets can still lead to less accurate predictions.

Finally, the biggest problem with KNN is its speed. The algorithm can be very slow to run when the training set is large. Suppose we build a training set with a million labeled flowers. Naively, finding the nearest neighbors of an unlabeled flower would require us to scan its distance to each of the million flowers. This will take a lot of time. Of course, we can optimize for speed by organizing the training data more efficiently. The process is analogous to organizing words in a dictionary. Imagine we want to look up the word *data* in an unalphabetized dictionary. Since words are stored haphazardly, we need to scan each page. In the 6,000-page Oxford dictionary, this would take a very long time. Fortunately, all dictionaries are alphabetized, so we quickly look up the word by flipping open the dictionary at approximately its middle point. Here, at page 3,000, we encounter the letters *M* and *N*. Then we can flip the pages to the halfway point between page 3,000 and the inside cover; this takes us to page 1,500, which should contain words with the letter *D*. We're thus much closer to our goal. Repeating this process several more times will take us to the word *data*.

In a similar manner, we can quickly scan nearest neighbors if we first organize the training set by spatial distance. Scikit-learn employs a special data structure called a *K-D tree* to ensure that proximate training points are stored more closely to each other. This leads to faster scanning and quicker neighbor lookup. The details of K-D tree construction are beyond the scope of this book, but you're encouraged to read Manning's *Advanced Algorithms and Data Structures* by Marcello La Rocca to learn more about this very useful technique (2021, www.manning.com/books/algorithms-and-data-structures-in-action).

Despite the built-in lookup optimization, as we mentioned, KNN can still be slow to run when the training set is large. The reduction is especially cumbersome during hyperparameter optimization. We'll illustrate this slowdown by increasing the elements in our training set (*X*, *y*) 2,000-fold. Then we'll time the grid search for the expanded data.

WARNING The following code will take a long time to run.

Listing 20.48 Optimizing KNN on a large training set

```
import time
X_large = np.vstack([X for _ in range(2000)])
y_large = np.hstack([y for _ in range(2000)])
clf_grid = GridSearchCV(KNeighborsClassifier(), hyperparams,
                        scoring='f1_macro')
start_time = time.time()
clf_grid.fit(X_large, y_large)
running_time = (time.time() - start_time) / 60
print(f"The grid search took {running_time:.2f} minutes to run.")

The grid search took 16.23 minutes to run.
```

Our grid search took over 16 minutes to run! This is not an acceptable running time. We need an alternative solution. In the following section, we explore new classifiers whose prediction running time is not dependent on the training set size. We develop these classifiers from commonsense first principles and then we utilize their scikit-learn implementations.

Summary

- In *supervised machine learning*, our goal is to find a mapping between inputted measurements called *features* and outputted categories called *classes*. A model that identifies classes based on features is called a *classifier*.
- To construct a classifier, we first require a dataset with both features and labeled classes. This dataset is called a *training set*.
- One very simple classifier is *K-nearest neighbors* (KNN). KNN can classify an unlabeled point based on the plurality class among the *K*-nearest labeled points in the training set. Essentially, these neighbors vote to decide the unknown classes. Optionally, the voting can be weighted based on the distance of the neighbors to the unlabeled point.
- We can evaluate the performance of a classifier by computing a *confusion matrix*, M . Each diagonal element $M[i][i]$ tracks the number of accurately predicted instances of class i . Such accurate predictions are called the *true positive* instances of a class. The fraction of total elements along the diagonal of M is the *accuracy score*.
- Predicted Class A elements that actually belong to Class B are called the *false positives* of Class A. Dividing the true positive count by the sum of true positives and false positives produces a metric called *precision*. A low precision indicates that the predicted class label is not very reliable.
- Actual Class A elements that are predicted to belong to Class B are called the *false negatives* of Class A. Dividing the true positive count by the sum of true positives and false negatives produces a metric known as *recall*. A low recall indicates that our predictor commonly misses valid instances of a class.
- A good classifier should yield both high precision and high recall. We can combine precision and recall into a single metric called the *f-measure*. Given precision p and recall r , we can compute the f-measure by running $2 * p * r / (p + r)$. Multiple f-measures across multiple classes can be averaged into a single score.
- The f-measure can sometimes be superior to the accuracy, especially when data is imbalanced, so it's the preferable evaluation metric.
- To optimize KNN performance, we need to choose an optimal value for K . We also need to decide whether to utilize weighted voting. These two parameterized inputs are called *hyperparameters*. Such hyperparameters must be set before training. All machine learning models have hyperparameters that can be tweaked to enhance predictive power.

- The simplest hyperparameter optimization technique is called a *grid search*, which is conducted by iterating over every possible hyperparameter combination. Before the iterations, the original dataset is split into a training set and a validation set. This splitting is referred to as *cross-validation*. Then we iterate over the parameters. At each iteration, the classifier is trained and evaluated. Finally, we choose the hyperparameter values that lead to the highest metric output.

Training linear classifiers with logistic regression

This section covers

- Separating data classes with simple linear cuts
- What is logistic regression?
- Training linear classifiers using scikit-learn
- Interpreting the relationship between class prediction and trained classifier parameters

Data classification, much like clustering, can be treated as a geometry problem. Similarly, labeled classes cluster together in an abstract space. By measuring the distance between points, we can identify which data points belong to the same cluster or class. However, as we learned in the last section, computing that distance can be costly. Fortunately, it's possible to find related classes without measuring the distance between all points. This is something we have done before: in section 14, we examined the customers of a clothing store. Each customer was represented by two features: height and weight. Plotting these features revealed a cigar-shaped plot. We flipped the cigar on its side and sliced it vertically into three segments representing three classes of customers: small, medium, and large.

It's possible to separate distinct classes of data by carving out those classes as though with a knife. The carving can be carried out with simple linear cuts. Previously,

we limited ourselves to vertical downward cuts. In this section, we learn how to cut the data at an angle to maximize class separation. Through directed linear cuts, we can classify our data without relying on distance calculations. In the process, we learn how to train and interpret linear classifiers. Let's get started by revisiting the problem of separating customers by size.

21.1 Linearly separating customers by size

In section 14, we simulated customer heights (in inches) and weights (in pounds). Customers with larger inch/pound combinations fell into the Large customer class. We'll now rerun that simulation. Our heights and weights are stored in a feature matrix X , and the customer classes are stored in the class-label array of y . For the purpose of this exercise, we focus on the two classes Large and Not Large. We assume that customers in the Large class are taller than 72 inches and heavier than 160 lb. After we simulate this data, we make a scatter plot of X in which the plotted points are colored based on the class labels in y (figure 21.1). This visual representation will help us look for the spatial separation between the different customer types.

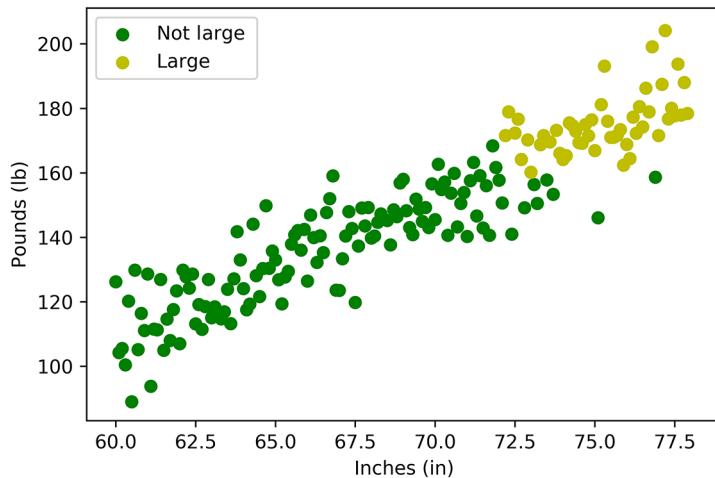


Figure 21.1 A plot of customer measurements: inches vs. lbs. Large and Not Large customers are colored differently based on their class.

Listing 21.1 Simulating categorized customer measurements

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(1)

def plot_customers(X, y, xlabel='Inches (in)', ylabel='Pounds (lb)'):
    colors = ['g', 'y']

    Plots customer measurements while coloring the
    customers based on class. Customer heights and weights
    are treated as two different features in the feature matrix
    X. Customer class type is stored with the label array y.
```

Customers fall into two classes, Large and Not Large.

```

    labels = ['Not Large', 'Large']
    for i, (color, label) in enumerate(zip(colors, labels)):
        plt.scatter(X[:,0][y == i], X[:,1][y == i], color=color, label=label)

    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    inches = np.arange(60, 78, 0.1)
    random_fluctuations = np.random.normal(scale=10, size=inches.size)
    pounds = 4 * inches - 130 + random_fluctuations
    X = np.array([inches, pounds]).T
    y = ((X[:,0] > 72) & (X[:,1] > 160)).astype(int)

plot_customers(X, y)
plt.legend()
plt.show()

```

Follows the linear formula from section 14 to model weight as a function of height

Customers are considered Large if their height is greater than 72 inches and their weight is greater than 160 lb.

Our plot resembles a cigar with different shades of colors at both ends. We can imagine a knife slicing through the cigar to separate the colors. The knife acts like a boundary that separates the two customer classes. We can represent this boundary using a line with a slope of -3.5 and a y -intercept of 415 . The formula for the line is $\text{lbs} = -3.5 \times \text{inches} + 415$. Let's add this linear boundary to the plot (figure 21.2).

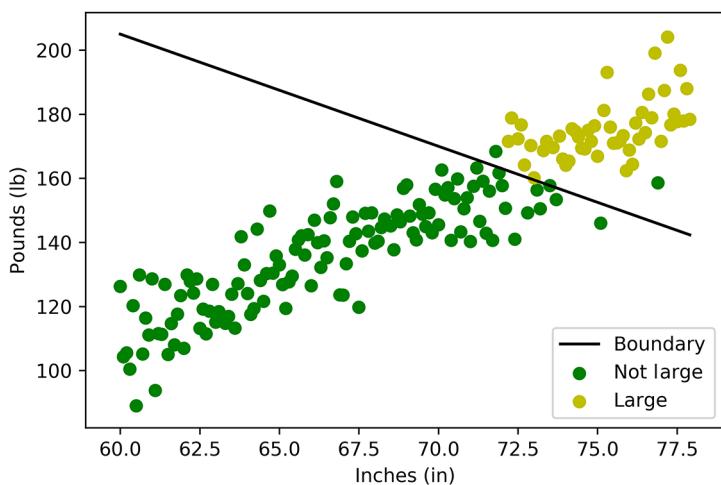


Figure 21.2 A plot of customer measurements: inches vs. lbs. A linear boundary separates the Large and Not Large customers.

NOTE We learn how to automatically compute the linear boundary later in this section.

Listing 21.2 Plotting a boundary to separate the two customer classes

```
def boundary(inches): return -3.5 * inches + 415
plt.plot(X[:,0], boundary(X[:,0]), color='k', label='Boundary')
plot_customers(X, y)
plt.legend()
plt.show()
```

The plotted line is called a *linear decision boundary* because it can be utilized to accurately choose a customer's class. Most of the customers in the Large class are located above that line. Given a customer with a measurement of $(\text{inches}, \text{lbs})$, we predict the customer's class by checking whether $\text{lbs} > -3.5 * \text{inches} + 415$. If the inequality is true, then the customer is Large. Let's use the inequality to predict the customer classes. We store our predictions in a `y_pred` array and evaluate our predictions by printing the f-measure.

NOTE As we discussed in section 20, the f-measure is our preferred way of evaluating class prediction quality. As a reminder, the f-measure equals the harmonic mean of a classifier's precision and recall.

Listing 21.3 Predicting classes using a linear boundary

```
from sklearn.metrics import f1_score
y_pred = []
for inches, lbs in X:
    prediction = int(lbs > -3.5 * inches + 415)
    y_pred.append(prediction)

f_measure = f1_score(y_pred, y)
print(f'The f-measure is {f_measure:.2f}')
```

If `b` is a Python Boolean, `int(b)` returns 1 if the Boolean is True and 0 otherwise. Hence, we can return the class label for measurements $(\text{inches}, \text{lbs})$ by running `int(lbs > -3.5 * inches + 415)`.

The f-measure is 0.97

As expected, the f-measure is high. Given the inequality $\text{lbs} > -3.5 * \text{inches} + 415$, we can accurately classify our data. Furthermore, we can run the classification more concisely using vector dot products. Consider the following:

- 1 Our inequality rearranges to $3.5 * \text{inches} + \text{lbs} - 415 > 0$.
- 2 The dot product of two vectors $[x, y, z]$ and $[a, b, c]$ is equal to $a * x + b * y + c * z$.
- 3 If we take the dot product of vectors $[\text{inches}, \text{lbs}, 1]$ and $[3.5, 1, -415]$, the result equals $3.5 * \text{inches} + \text{lbs} - 415$.
- 4 Thus our inequality reduces to $w @ v > 0$, where `w` and `v` are both vectors, and `@` is the dot-product operator, as shown in figure 21.3.

Note that only one of the vectors is dependent on values of `lbs` and `inches`. The second vector, $[3.5, 1, -415]$, does not vary with customer measurements. Data scientists refer to this invariant vector as the *weight vector* or simply *weights*.

NOTE This name is unrelated to our measured customer weights in pounds.

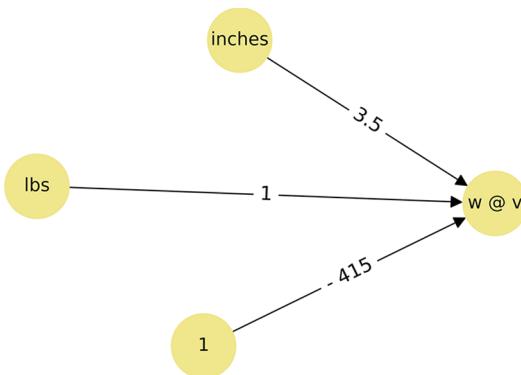


Figure 21.3 We can visualize the dot product between weights and [inches, lbs, 1] as a directed graph. In the graph, the leftmost nodes represent the measurements [inches, lbs, 1], and the edge weights represent the weights [3.5, 1, -415]. We multiply each node by its corresponding edge weight and sum the results. That sum equals the product between our two vectors v and w . Customer classification is determined by whether $w @ v > 0$.

Using vector dot products, we'll re-create the contents of `y_pred` in two lines of code:

- 1 Assign a weights vector to equal [3.5, 1, -415].
- 2 Classify each (inches, lbs) customer sample in X using the dot product of weights and [inches, lbs, 1].

Listing 21.4 Predicting classes using vector dot products

```

weights = np.array([3.5, 1, -415])
predictions = [int(weights @ [inches, lbs, 1] > 0) for inches, lbs in X]
assert predictions == y_pred
  
```

We can further consolidate our code if we use matrix multiplication.

Consider the following:

- 1 Currently, we must iterate over each [inches, lbs] row in matrix X and append a 1 to get vector [inches, lbs, 1].
- 2 Instead, we can concatenate a column of ones to matrix X and obtain a three-column matrix M . Each matrix row equals [inches, lbs, 1]. We refer to M as the *padded feature matrix*.
- 3 Running `[weights @ v for v in M]` returns the dot product between weights and every row in matrix M . Of course, this operation is equivalent to the matrix product between M and `weights`.
- 4 We can concisely compute the matrix product by running $M @ weights$.
- 5 Running $M @ weights > 0$ returns a Boolean array. Each element is true only if $3.5 * \text{inches} + \text{lbs} - 415 > 0$ for the corresponding customer measurements.

Essentially, $M @ weights > 0$ returns a Boolean vector whose i th value is true if $y_{\text{pred}}[i] == 1$ and false otherwise. We can transform the Booleans into numeric labels using NumPy's `astype` method. Consequently, we can generate our predictions just by running `(M @ weights > 0).astype(int)`. Let's confirm.

Listing 21.5 Predicting classes using matrix multiplication

```
M = np.column_stack([X, np.ones(X.shape[0])])
print("First five rows of our padded feature matrix are:")
print(np.round(M[:5], 2))

predictions = (M @ weights > 0).astype(int)
assert predictions.tolist() == y_pred
```

Concatenates a column of ones to feature matrix X to create the three-column matrix M

First five rows of our padded feature matrix are:

```
[[ 60.    126.24   1.  ]
 [ 60.1   104.28   1.  ]
 [ 60.2   105.52   1.  ]
 [ 60.3   100.47   1.  ]
 [ 60.4   120.25   1.  ]]
```

Checks to ensure that our predictions remain the same. Note that the matrix product returns a NumPy array, which must be converted to a list for this comparison.

We've boiled down customer classification to a simple matrix-vector product. This matrix product classifier is called a *linear classifier*. A weight vector is all that is required for a linear classifier to categorize input features. Here, we define a `linear_classifier` function that takes as input a feature matrix `X` and weight vector `weights`. It returns an array of class predictions.

Listing 21.6 Defining a linear classifier function

```
def linear_classifier(X, weights):
    M = np.column_stack([X, np.ones(X.shape[0])])
    return (M @ weights > 0).astype(int)

predictions = linear_classifier(X, weights)
assert predictions.tolist() == y_pred
```

Linear classifiers check whether weighted features and a constant add up to a value greater than zero. The constant value, which is stored in `weights[-1]`, is referred to as the *bias*. The remaining weights are called the *coefficients*. During classification, every coefficient is multiplied against its corresponding feature. In our case, the `inches` coefficient in `weights[0]` is multiplied against `inches`, and the `lbs` coefficient in `weights[1]` is multiplied against `lbs`. Thus, `weights` contains two coefficients and one bias, taking the form `[inches_coef, lbs_coef, bias]`.

We've derived our `weights` vector using a known decision boundary, but `weights` can also be computed directly from our training set (X, y) . In the following subsection, we discuss how to train a linear classifier. Training consists of finding coefficients and a bias that linearly separate our customer classes.

NOTE The trained results are not equal to `weights` because an infinite number of `weights` vectors satisfy the inequality $M @ \text{weights} > 0$. We can prove this by multiplying both sides by a positive constant k . Of course, $0 * k$ equals 0. Meanwhile, $\text{weights} * k$ produces a new vector w_2 . Hence, $M @ w_2$ is greater than 0 whenever $M @ \text{weights} > 0$ (and vice versa). There are infinite numbers of

k constants and hence an infinite number of w_2 vectors, but these vectors point in the same direction.

21.2 Training a linear classifier

We want to find a weight vector that optimizes class prediction on \mathbf{x} . Let's start by setting weights to equal three random values. Then we compute the f-measure associated with this random vector. We expect the f-measure to be very low.

Listing 21.7 Classification using random weights

```
np.random.seed(0)
weights = np.random.normal(size=3)
y_pred = linear_classifier(X, weights)
f_measure = f1_score(y_pred, y)

print('We inputted the following random weights:')
print(np.round(weights, 2))
print(f'\nThe f-measure is {f_measure:.2f}')
```

We inputted the following random weights:
[1.76 0.4 0.98]

The f-measure is 0.43

As expected, our f-measure is terrible! We can gain insight into why by printing `y_pred`.

Listing 21.8 Outputting the predicted classes

All our data points are assigned a class label of 1! The product of our weights and each feature vector is always greater than zero, so our weights must be too high. Lowering the weights will yield more Class 0 predictions. For instance, if we set the weights to $[0, 0, 0]$, all our class predictions equal 0.

Listing 21.9 Shifting the class predictions by lowering the weights

```
assert np.all(linear_classifier(X, [0, 0, 0]) == 0)
```

Lowering the weights yields more Class 0 predictions. Raising them yields more Class 1 predictions. Thus, we can intelligently raise and lower the weights until our predictions align with the actual class labels. Let's devise a strategy for adjusting the weights to match the labels. We start by adjusting the bias at weights [-1].

NOTE Adjusting the coefficients requires a bit more nuance, so for now, we'll focus on the bias.

Our goal is to minimize the difference between the predictions in `y_pred` and the actual labels in `y`. How do we do this? One simple strategy entails comparing each *predicted/actual* class-label pair. Based on each comparison, we can tweak the bias like this:

- If the prediction equals the actual class, then the prediction is correct. Hence, we will not modify the bias.
- If the prediction is 1 and the actual class is 0, then the weight is too high. Hence, we'll lower the bias by one unit.
- If the prediction is 0 and the actual class is 1, the weight is too low. Hence, we'll increase the bias by one unit.

Let's define a function to compute this bias shift based on predicted and actual labels.

NOTE Keep in mind that per existing conventions, the bias shift is subtracted from the weights. So, our `get_bias_shift` function returns a positive value when the weights are intended to decrease.

Listing 21.10 Computing the bias shift based on prediction quality

```
def get_bias_shift(predicted, actual):
    if predicted == actual:
        return 0
    if predicted > actual:
        return 1

    return -1
```

Mathematically, we can show that our `get_bias_shift` function is equivalent to `predicted - actual`. The following code definitively proves this for all four combinations of predicted and actual class labels.

Listing 21.11 Computing the bias shift using arithmetic

```
for predicted, actual in [(0, 0), (1, 0), (0, 1), (1, 1)]:
    bias_shift = get_bias_shift(predicted, actual)
    assert bias_shift == predicted - actual
```

It's worth noting that our single unit shift is an arbitrary value. Rather than shifting the bias by a single unit, we can shift it one-tenth of a unit, or 10 units, or 100 units. The value of the shift can be controlled by a parameter called the *learning rate*. The learning rate is multiplied against `predicted - actual` to adjust the shift size. So if we want to lower the shift to 0.1, we can easily do so by running `learning_rate * (predicted - actual)`, where `learning_rate` is equal to 0.1. This adjustment can influence the quality of training. We'll therefore redefine our `get_bias_shift` function with a `learning_rate` parameter that is preset to 0.1.

Listing 21.12 Computing the bias shift with a learning rate

```
def get_bias_shift(predicted, actual, learning_rate=0.1):
    return learning_rate * (predicted - actual)
```

Now we are ready to adjust our bias. Listing 21.13 iterates over each [inches, lbs, 1] vector in M . For every i th vector, we predict the class label and compare it to the actual class in $y[i]$.

NOTE As a reminder, the class prediction for every vector v is equal to $\text{int}(v @ \text{weights} > 0)$.

Using each prediction, we compute the bias shift and subtract it from the bias stored in $\text{weights}[-1]$. When all the iterations are complete, we print the adjusted bias and compare it to its original value.

Listing 21.13 Iteratively shifting the bias

```
def predict(v, weights): return int(v @ weights > 0) ← Predicts the class
starting_bias = weights[-1]
for i, actual in enumerate(y):
    predicted = predict(M[i], weights)
    bias_shift = get_bias_shift(predicted, actual)
    weights[-1] -= bias_shift

new_bias = weights[-1]
print(f"Our starting bias equaled {starting_bias:.2f}.")
print(f"The adjusted bias equals {new_bias:.2f}.")
```

Our starting bias equaled 0.98.
The adjusted bias equals -12.02

The bias has drastically decreased. This makes sense, given that our weights were way too large. Let's check whether the shift improved our f-measure.

Listing 21.14 Checking performance after the bias shift

```
y_pred = linear_classifier(X, weights)
f_measure = f1_score(y_pred, y)
print(f'The f-measure is {f_measure:.2f}')
```

The f-measure is 0.43

Our f-measure remains the same. Simply adjusting the bias is insufficient. We need to adjust the coefficients as well, but how? Naively, we could subtract the bias shift from every coefficient. We could just iterate over each training example and run $\text{weights} -= \text{bias_shift}$. Unfortunately, this naive approach is flawed: it always adjusts the coefficients, but it is dangerous to adjust the coefficients when their associated features are equal to zero. We'll illustrate why with a simple example.

Imagine that a blank entry in our customer dataset is erroneously recorded as $(0, 0)$. Our model treats this data point as a customer who weighs nothing and is 0 inches tall. Of course, such a customer is not physically possible, but that's beside the point. This theoretical customer is definitely Not Large, so their correct class label should be 0. When our linear model classifies the customer, it takes the dot product of $[0, 0, 1]$ and $[inches_coef, lbs_coef, bias]$. Of course, the coefficients are multiplied by zero and cancel out, so the final dot product is equal to just bias (figure 21.4). If $bias > 0$, the classifier incorrectly assigns a Class 1 label. Here, we'd need to decrease the bias using `bias_shift`. Would we also adjust the coefficients? No! Our coefficients did not impact the prediction. Thus, we can't evaluate the coefficient quality. For all we know, the coefficients are set to their optimal values. If so, then subtracting the bias shift would make the model worse.

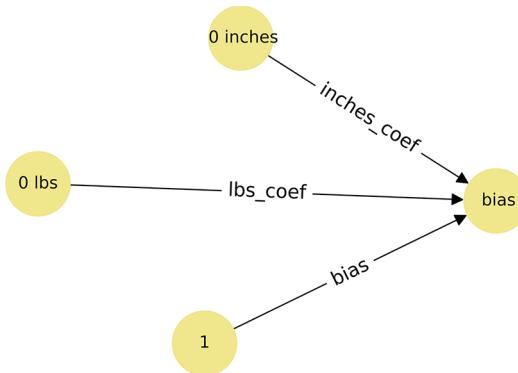


Figure 21.4 We can visualize the dot product between weights and $[0, 0, 1]$ as a directed graph. In the graph, the leftmost nodes represent the zero-value features, and the edge weights represent the coefficients and the bias. We multiply each node by its corresponding edge weight and sum the results. That sum equals bias. Customer classification is determined by whether $bias > 0$. The coefficients don't impact the prediction and therefore should not be altered in any way.

We should never shift `lbs_coef` if the `lbs` features equal zero. However, for nonzero inputs, subtracting `bias_shift` from `lbs_coef` remains appropriate. We could ensure this by setting the `lbs_coef` shift to equal `bias_shift` if `lbs` else 0. Alternatively, we can set the shift to equal `bias_shift * lbs`. This product is zero when `lbs` is zero. Otherwise, the product shifts `lbs_coef` in the same direction as the bias. Similarly, we can shift `inches_coef` by `bias_shift * inches` units. In other words, we'll shift each coefficient by the product of its feature and `bias_shift`.

NumPy allows us to compute our weight shifts all at once by running `bias_shift * [inches, lbs, 1]`. Of course, the $[inches, lbs, 1]$ vector corresponds to a row in the padded feature matrix M . Thus, we can adjust the weights based on each i th prediction by running `weights -= bias_shift * M[i]`.

With this in mind, let's iterate over each actual label in y and adjust the weights based on the predicted values. Then we check whether the f-measure has improved.

Listing 21.15 Computing all weight shifts in one line of code

```
old_weights = weights.copy()
for i, actual in enumerate(y):
    predicted = predict(M[i], weights)
    bias_shift = get_bias_shift(predicted, actual)
    weights -= bias_shift * M[i]

y_pred = linear_classifier(X, weights)
f_measure = f1_score(y_pred, y)

print("The weights previously equaled:")
print(np.round(old_weights, 2))
print("\nThe updated weights now equal:")
print(np.round(weights, 2))
print(f'\nThe f-measure is {f_measure:.2f}')
```

The weights previously equaled:
[1.76 0.4 -12.02]

The updated weights now equal:
[-4.64 2.22 -12.12]

The f-measure is 0.78

During the iteration, `inches_coef` has decreased by 6.39 units (from 1.76 to -4.63), and the bias has decreased by just 0.1 units (from -12.02 to -12.12). This discrepancy makes sense because the coefficient shift is proportional to height. Customers are on average 64 inches tall, so the coefficient shift is 64-fold greater than the bias. As we'll soon discover, large differences in weight shifts can lead to problems. Later, we eliminate these problems through a process called *standardization*; but first let's turn to our f-measure.

Our f-measure has risen from 0.43 to 0.78. The weight-shift strategy is working! What happens if we repeat the iteration 1,000 times? Let's find out. Listing 21.16 monitors the changes in f-measure over 1,000 weight-shift iterations. Then we plot each i th f-measure relative to the i th iteration (figure 21.5). We utilize the plot to monitor how the classifier's performance improves over time.

NOTE For the purpose of this exercise, we set weights to their original seeded random values. This allows us to monitor how performance improves relative to our starting f-measure of 0.43.

Listing 21.16 Tweaking the weights over multiple iterations

Tracks
performance
for the
weights at
each iteration

```
np.random.seed(0)
weights = np.random.normal(size=3) ← Sets the starting weights to random values
f_measures = []
for _ in range(1000): ← Repeats the weight-shift logic across 1,000 iterations
    y_pred = linear_classifier(X, weights)
    f_measures.append(f1_score(y_pred, y))
```

```

for i, actual in enumerate(y):
    predicted = predict(M[i], weights)
    bias_shift = get_bias_shift(predicted, actual)
    weights -= bias_shift * M[i] ←
    Shifts the weights by
    iterating over each
    predicted/actual
    class-label pair

print(f'The f-measure after 1000 iterations is {f_measures[-1]:.2f}')
plt.plot(range(len(f_measures)), f_measures)
plt.xlabel('Iteration')
plt.ylabel('F-measure')
plt.show()

```

The f-measure after 1000 iterations is 0.68

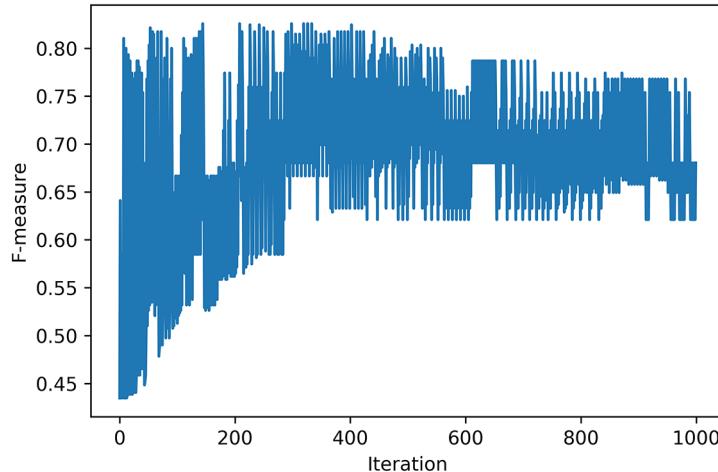


Figure 21.5 Plotted iterations vs. model f-measure. The model weights are tweaked at each iteration. The f-measure oscillates widely between low and reasonable values. These oscillations need to be eliminated.

The final f-measure is 0.68. Our classifier is very poorly trained. What happened? Well, according to our plot, the classifier performance oscillates wildly throughout the iterations. Sometimes the f-measure goes as high as 0.80; other times, it drops to approximately 0.60. After about 400 iterations, the classifier fluctuates nonstop between these two values. The rapid fluctuations are caused by a weight shift that is consistently too high. This is analogous to an airplane that is flying much too fast. Imagine an airplane flying 600 miles per hour after takeoff. The airplane maintains this rapid speed, allowing it to cover 1,500 miles in under three hours. However, as the airplane approaches its destination, the pilot refuses to slow down, so the plane overshoots its target airport and is forced to turn around. If the pilot doesn't lower the velocity, the plane will miss the landing again. This will lead to a sequence of never-ending aerial U-turns, similar to the oscillations in our plot. For the pilot, the solution is simple: reduce speed over the course of the flight.

We face an analogous solution: we should slowly lower the weight shift over each additional iteration. How do we lower the weight shift? One approach is to divide the shift by k for each k th iteration. Let's execute this strategy. We reset our weights to random values and iterate over k values ranging from 1 to 1,001. In each iteration, we set the weight shift equal to $\text{bias_shift} * M[i] / k$. Then we regenerate our performance plot (figure 21.6).

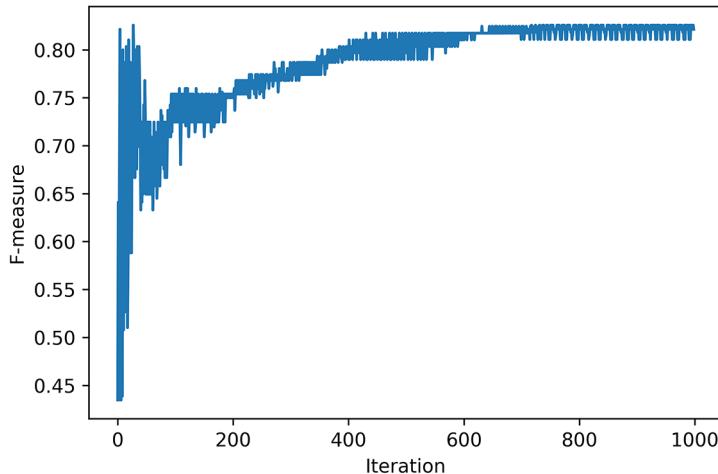


Figure 21.6 Plotted iterations vs. model f-measure. The model weights are tweaked at each k th iteration, in proportion to $1/k$. Dividing the weight shifts by k limits oscillation. Thus, the f-measure converges to a reasonable value.

Listing 21.17 Reducing weight shifts over multiple iterations

Trains a linear model from features X and labels y . The function is reused elsewhere in this section.

```
np.random.seed(0)
def train(X, y,
          predict=predict):
    M = np.column_stack([X, np.ones(X.shape[0])])
    weights = np.random.normal(size=X.shape[1] + 1)
    f_measures = []
    for k in range(1, 1000):
        y_pred = linear_classifier(X, weights)
        f_measures.append(f1_score(y_pred, y))

        for i, actual in enumerate(y):
            predicted = predict(M[i], weights)
            bias_shift = get_bias_shift(predicted, actual)
            weights -= bias_shift * M[i] / k
```

The predict function drives the weight shift by allowing us to compare predicted and actual class outputs. Later in this section, we modify predict to add nuance to the weight shifts.

A model with N features has $N + 1$ total weights representing N coefficients and one bias.

At each k th iteration, we dampen the weight shift by dividing by k . This reduces the weight-shift oscillations.

```

    return weights, f_measures
weights, f_measures = train(X, y)
print(f'The f-measure after 1000 iterations is {f_measures[-1]:.2f}')
plt.plot(range(len(f_measures)), f_measures)
plt.xlabel('Iteration')
plt.ylabel('F-measure')
plt.show()

```

The f-measure after 1000 iterations is 0.82

Our gradual weight-shift reduction was successful. The f-measure converges to a steady value of 0.82. We achieved convergence using a *perceptron training algorithm*. A *perceptron* is a simple linear classifier that was invented in the 1950s. Perceptrons are very easy to train. We just need to apply the following steps to training set (X, y):

- 1** Append a column of ones to feature matrix X to create a padded matrix M .
- 2** Create a weights vector containing $M.shape[1]$ random values.
- 3** Iterate over every i th row in M , and predict the i th class by running $M[i] @ weights > 0$.
- 4** Compare the i th prediction to the actual class label in $y[i]$. Then, compute the bias shift by running $(predicted - actual) * lr$, where lr is the learning rate.
- 5** Adjust the weights by running $weights -= bias_shift * M[i] / k$. Initially, the constant k is set to 1.
- 6** Repeat steps 3 through 5 over multiple iterations. At each iteration, increment k by 1 to limit oscillation.

Through repetition, the perceptron training algorithm eventually converges to a steady f-measure. However, that f-measure is not necessarily optimal. For instance, our perceptron converged to an f-measure of 0.82. This level of performance is acceptable, but it doesn't match our initial performance of 0.97. Our trained decision boundary doesn't separate the data as well as our initial decision boundary.

How do the two boundaries compare visually? We can easily find out. Using algebraic manipulation, we can transform a weight vector $[inches_coef, lbs_coef, bias]$ into a linear decision boundary equal to $lbs = -(inches_coef * inches + bias) / lbs_coef$. With this in mind, we'll plot both our new and old decision boundaries, together with our customer data (figure 21.7).

Listing 21.18 Comparing new and old decision boundaries

```

inches_coef, lbs_coef, bias = weights
def new_boundary(inches):
    return -(inches_coef * inches + bias) / lbs_coef

plt.plot(X[:,0], new_boundary(X[:,0]), color='k', linestyle='--',
         label='Trained Boundary', linewidth=2)

```

```
plt.plot(X[:,0], boundary(X[:,0]), color='k', label='Initial Boundary')
plot_customers(X, y)
plt.legend()
plt.show()
```

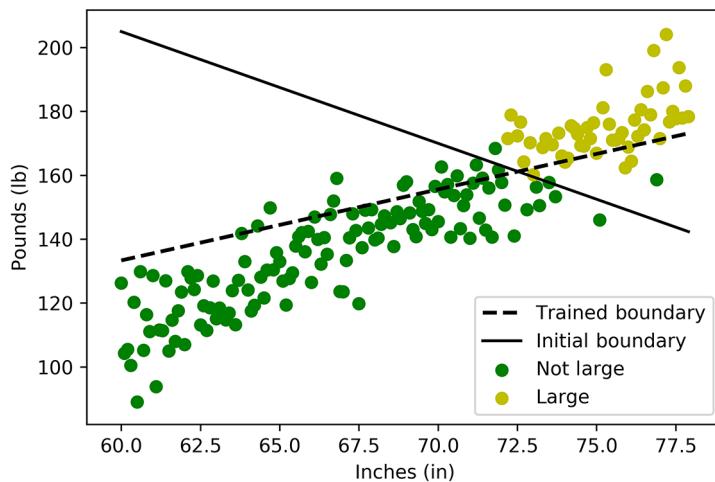


Figure 21.7 A plot of customer measurements: inches vs. lbs. Two linear boundaries separate the Large and Not Large customers. The trained boundary's separation is worse relative to the baseline boundary.

Our trained linear boundary is inferior to our initial linear boundary, but this is not the fault of the perceptron algorithm. Instead, the training is hindered by large, fluctuating features in matrix X . In the next subsection, we discuss why large X values impede performance. We'll limit that impediment through a process called *standardization*, in which X is adjusted to equal $(X - X.\text{mean}(\text{axis}=0)) / X.\text{std}(\text{axis}=0)$.

21.2.1 Improving perceptron performance through standardization

Perceptron training is impeded by large feature values in X . This is due to the discrepancy between the coefficient shifts and bias shifts. As we discussed, the coefficient shift is proportional to the associated feature value. Furthermore, these values can be quite high. For instance, the average customer height is greater than 60 inches: the `inches_coef` shift is more than 60-fold higher than the bias shift, so we can't tweak the bias by a little without tweaking the coefficients by a lot. Thus, by tuning the bias, we are liable to significantly shift `inches_coef` toward a less-than-optimal value.

Our training lacks all nuance because the coefficient shifts are much too high. However, we can lower these shifts by reducing column means in matrix X . Additionally, we need to lower the dispersion in the matrix. Otherwise, unusually large customer measurements could cause overly large coefficient shifts. We therefore need to

decrease the column means and standard deviations. To start, let's print the current values of `X.mean(axis=0)` and `X.std(axis=0)`.

Listing 21.19 Printing feature means and standard deviations

```
means = X.mean(axis=0)
stds = X.std(axis=0)
print(f"Mean values: {np.round(means, 2)}")
print(f"STD values: {np.round(stds, 2)}")

Mean values: [ 68.95 146.56]
STD values: [ 5.2 23.26]
```

The feature means and standard deviations are relatively high. How do we make them smaller? Well, as we learned in section 14, it is trivial to shift a dataset's mean toward zero: we simply need to subtract `means` from `X`. Adjusting the standard deviations is less straightforward, but mathematically we can show that $(X - \text{means}) / \text{stds}$ returns a matrix whose column dispersions all equal 1.0.

NOTE Here is the proof. Running `X - means` returns a matrix whose every column `v` has a mean of 0.0. Hence, the variance of each `v` equals $\frac{\sum_{e \in v} e^2}{N}$, where `N` is the number of column elements. Of course, this operation can be expressed as a simple dot product, `v @ v / N`. The standard deviation `std` equals the square root of the variance, so `std = sqrt(v @ v) / sqrt(N)`. Note that `sqrt(v @ v)` is equal to the magnitude of `v`, which we can express as `norm(v)`. Thus, `std = norm(v) / sqrt(N)`. Suppose we divide `v` by `std` to generate a new vector `v2`. Since `v2 = v / std`, we expect the magnitude of `v2` to equal `norm(v) / std`. The standard deviation of `v2` is equal to `norm(v2) / sqrt(N)`. By substituting out `norm(v2)`, we get `norm(v) / (sqrt(N) * std)`. However, `norm(v) / sqrt(N) = std`. So the standard deviation of `v2` reduces to `std / std`, which equals 1.0.

This simple process is called *standardization*. Let's standardize our feature matrix by running $(X - \text{means}) / \text{stds}$. The resulting matrix has column means of 0 and column standard deviations of 1.0.

Listing 21.20 Standardizing the feature matrix

```
def standardize(X):
    return (X - means) / stds
X_s = standardize(X)
assert np.allclose(X_s.mean(axis=0), 0)
assert np.allclose(X_s.std(axis=0), 1)
```

Standardizes measurements derived from the customer distribution. We reuse the function elsewhere in this section.

We now check whether training on the standardized feature matrix improves our results. We also plot the trained decision boundary relative to the standardized data (figure 21.8).

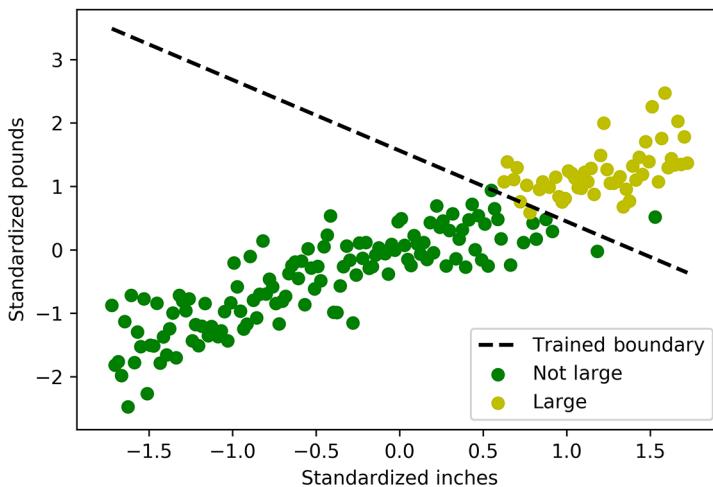


Figure 21.8 A plot of standardized customer measurements. A trained decision boundary separates Large and Not Large customers. The trained boundary's separation is on par with the baseline decision boundary in figure 21.2.

Listing 21.21 Training on the standardized feature matrix

```

np.random.seed(0)
weights, f_measures = train(X_s, y)
print(f'After standardization, the f-measure is {f_measures[-1]:.2f}')

def plot_boundary(weights):
    a, b, c = weights
    new_boundary = lambda x: -(a * x + c) / b
    plt.plot(X_s[:, 0], new_boundary(X_s[:, 0]), color='k', linestyle='--',
             label='Trained Boundary', linewidth=2)
    plot_customers(X_s, y, xlabel='Standardized Inches',
                  ylabel='Standardized Pounds')
    plt.legend()
    plt.show()

plot_boundary(weights)

```

After standardization, the f-measure is 0.98

Plots the linear decision boundary derived from weights, together with the standardized data

Transforms the weights into a linear function

Success! Our new f-measure equals 0.98. This f-measure is higher than our baseline value of 0.97. Furthermore, the angle of our new decision boundary closely resembles the baseline boundary in figure 21.2. We achieved improvement in performance through standardization.

NOTE Standardization is similar to normalization. Both techniques lower the values in inputted data and eliminate unit differences (such as inches versus

centimeters). For some tasks, such as PCA analysis, the two techniques can be used interchangeably. However, when we're training linear classifiers, standardization achieves superior results.

We should note that our trained classifier now requires all input data to be standardized before classification. Hence, given any new data d , we need to classify that data by running `linear_classifier(standardize(d), weights)`.

Listing 21.22 Standardizing new classifier inputs

```
new_data = np.array([[63, 110], [76, 199]])
predictions = linear_classifier(standardize(new_data), weights)
print(predictions)

[0 1]
```

We've standardized our data and achieved a high level of performance. Unfortunately, this optimal f-measure is still not guaranteed by the training algorithm. Perceptron training quality can fluctuate, even if the algorithm is run repeatedly on the same training set. This is due to the random weights assigned in the initial training step: certain starting weights converge to a worse decision boundary. Let's illustrate the model's inconsistency by training a perceptron five times. After each training run, we check whether the resulting f-measure falls below our initial baseline of 0.97.

Listing 21.23 Checking a perceptron's training consistency

```
np.random.seed(0)
poor_train_count = sum([train(X_s, y)[1][-1] < 0.97 for _ in range(5)])
print("The f-measure fell below our baseline of 0.97 in "
      f"{poor_train_count} out of 5 training instances")
```

```
The f-measure fell below our baseline of 0.97 in 4 out of 5
training instances
```

In 80% of instances, the trained model performance falls below the baseline. Our basic perceptron model is clearly flawed. We discuss its flaws in the subsequent subsection. In the process, we derive one of the most popular linear models in data science: logistic regression.

21.3 Improving linear classification with logistic regression

During class prediction, our linear boundary makes a simple binary decision. However, as we learned in section 20, not all predictions should be treated equally. Sometimes we are more confident in some predictions than others. For instance, if all neighbors in a KNN model vote unanimously for Class 1, we are 100% confident in that prediction. But if just six of nine neighbors vote for Class 1, we are 66% confident in that prediction. This measure of confidence is lacking in our perceptron model.

The model has just two outputs: 0 and 1, based on whether the data lies above or below the decision boundary.

What about a data point that lies exactly on the decision boundary? Currently, our logic will assign Class 0 to that point.

NOTE If measurements in v lie on the decision boundary, then $\text{weights} @ v == 0$. Hence $\text{int}(\text{weights} @ v > 0)$ returns 0.

However, that assignment is arbitrary. If the point is not positioned above or below the decision boundary, we cannot decide on either class! Thus, our confidence in either class should equal 50%. What if we shift the point 0.0001 units above the boundary? Our confidence in Class 1 should go up, but not by much. We can assume that the Class 1 likelihood increases to 50.001% while the Class 0 likelihood decreases to 49.999%. Only if the point is positioned far from the boundary should our confidence rise sharply, as illustrated in figure 21.9. For instance, if the point is 100 units above the boundary, then our confidence in Class 1 should reach 100% and our confidence in Class 0 should drop to 0%.

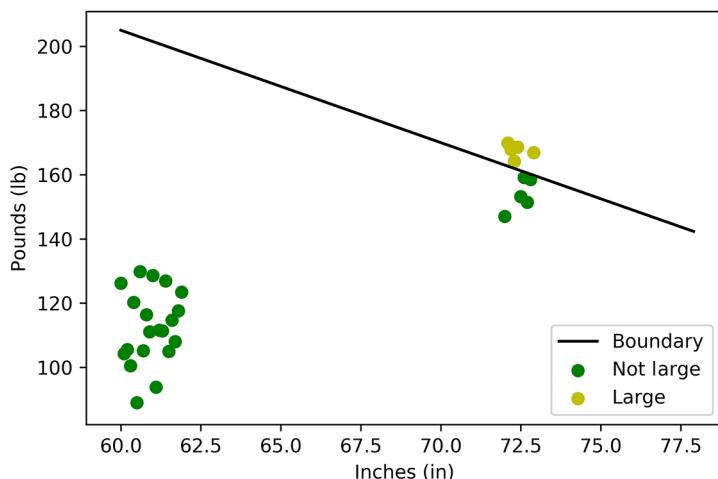


Figure 21.9 A plot of customer measurements: inches vs. lbs. A linear boundary separates our two customer classes. Only customers who are either close to or far from the boundary are displayed. Customers who are too close to the boundary are harder to classify. We are much more confident in the class label of those customers who lie far from the decision boundary.

Class confidence is determined by distance from the boundary and position relative to the boundary. If a point lies 100 units below the decision boundary, its Class 1 and 0 likelihoods should be flipped. We can capture both distance and position with *directed*

distance. Unlike regular distance, directed distance can be negative. We'll assign each point a negative distance if it falls below the decision boundary.

NOTE Hence, if a point is 100 units below the boundary, its directed distance to the boundary equals -100 .

Let's select a function to compute the Class 1 confidence based on directed distance from the boundary. The function should rise to 1.0 as the directed distance rises to infinity. Conversely, it should drop to 0.0 as the directed distance drops to negative infinity. Finally, the function should equal 0.5 when the directed distance equals zero. In this book, we have encountered a function that fits these criteria: in section 7, we introduced the cumulative distribution function of the normal curve. This S-shaped curve equals the probability of randomly drawing a value from a normal distribution that's less than or equal to some z . The function starts at 0.0 and increases to 1.0. It's also equal to 0.5 when $z == 0$. As a reminder, the cumulative distribution can be computed by running `scipy.stats.norm.cdf(z)`. Here, we plot the CDF for z -values ranging from -10 to 10 (figure 21.10).

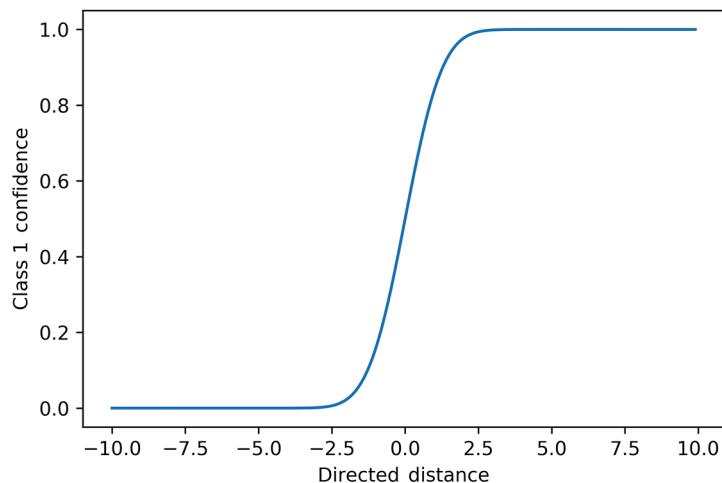


Figure 21.10 A cumulative distribution function of a normal distribution. The S-shaped curve starts at 0.0 and rises toward 1.0. It equals 0.5 when the input is 0.0. This plot fits our criteria for capturing uncertainty based on the directed distance from the decision boundary.

Listing 21.24 Measuring uncertainty using `stats.norm.cdf`

```
from scipy import stats
z = np.arange(-10, 10, 0.1)
assert stats.norm.cdf(0.0) == 0.5
plt.plot(z, stats.norm.cdf(z))
plt.xlabel('Directed Distance')
```

Confirms the curve has equal confidence in both classes when z lies directly on the 0.0 threshold

```
plt.ylabel('Confidence in Class 1')
plt.show()
```

The S-shaped cumulative normal distribution curve fits our stated confidence criteria. It's an adequate function for computing classifier uncertainty. But in recent decades, this curve's usage has fallen out of favor. There are several reasons. One of the most pressing concerns is that no exact formula exists for calculating `stats.norm.cdf`: instead, the area under the normal distribution is computed by approximation. Consequently, data scientists have turned to a different S-shaped curve, whose straightforward formula is easy to remember: the *logistic* curve. The logistic function of z is $1 / (1 - e^{-z})$ where e is a constant equal to approximately 2.72. Much like the cumulative normal distribution, the logistic function ranges from 0 to 1 and is equal to 0.5 when $z == 0$. Let's plot the logistic curve, together with `stats.norm.cdf` (figure 21.11).

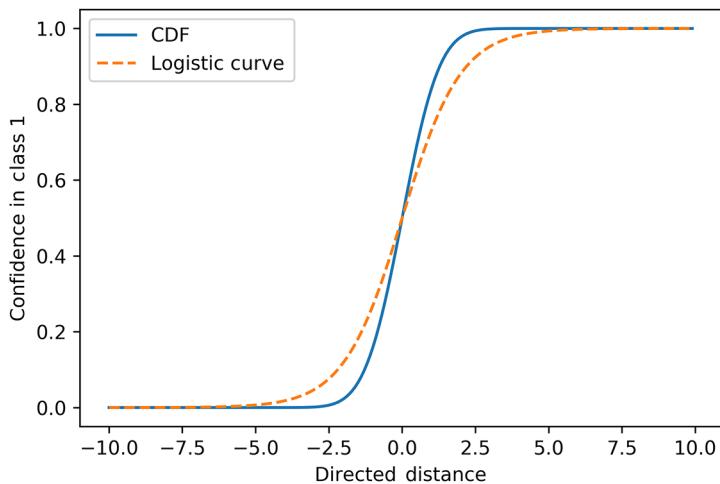


Figure 21.11 A cumulative distribution function of a normal distribution, plotted together with the logistic curve. Both S-shaped curves start at 0.0 and rise toward 1.0. They equal 0.5 when the input is 0.0. Both curves fit our criteria for capturing uncertainty based on the directed distance from the decision boundary.

Listing 21.25 Measuring uncertainty using the logistic curve

```
from math import e
plt.plot(z, stats.norm.cdf(z), label='CDF')
plt.plot(z, 1 / (1 + e ** -z), label='Logistic Curve', linestyle='--')
plt.xlabel('Directed Distance')
plt.ylabel('Confidence in Class 1')
plt.legend()
plt.show()
```

The two curves don't precisely overlap, but they both

- Equal approximately 1 when $z > 5$
- Equal approximately 0 when $-z > 5$
- Equal an ambiguous value between 0 and 1 when $-5 < z < 5$
- Equal 0.5 when $z == 0$

Hence, we can use the logistic curve as our measure of uncertainty. Let's utilize the curve to assign Class 1 label likelihoods for all our customers. This requires us to compute the directed distance between each customer's measurements and the boundary. Computing these distances is surprisingly simple: we just need to execute $M @ \text{weights}$, where M is the padded feature matrix. Essentially, we were computing these distances all along—we just weren't fully utilizing them until now!

NOTE Let's quickly prove that $M @ \text{weights}$ returns the distances to the decision boundary. For clarity's sake, we'll use our initial weights of $[3.5, 1, -415]$, representing the decision boundary $\text{lbs} = -3.5 * \text{inches} - 415$. Thus, we're taking the distance between measurements ($\text{inches}, \text{lbs}$) and the decision boundary point ($\text{inches}, -3.5 * \text{inches} + 415$). Of course, the x-axis coordinates both equal inches , so we're taking the distance along the y-axis. This distance equals $\text{lbs} - (-3.5 * \text{inches} + 415)$. The formula rearranges to $3.5 * \text{inches} + \text{lbs} - 415$. This equals the dot product of $[3.5, 1, -415]$ and $[\text{inches}, \text{lbs}, 1]$. The first vector equals weights , and the second vector represents a row in M . Therefore, $M @ \text{weights}$ returns an array of directed distances.

If $M @ \text{weights}$ returns the directed distances, then $1 / (1 + e^{-(M @ \text{weights})})$ returns the Class 1 likelihoods. Listing 21.26 plots distance versus likelihood. We also add our binary perceptron predictions to the plot: these correspond to $M @ \text{weights} > 0$ (figure 21.12).

NOTE As a reminder, we computed weights by training on the standardized features in X_s . Hence, we must append a column of ones to X_s to pad the feature matrix.

Listing 21.26 Comparing logistic uncertainty to the perceptron's predictions

```
M = np.column_stack([X_s, np.ones(X_s.shape[0])])
distances = M @ weights
likelihoods = 1 / (1 + e ** -distances)
plt.scatter(distances, likelihoods, label='Class 1 Likelihood')
plt.scatter(distances, distances > 0,
           label='Perceptron Prediction', marker='x')

plt.xlabel('Directed Distance')
plt.legend()
plt.show()
```

Perceptron predictions are determined by distances > 0 . Note that Python automatically converts Booleans True and False to integers 1 and 0, so we can plug distances > 0 directly into plt.scatter without an integer conversion.

Directed distances to the boundary equals the product of the padded feature matrix and weights.

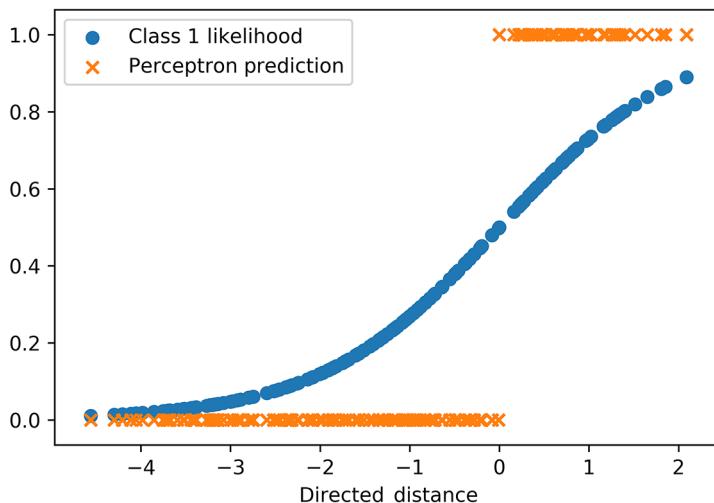


Figure 21.12 Class 1 likelihoods from the logistic curve plotted together with the perceptron predictions. The likelihoods show nuance, while the perceptron predictions are limited to either 0 or 1.

The plotted logistic likelihoods continuously increase with directed distance. In contrast, the perceptron predictions are brutally simple: the perceptron has either 100% confidence in a Class 1 label or 0% confidence. Interestingly, both the logistic curve and the perceptron are 0% confident when the directed distance is very negative. However, as the directed distance rises, the plots begin to diverge. The logistic plot is more conservative: its confidence increases slowly and mostly falls below 85%. Meanwhile, the perceptron model's confidence jumps to 100% when distances > 0 . This jump is unwarranted. The model is overconfident, like an inexperienced teenager—it is bound to make mistakes! Fortunately, we can teach the model caution by incorporating the uncertainty captured by the logistic curve.

We can incorporate uncertainty by updating our weight-shift computation. Currently, the weight shift is proportional to predicted - actual, where the variables represent predicted and actual class labels. Instead, we can make the shift proportional to confidence(predicted) - actual, where confidence(predicted) captures our confidence in the predicted class. In a perceptron model, confidence(predicted) always equals 0 or 1. By contrast, in the nuanced logistic model, the weight shift takes on a more granular range of values.

Consider, for example, a data point that has a class label of 1 and lies directly on the decision boundary. The perceptron computes a 0 weight shift when presented with this data during training, so the perceptron will not adjust its weights. It learns absolutely nothing from the observation. By contrast, a logistic model returns a weight shift that's proportional to $0.5 - 1 = -0.5$. The model will tweak its appraisal of the

class label's uncertainty and adjust the weights accordingly. Unlike the perceptron, the logistic model has a flexible capacity to learn.

Let's update our model training code to incorporate logistic uncertainty. We simply need to swap our predict function output from `int(weights @ v > 0)` to `1 / (1 + e ** - (weights @ v))`. Here, we make the swap using two lines of code. Then we train our improved model to generate a new vector of weights and plot the new decision boundary to validate our result (figure 21.13).

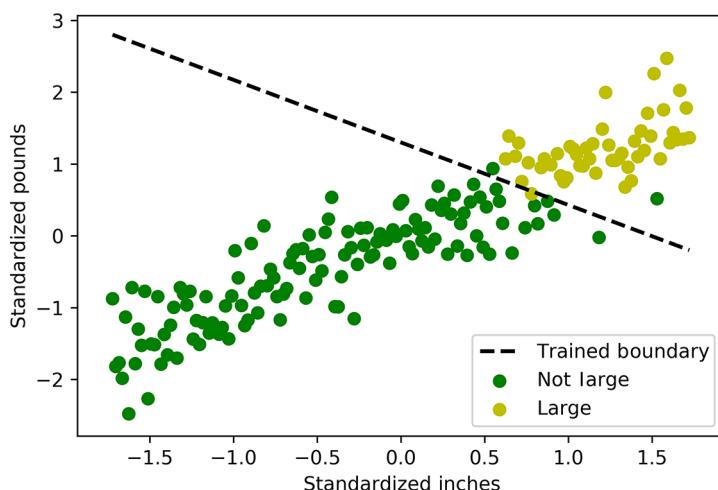


Figure 21.13 A plot of standardized customer measurements. A logically trained decision boundary separates Large and Not Large customers. The trained boundary's separation is on par with the baseline decision boundary in figure 21.2.

Listing 21.27 Incorporating uncertainty into training

Our train function takes an optional row-level class predictor called `predict`. This predictor is preset to return `int(weights @ v > 0)`. Here, we swap it out for the more nuanced `logistic_predict` function.

```
np.random.seed(0)
def logistic_predict(v, weights): return 1 / (1 + e ** -(weights @ v))
def train_logistic(X, y): return train(X, y, predict=logistic_predict)
logistic_weights = train_logistic(X_s, y) [0]
plot_boundary(logistic_weights)
```

The learned decision boundary is nearly identical to that of the perceptron output. However, our `train_logistic` function is subtly different: it produces more consistent results than the perceptron. Previously, we showed that the trained perceptron model performs below our baseline in four out of five training runs. Is this the case for `train_logistic`? Let's find out.

Listing 21.28 Checking the logistic model's training consistency

```
np.random.seed(0)
poor_train_count = sum([train_logistic(X_s, y)[1][-1] < 0.97
                       for _ in range(5)])
print("The f-measure fell below our baseline of 0.97 in "
      f"{poor_train_count} out of 5 training instances")

The f-measure fell below our baseline of 0.97 in 0 out of 5
training instances
```

The trained model does not fall below the baseline in any of the runs, so it is superior to the perceptron. This superior model is called a *logistic regression classifier*. The model's training algorithm is also commonly called *logistic regression*.

NOTE Arguably, this name is not semantically correct. Classifiers predict categorical variables, while regression models predict numeric values. Technically speaking, the logistic regression classifier uses logistic regression to predict the numeric uncertainty, but it is not a regression model. But the term *logistic regression* has become ubiquitous with the term *logistic regression classifier* in the machine learning community.

A logistic regression classifier is trained just like a perceptron, but with one small difference. The weight shift is not proportional to `int(distance - y[i] > 0)`, where `distance = M[i] @ weights`. Instead, it is proportional to `1 / (1 + e ** -distance) - y[i]`. This difference leads to much more stable performance over random training runs.

NOTE What happens if the weight shift is directly proportional to `distance - y[i]`? Well, the trained model learns to minimize the distance between a line and the values in `y`. For classification purposes, this is not really useful; but for regression, it is invaluable. For instance, if we set `y` to equal `lbs` and `X` to equal `inches`, we could train a line to predict customer weight using customer height. With two lines of code, we utilize `train` to implement this type of linear regression algorithm. Can you figure out how?

21.3.1 Running logistic regression on more than two features

We've trained our logistic regression model on two customer measurements: height (inches) and weight (lbs). However, our `train_logistic` function can process any number of input features. We'll prove this by adding a third feature: customer waist circumference. On average, waist circumference is equal to 45% of an individual's height. We'll use this fact to simulate the customer waist measurements. Then we'll input all three measurements into `train_logistic` and evaluate the trained model's performance.

Listing 21.29 Training a three-feature logistic regression model

```
np.random.seed(0)
random_fluctuations = np.random.normal(size=X.shape[0], scale=0.1)
```

```

waist = 0.45 * X[:,0] + random_fluctuations
X_w_waist = np.column_stack([X_s, (waist - waist.mean()) / waist.std()])
weights, f_measures = train_logistic(X_w_waist, y)

print("Our trained model has the following weights:")
print(np.round(weights, 2))
print(f'\nThe f-measure is {f_measures[-1]:.2f}')

```

Our trained model has the following weights:
[1.65 2.91 1.26 -4.08]

The f-measure is 0.97

We need to standardize waists before appending that array to other standardized customer measurements.

Each waist measurement equals 45% of a customer's height, with a random fluctuation.

The trained three-feature model continues to perform exceptionally, with an f-measure of 0.97. The main difference is that the model now contains four weights. The first three weights are coefficients corresponding to the three customer measurements, and the final weight is the bias. Geometrically, the four weights represent a higher-dimensional linear boundary that takes the form of a three-dimensional line called a *plane*. The plane separates our two customer classes in 3D space, as illustrated in figure 21.14.

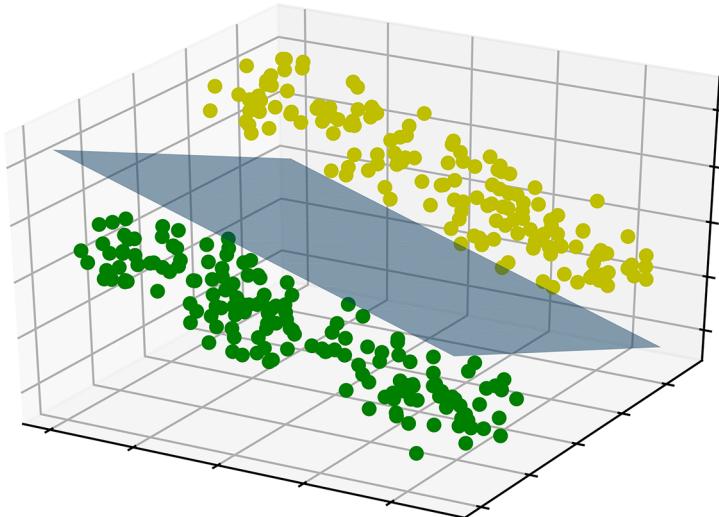


Figure 21.14 Linear classification in 3D space. A linear plane slices through the data like a cleaver and separates that data into two distinct classes.

Similarly, we can optimize for linear separation in any arbitrary number of dimensions. The resulting weights represent a multidimensional linear decision boundary. Shortly, we will run logistic regression on a dataset with 13 features. Scikit-learn's implementation of the logistic regression classifier will prove useful for this purpose.

21.4 Training linear classifiers using scikit-learn

Scikit-learn has a built-in class for logistic regression classification. We start by importing this `LogisticRegression` class.

NOTE Scikit-learn also includes a perceptron class, which can be imported from `sklearn.linear_model`.

Listing 21.30 Importing scikit-learn's LogisticRegression class

```
from sklearn.linear_model import LogisticRegression
```

Next, we initialize the classifier object `clf`.

Listing 21.31 Initializing scikit-learn's LogisticRegression classifier

```
clf = LogisticRegression()
```

As discussed in section 20, we can train any `clf` by running `clf.fit(X, y)`. Let's train our logistic classifier using the two-feature standardized matrix `X_s`.

Listing 21.32 Training scikit-learn's LogisticRegression classifier

```
clf.fit(X_s, y)
```

The classifier has learned the weight vector `[inches_coef, lbs_coef, bias]`. The vector's coefficients are stored in the `clf.coef_` attribute. Meanwhile, the bias must be accessed separately using the `clf.intercept_` attribute. Combining these attributes gives us the full vector, which can be visualized as a decision boundary (figure 21.15).

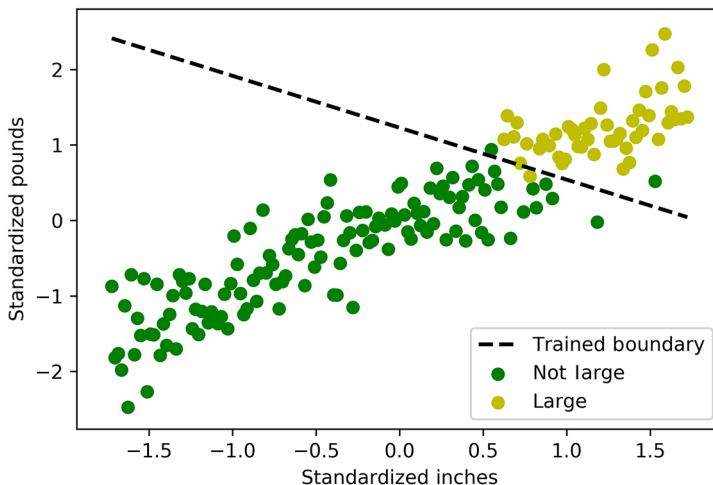


Figure 21.15 A plot of standardized customer measurements. A logically trained decision boundary derived with scikit-learn separates Large and Not Large customers.

Listing 21.33 Accessing the trained decision boundary

```

coefficients = clf.coef_
bias = clf.intercept_
print(f"The coefficients equal {np.round(coefficients, 2)}")
print(f"The bias equals {np.round(bias, 2)}")
plot_boundary(np.hstack([clf.coef_[0], clf.intercept_]))

The coefficients equal [[2.22 3.22]]
The bias equals [-3.96]

```

We can make predictions on new data by executing `clf.predict`. As a reminder, the inputted data must be standardized for our predictions to make sense.

Listing 21.34 Predicting classes with the linear classifier

```

new_data = np.array([[63, 110], [76, 199]])
predictions = clf.predict(standardize(new_data))
print(predictions)

[0 1]

```

Additionally, we can output the class-label probabilities by running `clf.predict_proba`. These probabilities represent the class-label uncertainties generated by the logistic curve.

Listing 21.35 Outputting the uncertainty associated with each class

```

probabilities = clf.predict_proba(standardize(new_data))
print(probabilities)

[[9.99990471e-01 9.52928118e-06]
 [1.80480919e-03 9.98195191e-01]]

```

In the previous two code listings, we've relied on a custom `standardize` function to standardize our input data. Scikit-learn includes its own standardization class called `StandardScaler`. Here, we import and initialize that class.

Listing 21.36 Initializing scikit-learn's standardization class

```

from sklearn.preprocessing import StandardScaler
standard_scaler = StandardScaler()

```

Running `standard_scaler.fit_transform(X)` returns a standardized matrix. The means of the matrix columns equal 0, and the standard deviations equal 1. Of course, the matrix is identical to our existing standardized matrix, `X_s`.

Listing 21.37 Standardizing training data using scikit-learn

```

X_transformed = standard_scaler.fit_transform(X)
assert np.allclose(X_transformed.mean(axis=0), 0)

```

```
assert np.allclose(X_transformed.std(axis=0), 1)
assert np.allclose(X_transformed, X_s)
```

The standard_scaler object has learned the means and standard deviations associated with our feature matrix, so it can now standardize data based on these statistics. Listing 21.38 standardizes our new_data matrix by running standard_scaler.transform(new_data). We pass the standardized data into our classifier. The predicted output should equal our previously seen predictions array.

Listing 21.38 Standardizing new data using scikit-learn

```
data_transformed = standard_scaler.transform(new_data)
assert np.array_equal(clf.predict(data_transformed), predictions)
```

By combining the LogisticRegression and StandardScaler classes, we can train logistic models on complex inputs. In the next subsection, we train a model that can process more than two features and predict more than two class labels.

Relevant scikit-learn linear classifier methods

- clf = LogisticRegression()—Initializes a logistic regression classifier
- scaler = StandardScaler()—Initializes a standard scaler
- clf.fit(scaler.fit_transform(X))—Trains the classifier on standardized data
- clf.predict(scaler.transform(new_data))—Predicts classes from the standardized data
- clf.predict_proba(scaler.transform(new_data))—Predicts class probabilities from the standardized data

21.4.1 Training multiclass linear models

We've shown how linear classifiers can find decision boundaries that separate two classes of data. However, many problems require us to differentiate between more than two classes. Consider, for example, the centuries-old practice of wine tasting. Some experts are renowned for being able to distinguish between many classes of wine using sensory input. Suppose we try to build a wine-tasting machine. Using sensors, the machine will detect chemical patterns in a glass of wine. These measurements will be fed into a linear classifier as features. The classifier will then identify the wine (impressing us with its refinement and sophistication). To train the linear classifier, we need a training set. Fortunately, such a dataset is provided via scikit-learn. Let's load this dataset by importing and running the load_wine function and then print the feature names and class labels from the data.

Listing 21.39 Importing scikit-learn's wine dataset

```
from sklearn.datasets import load_wine
data = load_wine()
```

```

num_classes = len(data.target_names)
num_features = len(data.feature_names)
print(f"The wine dataset contains {num_classes} classes of wine:")
print(data.target_names)
print(f"\nIt contains the {num_features} features:")
print(data.feature_names)

The wine dataset contains 3 classes of wine:
['class_0' 'class_1' 'class_2']

It contains the 13 features:
['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium',
'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins',
'color_intensity', 'hue', 'od280/od315_of_diluted_wines', 'proline']

```

The dataset has
“flavonoids” misspelled
as “flavanoids.”

The dataset contains 13 measured features, including alcohol content (Feature 0), magnesium level (Feature 4), and hue (Feature 10). It also contains three classes of wine.

NOTE The actual identities of these wines are lost to time, although they probably correspond to different types of red wines such as Cabernet, Merlot, and Pinot Noir.

How do we train a logistic regression model to distinguish between the three wine types? Well, we could initially train a simple binary classifier to check whether a wine belongs to Class 0. Alternatively, we could train a different classifier that predicts whether a wine belongs to Class 1. Finally, a third classifier would determine whether the wine is a Class 2 wine. This is essentially scikit-learn’s built-in logic for multiclass linear classification. Given three class categories, scikit-learn learns three decision boundaries, one for each class. Then the model computes three different predictions on inputted data and chooses the prediction with the highest confidence level.

NOTE This is another reason computed confidence is critical to carrying out linear classification.

If we train our logistic regression pipeline on the three-class wine data, we’ll obtain three decision boundaries corresponding to Classes 0, 1, and 2. Each decision boundary will have its own weight vector. Every weight vector will have a bias, so the trained model will have three biases. These three biases will be stored in a three-element `clf.intercept_` array. Accessing `clf.intercept_[i]` will provide us with the bias for Class *i*. Let’s train the wine model and print the resulting three biases.

Listing 21.40 Training a multiclass wine predictor

```

X, y = load_wine(return_X_y=True)
clf.fit(standard_scaler.fit_transform(X), y)
biases = clf.intercept_

print(f"We trained {biases.size} decision boundaries, corresponding to "
      f"the {num_classes} classes of wine.\n")

```

```

for i, bias in enumerate(biases):
    label = data.target_names[i]
    print(f"The {label} decision boundary has a bias of {bias:.2f}")

```

We trained 3 decision boundaries, corresponding to the 3 classes of wine.

```

The class_0 decision boundary has a bias of 0.41
The class_1 decision boundary has a bias of 0.70
The class_2 decision boundary has a bias of -1.12

```

Along with the bias, each decision boundary must have coefficients. The coefficients are used to weigh the inputted features during classification, so there is a one-to-one correspondence between coefficients and features. Our dataset contains 13 features representing various properties of wine, so each decision boundary must have 13 corresponding coefficients. The coefficients for the three different boundaries can be stored in a 3-by-13 matrix. In scikit-learn, that matrix is contained in `clf.coef_`. Each i th row of the matrix corresponds to the boundary of Class i , and each j th column corresponds to the j th feature coefficient. For example, we know that Feature 0 equals the alcohol content of a wine, so `clf_coeff_[2][0]` equals the Class 2 boundary's alcohol coefficient.

Let's visualize the coefficient matrix as a heatmap (figure 21.16). This will allow us to display the feature names and class labels corresponding to the rows and columns. Note that the lengthy feature names are easier to read if we display a transpose of the matrix. Thus, we input `clf.coeff_.T` into `sns.heatmap`.

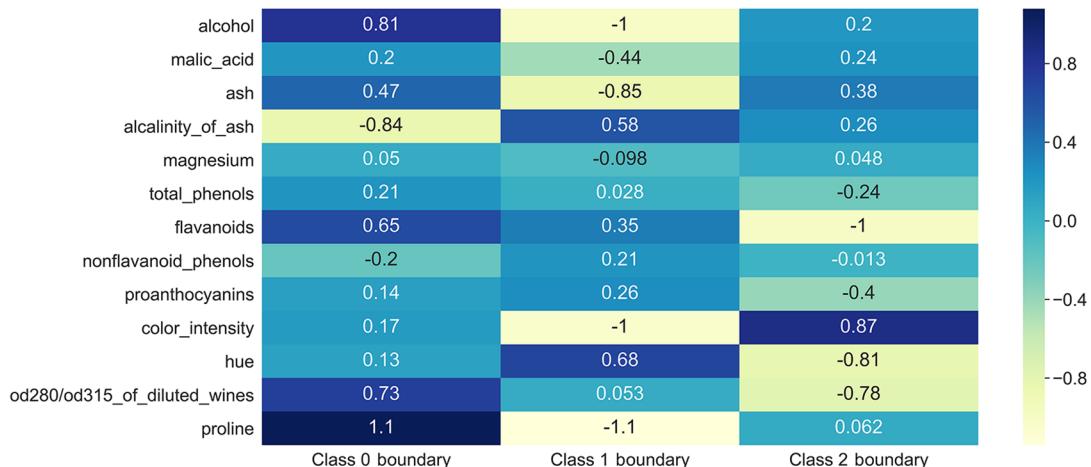


Figure 21.16 A heatmap representing 13 feature coefficients across the three decision boundaries

Listing 21.41 Displaying a transpose of the coefficient matrix

```

import seaborn as sns
plt.figure(figsize = (20, 10))
coefficients = clf.coef_
sns.heatmap(coefficients.T, cmap='YlGnBu', annot=True,
            xticklabels=[f"Class {i} Boundary" for i in range(3)],
            yticklabels=data.feature_names)
plt.yticks(rotation=0)
sns.set(font_scale=2)
plt.show()

```

← Adjusts the width and height of the plotted heatmap to 20 inches and 10 inches, respectively

← Transposes the coefficient matrix for easier display of the coefficient names

← Adjusts the label font for readability

In the heatmap, the coefficients vary from boundary to boundary. For example, the alcohol coefficients equal -0.81 , -1 , and 0.2 for class boundaries 0, 1, and 2, respectively. Such differences in coefficients can be very useful; they allow us to better understand how the inputted features drive prediction.

Relevant scikit-learn linear classifier attributes

- `clf.coef_`—Accesses the coefficient matrix of a trained linear classifier
- `clf.intercept_`—Accesses all bias values in a trained linear classifier

21.5 Measuring feature importance with coefficients

In section 20, we discussed how the KNN classifier is not interpretable. Using KNN, we can predict the class associated with the inputted features, but we cannot comprehend why these features belong to that class. Fortunately, the logistic regression classifier is easier to interpret. We can gain insights into how the model's features drive the prediction by examining their corresponding coefficients.

Linear classification is driven by the weighted sum of the features and the coefficients. So if a model takes three features A, B, and C and relies on three coefficients [1, 0, 0.25], then the prediction is partially determined by the value $A + 0.25 * C$. Note that in this example, feature B is zeroed out. Multiplying a zero coefficient by a feature always produces a zero value, so that feature never impacts the model's predicted output.

Now, let's consider a feature whose coefficient is very close to zero. The feature influences predictions, but its impact is minimal. Alternatively, if a coefficient is far from zero, the associated feature will impact the model's prediction much more heavily. Basically, coefficients with higher absolute values have more impact on the model, so their associated features are more important when assessing model performance. For instance, in our example, feature A is the most impactful because its coefficient is furthest from zero (figure 21.17).

Features can be rated by their coefficients to assess their *feature importance*: a score that ranks the usefulness of features during classification. Different classifier models

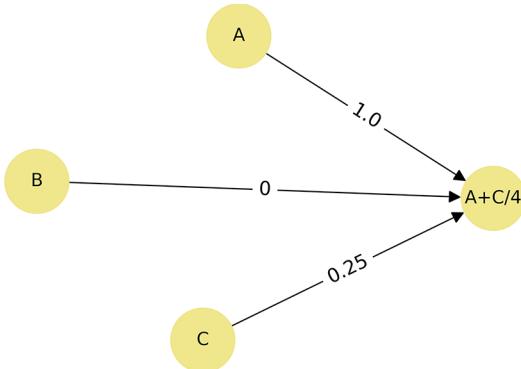


Figure 21.17 We can visualize the weighted sum of features [A, B, C] and coefficients [1, 0, 0.25] as a directed graph. In the graph, the leftmost nodes represent the features, and the edge weights represent the coefficients. We multiply each node by its corresponding edge weight and sum the results. That sum equals $A + C / 4$, so A is four times more impactful than C. Meanwhile, B is zeroed out and has no impact on the final results.

yield different feature importance scores. In linear classifiers, the absolute values of the coefficients serve as crude measures of importance.

NOTE The models presented in section 22 have more nuanced feature importance scores.

What feature is most useful for correctly detecting a Class 0 wine? We can check by sorting the features based on the absolute values of the Class 0 coefficients in `clf.coef_[0]`.

Listing 21.42 Ranking Class 0 features by importance

```

def rank_features(class_label):
    absolute_values = np.abs(clf.coef_[class_label])
    for i in np.argsort(absolute_values)[::-1]:
        name = data.feature_names[i]
        coef = clf.coef_[class_label][i]
        print(f"{name}: {coef:.2f}")

rank_features(0)

proline: 1.08
alcalinity_of_ash: -0.84
alcohol: 0.81
od280/od315_of_diluted_wines: 0.73
flavanoids: 0.65
ash: 0.47
total_phenols: 0.21
malic_acid: 0.20
nonflavanoid_phenols: -0.20
color_intensity: 0.17
proanthocyanins: 0.14
hue: 0.13
magnesium: 0.05
  
```

Ranks the features based on the absolute value of the coefficients in `clf.coef_[class_label]`

Computes the absolute values

Sorts feature indices by absolute values in descending order

Proline appears at the top of the ranked list; it is a chemical commonly found in wine whose concentration is dependent on grape type. Proline concentration is the most

important feature for identifying Class 0 wines. Now, let's check which feature drives Class 1 wine identification.

Listing 21.43 Ranking Class 1 features by importance

```
rank_features(1)

proline: -1.14
color_intensity: -1.04
alcohol: -1.01
ash: -0.85
hue: 0.68
alcalinity_of_ash: 0.58
malic_acid: -0.44
flavanoids: 0.35
proanthocyanins: 0.26
nonflavanoid_phenols: 0.21
magnesium: -0.10
od280/od315_of_diluted_wines: 0.05
total_phenols: 0.03
```

Proline concentration is the most important feature for both Class 0 and Class 1 wines. However, that feature influences the two class predictions in different ways: the Class 0 proline coefficient is positive (1.08), while the Class 1 coefficient is negative (-1.14). The coefficient signs are very important. Positive coefficients increase the weighted sum of linear values, and negative values decrease that sum. Therefore, proline decreases the weighted sum during Class 1 classification. That decrease leads to a negative directed distance from the decision boundary, so the Class 1 likelihood drops to zero. Meanwhile, the positive Class 0 coefficient has a completely opposite effect. Thus a high proline concentration implies the following:

- A wine is less likely to be a Class 1 wine.
- A wine is more likely to be a Class 0 wine.

We can check our hypothesis by plotting histograms of proline concentration for the two classes of wine (figure 21.18).

Listing 21.44 Plotting proline histograms across Classes 0 and 1 wines

```
index = data.feature_names.index('proline')
plt.hist(X[y == 0][:, index], label='Class 0')
plt.hist(X[y == 1][:, index], label='Class 1', color='y')
plt.xlabel('Proline concentration')
plt.legend()
plt.show()
```

On average, proline concentration is higher in Class 0 than in Class 1. This difference serves as a signal for distinguishing between the two wines. Our classifier has successfully learned this signal. By probing the classifier's coefficients, we have also learned something about the chemical makeup of different wines.

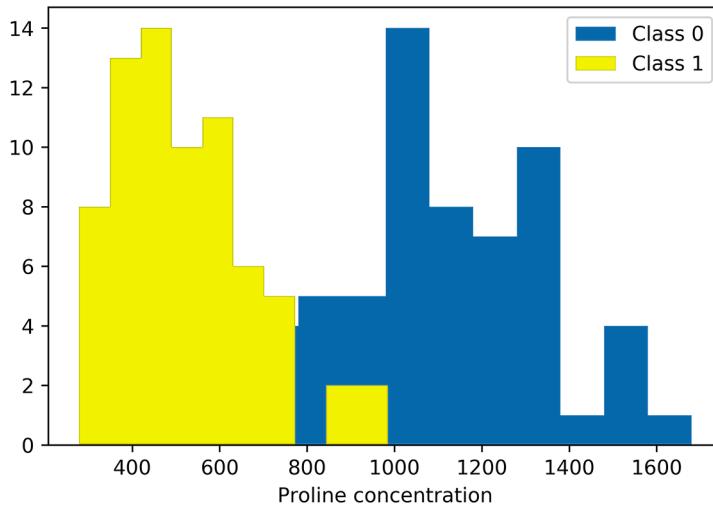


Figure 21.18 A histogram of proline concentrations across Class 0 and Class 1 wines. The Class 0 concentrations are noticeably greater than those of Class 1. Our classifier has picked up on this signal by making proline the top-ranking coefficient for both Class 0 and Class 1.

Unlike KNN models, logistic regression classifiers are interpretable. They’re also easy to train and fast to run, so linear classifiers are an improvement over KNN models. Unfortunately, linear classifiers still suffer from some very serious flaws that limit their practical use in certain circumstances.

21.6 Linear classifier limitations

Linear classifiers work poorly on raw data. As we’ve observed, standardization is required to achieve the best results. Similarly, linear models cannot handle categorical features without data preprocessing. Suppose we’re building a model to predict whether a pet will be adopted from a shelter. Our model can predict on three pet categories: cat, dog, and bunny. The simplest way to represent these categories is with numbers: 0 for cat, 1 for dog, and 2 for bunny. However, this representation will cause a linear model to fail. The model gives bunnies twice the attention that it gives dogs, and it entirely ignores cats. For the model to treat each pet with equal attention, we must transform the categories into a three-element binary vector v . If a pet belongs to category i , $v[i]$ is set to 1. Otherwise, $v[i]$ equals 0. Thus, we represent a cat as $v = [1, 0, 0]$, a dog as $v = [0, 1, 0]$, and a bunny as $v = [0, 0, 1]$. This vectorization is similar to the text vectorization seen in section 13. We can carry it using scikit-learn. Still, such transformations can be cumbersome. The models covered in the subsequent section can analyze raw data without additional preprocessing.

NOTE Categorical variable vectorization is often called *one-hot encoding*. Scikit-learn includes a OneHotEncoder transformer, which can be imported from `sklearn.preprocessing`. The OneHotEncoder class can automatically detect and vectorize all categorical features in your training set.

The most serious limitation of linear classifiers is right there in the name: linear classifiers learn *linear* decision boundaries. More precisely, a line (or plane in higher dimensions) is required to separate the classes of data. However, there are countless classification problems that are not linearly separable. Consider, for example, the problem of classifying urban and non-urban households. Let's assume prediction is driven by the distance to the city center. All households less than two distance units from the center are classified as urban; all other households are considered suburban. The following code simulates these households with a 2D normal distribution. We also train a logistic regression classifier to distinguish between household classes. Finally, we visualize the model's linear boundary and the actual households in 2D space (figure 21.19).

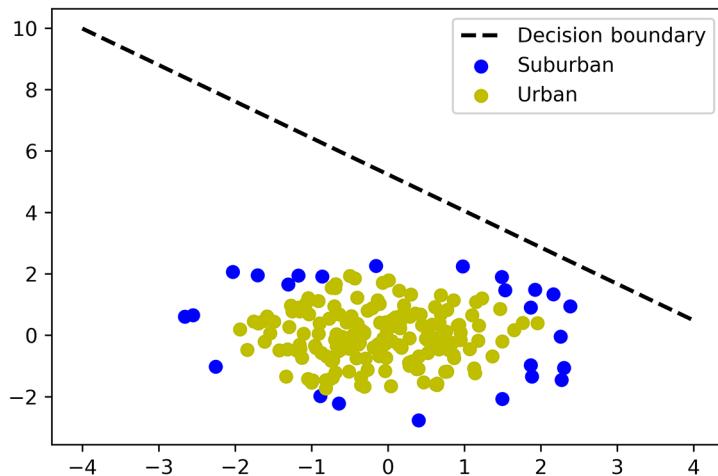


Figure 21.19 Simulated households plotted relative to the city center at $(0, 0)$. Households that are closer to the center are considered urban. No linear separation exists between the urban and suburban households, so the trained linear boundary is unable to distinguish between them.

Listing 21.45 Simulating a nonlinearly separable scenario

The x and y coordinates of each household are drawn from two standard normal distributions.

```
np.random.seed(0)
X = np.array([[np.random.normal(), np.random.normal()]
             for _ in range(200)])
y = (np.linalg.norm(X, axis=1) < 2).astype(int)
```

The city center is located at a coordinate $(0, 0)$. A household's spatial distance from the center is therefore equal to its norm. Households within two units of the center are labeled urban.

```

clf = LogisticRegression()
clf.fit(X, y)
weights = np.hstack([clf.coef_[0], clf.intercept_]) ←
a, b, c = weights
boundary = lambda x: -(a * x + c) / b
plt.plot(range(-4, 5), boundary(range(-4, 5)), color='k', linestyle='--',
         linewidth=2, label='Decision Boundary') ←
for i in [0, 1]:
    plt.scatter(X[y == i][:, 0], X[y == i][:, 1],
                label=['Suburban', 'Urban'][i],
                color=['b', 'y'][i])
plt.legend()
plt.show()

```

Our data was drawn from a distribution with a mean of 0 and an std of 1. Standardization is therefore not required to train the linear model.

Plots the trained decision boundary alongside the household coordinates

The linear boundary fails to separate the classes. The dataset's geometry does not allow for such a separation. In data science terms, the data is *not linearly separable*. Hence, a linear classifier cannot be adequately trained. We need to run a nonlinear approach. In the subsequent section, we learn about decision tree techniques that can overcome this limitation.

Summary

- In certain instances, we can separate data classes using *linear decision boundaries*. All data points below the linear boundary are classified as belonging to Class 0, and all data points above the linear boundary are classified as belonging to Class 1. Effectively, the linear boundary checks whether the weighted features and a constant add to a value greater than zero. The constant value is called the *bias*, and the remaining weights are called the *coefficients*.
- Through algebraic manipulation, we can transform linear classification into the matrix product inequality defined by $M @ \text{weights} > 0$. Such multiplication-driven classification defines a *linear classifier*. The matrix M is a *padded feature matrix* with an appended column of ones, weights is a vector, and the final vector element is the bias. The remaining weights are coefficients.
- To obtain a good decision boundary, we start by randomly initializing weights . We then iteratively adjust the weights based on the difference between predicted and actual classes. In the simplest possible linear classifier, this weight shift is proportional to the difference between the predicted and actual classes. Hence, the weight shift is proportional to one of three values: $-1, 0$, or 1 .
- We should never tweak a coefficient if the associated feature equals zero. We can ensure this constraint if we multiply the weight shift by the corresponding feature value in matrix M .
- Iteratively tweaking the weights can cause the classifier to fluctuate between good and subpar performance. To limit oscillation, we need to lower the weight

shift with every subsequent iteration. This can be done by dividing the weight shift by k over each k th iteration.

- Iterative weight adjustment can converge on a decent decision boundary, but it is not guaranteed to locate the optimal decision boundary. We can improve the boundary by decreasing the data's mean and standard deviation. Such *standardization* can be achieved if we subtract the means and then divide by the standard deviations. The resulting dataset has a mean of 0 and a standard deviation of 1.
- The simplest linear classifier is known as a *perceptron*. Perceptrons perform well, but their results can be inconsistent. The failure of the perceptrons are partially caused by a lack of nuance. Points closer to the decision boundary are more ambiguous with regard to their classification. We can capture this uncertainty using an S-shaped curve that ranges between values of 0 and 1. The cumulative normal distribution function serves as a decent measure of uncertainty, but the simpler *logistic curve* is easier to compute. The logistic curve is equal to $1 / (1 + e^{-z})$.
- We can incorporate uncertainty into model training by setting the weight shift proportionally to $\text{actual} - 1 / (1 + e^{-\text{distance}})$. Here, distance represents the directed distance to the decision boundary. We can compute all directed distances by running `M @ weights`.
- A classifier trained using logistic uncertainty is called a *logistic regression classifier*. This classifier yields more consistent results than a simple perceptron.
- Linear classifiers can be extended to N classes by training N different linear decision boundaries.
- The coefficients in a linear classifier serve as a measure of *feature importance*. Coefficients with the largest absolute values map to features that have a significant impact on a model's predictions. The coefficient's sign determines whether the presence of a feature indicates the presence or absence of a class.
- Linear classification models fail whenever the data is not *linearly separable* and a good linear decision boundary does not exist.



Training nonlinear classifiers with decision tree techniques

This section covers

- Classifying datasets that are not linearly separable
- Automatically generating `if/else` logical rules from training data
- What is a decision tree?
- What is a random forest?
- Training tree-based models using scikit-learn

Thus far, we have investigated supervised learning techniques that rely on the geometry of data. This association between learning and geometry does not align with our everyday experiences. On a cognitive level, people do not learn through abstract spatial analysis; they learn by making logical inferences about the world. These inferences can then be shared with others. A toddler realizes that by throwing a fake tantrum, they can sometimes get an extra cookie. A parent realizes that indulging the toddler inadvertently leads to even more bad behavior. A student realizes that through preparation and study, they will usually do well on their exam. Such realizations are not particularly new; they are part of our collective social wisdom. Once a useful logical inference has been made, it can be shared with others.

for broader use. Such sharing is the basis of modern science. A scientist realizes that certain viral proteins make good targets for a drug. They publish their inferences in a journal, and that knowledge propagates across the entire scientific community. Eventually, a new antiviral drug is developed based on the scientific findings.

In this section, we learn how to algorithmically derive logical inferences from our training data. These simple, logical rules will provide us with predictive models that are not constrained by the data's geometry.

22.1 Automated learning of logical rules

Let's analyze a seemingly trivial problem. Suppose a lightbulb hangs above a stairwell. The bulb is connected to two switches at the top and bottom of the stairs. When both switches are off, the bulb stays off. If either switch is turned on, the bulb shines. However, if both switches are flipped on, the bulb turns off. This arrangement allows us to activate the light when we are at the bottom of the stairs and then deactivate it after we ascend.

We can represent the off and on states of the switches and the bulb as binary digits 0 and 1. Given two switch variables `switch0` and `switch1`, we can trivially show that the bulb is on whenever `switch0 + switch1 == 1`. Using the material from the previous two sections, can we train a classifier to learn this trivial relationship? We'll find out by storing every possible light switch combination in a two-column feature matrix `x`. Then we'll plot the matrix rows in 2D space while marking each point based on the corresponding on/off state of the bulb (figure 22.1). The plot will give us insight into how KNN and linear classifiers can handle this classification problem.

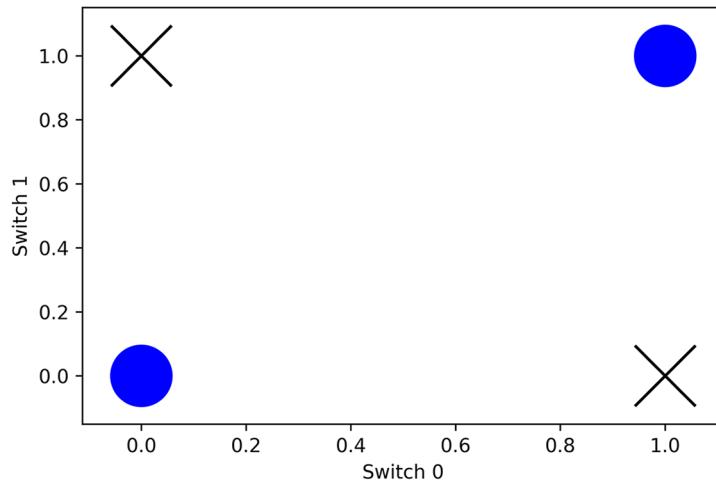


Figure 22.1 Plotting all the states of the light switch system. An activated lightbulb is represented by an X, and a deactivated bulb is represented by an O. The nearest neighbors of each O are X markers (and vice versa). Thus, KNN cannot be used for classification. Also, there's no linear separation between the X markers and O markers, so linear classification cannot be applied.

Listing 22.1 Plotting the two-switch problem in 2D space

```

import numpy as np
import matplotlib.pyplot as plt
X = np.array([[0, 0], [1, 0], [0, 1], [1, 1]])
y = (X[:,0] + X[:,1] == 1).astype(int)

for i in [0, 1]:
    plt.scatter(X[y == i][:,0], X[y == i][:,1],
                marker=['o', 'x'][i], color=['b', 'k'][i],
                s=1000)
plt.xlabel('Switch 0')
plt.ylabel('Switch 1')
plt.show()

```

The four plotted points lie on the four corners of a square. Each pair of points in the same class is positioned diagonally on that square, and all adjacent points belong to different classes. The two nearest neighbors of each on-switch combination are members of the off class (and vice versa). Therefore, KNN will fail to properly classify the data. There also is no linear separation between the labeled classes, so we cannot draw a linear boundary without cutting through a diagonal that connects two identically classified points. Consequently, training a linear classifier is also out of the question. What should we do? One approach is to define two nested if/else statements as our prediction model. Let's code and test this if/else classifier.

Listing 22.2 Classifying data using nested if/else statements

```

def classify(features):
    switch0, switch1 = features

    if switch0 == 0:
        if switch1 == 0:
            prediction = 0
        else:
            prediction = 1
    else:
        if switch1 == 0:
            prediction = 1
        else:
            prediction = 0

    return prediction

for i in range(X.shape[0]):
    assert classify(X[i]) == y[i]

```

Our if/else classifier is 100% accurate, but we didn't train it. Instead, we programmed the classifier ourselves. Manual model construction does not count as supervised machine learning, so we need to find a way to automatically derive accurate if/else statements from the training data. Let's figure out how.

We start with a simple training example. Our training set represents a series of recorded observations between a single light switch and a single bulb. Whenever the switch is on, the bulb is on, and vice versa. We randomly flip the lightbulb on and off and record what we see. The state of the bulb is recorded in a `y_simple` array. Our single feature, corresponding to the switch, is recorded in a single-column `X_simple` matrix. Of course, `X_simple[i][0]` will always equal `y[i]`. Let's generate this basic training set.

Listing 22.3 Generating a single-switch training set

```
np.random.seed(0)
y_simple = np.random.binomial(1, 0.5, size=10)
X_simple = np.array([[e] for e in y_simple])
print(f"features: {X_simple}")
print(f"\nlabels: {y_simple}")

features: [[1]
[1]
[1]
[0]
[1]
[0]
[1]
[1]
[0]
[0]]

labels: [1 1 1 1 0 1 0 1 1 0]
```

The state of the switch is always equal to the state of the bulb.

The state of the bulb is simulated using a random coin flip.

Next, we count all observations in which the switch is off and the light is off.

Listing 22.4 Counting the off-state co-occurrences

```
count = (X_simple[:,0][y_simple == 0] == 0).sum()
print(f"In {count} instances, both the switch and the light are off")
```

In 3 instances, both the switch and the light are off

Now, let's count the instances in which both the switch and the lightbulb are turned on.

Listing 22.5 Counting the on-state co-occurrences

```
count = (X_simple[:,0][y_simple == 1] == 1).sum()
print(f"In {count} instances, both the switch and the light are on")
```

In 7 instances, both the switch and the light are on

These co-occurrences will prove useful during classifier training. Let's track the counts more systematically in a co-occurrence matrix `M`. The matrix rows track the off/on state of the switch, and the matrix columns track the state of the bulb. Each

element $M[i][j]$ counts the number of examples where the switch is in state i and the bulb is in state j . Hence, $M[0][0]$ should equal 7, and $M[1][1]$ should equal 3.

We now define a `get_co_occurrence` function to compute the co-occurrence matrix. The function takes as input a training set (X, y) , as well as a column index `col`. It returns the co-occurrences between all classes in y and all feature states in $X[:, col]$.

Listing 22.6 Computing a co-occurrence matrix

```
def get_co_occurrence(X, y, col=0):
    co_occurrence = []
    for i in [0, 1]:
        counts = [(X[:, col][y == i] == j).sum()
                   for j in [0, 1]]
        co_occurrence.append(counts)

    return np.array(co_occurrence)

M = get_co_occurrence(X_simple, y_simple)
assert M[0][0] == 3
assert M[1][1] == 7
print(M)

[[3 0]
 [0 7]]
```

Using `get_co_occurrence`, we've computed matrix M . All co-occurrences lie along the diagonal of the matrix. The bulb is never on when the switch is flipped off, and vice versa. However, let's suppose that there's a flaw in the switch toggle. We flip the switch off, but the bulb stays on! Let's add this anomalous observation to our data and then recompute matrix M .

Listing 22.7 Adding a flawed mismatch to the data

```
X_simple = np.vstack([X_simple, [1]])
y_simple = np.hstack([y_simple, [0]])
M = get_co_occurrence(X_simple, y_simple)
print(M)

[[3 1]
 [0 7]]
```

When we shut off the switch, the lightbulb turns off most of the time, but it doesn't turn off every time. How accurately can we predict the state of the lightbulb if we know that the switch is off? To find out, we must divide $M[0]$ by $M[0].sum()$. Doing so produces a probability distribution over possible lightbulb states whenever the switch state is set to 0.

Listing 22.8 Computing bulb probabilities when the switch is off

```
bulb_probs = M[0] / M[0].sum()
print("When the switch is set to 0, the bulb state probabilities are:")
print(bulb_probs)
```

```
prob_on, prob_off = bulb_probs
print(f"\nThere is a {100 * prob_on:.0f}% chance that the bulb is off.")
print(f"There is a {100 * prob_off:.0f}% chance that the bulb is on.")
```

When the switch is set to 0, the bulb state probabilities are:
[0.75 0.25]

There is a 75% chance that the bulb is off.
There is a 25% chance that the bulb is on.

When the switch is off, we should assume that the bulb is off. Our guess will be correct 75% of the time. This fraction of correction fits our definition of accuracy from section 20, so when the switch is off, we can predict the state of the bulb with 75% accuracy.

Now let's optimize the accuracy for the scenario in which the switch is on. We start by computing `bulb_probs` over `M[1]`. Next, we choose the switch state corresponding to the maximum probability. Basically, we infer that the bulb state is equal to `bulb_probs.argmax()` with an accuracy score of `bulb_probs.max()`.

Listing 22.9 Predicting the state of the bulb when the switch is on

```
bulb_probs = M[1] / M[1].sum()
print("When the switch is set to 1, the bulb state probabilities are:")
print(bulb_probs)

prediction = ['off', 'on'][bulb_probs.argmax()]
accuracy = bulb_probs.max()
print(f"\nWe assume the bulb is {prediction} with "
      f"{100 * accuracy:.0f}% accuracy")
```

When the switch is set to 1, the bulb state probabilities are:
[0. 1.]

We assume the bulb is on with 100% accuracy

When the switch is off, we assume that the bulb is off with 75% accuracy. When the switch is on, we assume the bulb is on with 100% accuracy. How do we combine these values into a single accuracy score? Naively, we could simply average 0.75 and 1.0, but that approach would be erroneous. The two accuracies should not be weighted evenly since the switch is on nearly twice as often as it is off. We can confirm by summing over the columns of our co-occurrence matrix `M`. Running `M.sum(axis=1)` returns the count of off states and on states for the switch.

Listing 22.10 Counting the on and off states of the switch

```
for i, count in enumerate(M.sum(axis=1)):
    state = ['off', 'on'][i]
    print(f"The switch is {state} in {count} observations.")
```

```
The switch is off in 4 observations.  
The switch is on in 7 observations.
```

The switch is on more frequently than it is off. Hence, to get a meaningful accuracy score, we need to take the weighted average of 0.75 and 1.0. The weights should correspond to the on/off switch counts obtained from M.

Listing 22.11 Computing total accuracy

```
accuracies = [0.75, 1.0]
total_accuracy = np.average(accuracies, weights=M.sum(axis=1))
print(f"Our total accuracy is {100 * total_accuracy:.0f}%)
```

```
Our total accuracy is 91%
```

If the switch is off, we predict that the bulb is off; otherwise, we predict that the bulb is on. This model is 91% accurate. Furthermore, the model can be represented as a simple if/else statement in Python. Most importantly, we are able to train the model from scratch using the following steps:

- 1 Choose a feature in our feature matrix X.
- 2 Count the co-occurrences between the two possible feature states and the two class types. These co-occurrence counts are stored in a two-by-two matrix M.
- 3 For row i in M, compute the probability distribution of classes whenever the feature is in state i. This probability distribution is equal to M[i] / M[i].sum(). There are just two rows in M, so we can store distributions in two variables: probs0 and probs1.
- 4 Define the if portion of our conditional model. If the feature equals 0, we return a label of probs0.argmax(). This maximizes the accuracy of the if statement. That accuracy is equal to probs0.max().
- 5 Define the else portion of our conditional model. When the feature does not equal 0, we return a label of probs1.argmax(). This maximizes the accuracy of the else statement. That accuracy is equal to probs1.max().
- 6 Combine the if and else statements into a single conditional if/else statement. Occasionally, probs0.argmax() will equal probs1.argmax(). In such circumstances, using an if/else statement is redundant. Instead, we can return the trivial rule f"prediction = {probs0.argmax()}".
- 7 The accuracy of the combined if/else statement equals the weighted average of probs0.max() and probs1.max(). The weights correspond to the count of feature states obtained by summing up the columns of M.

Let's define a `train_if_else` function to carry out these seven steps. The function returns a trained if/else statement along with the corresponding accuracy.

Listing 22.12 Training a simple if/else model

Creates the written if/else statement

If both parts of the conditional statement return the same prediction, we simplify the statement to just that prediction.

```
def train_if_else(X, y, feature_col=0, feature_name='feature'):
    M = get_co_occurrence(X, y, col=feature_col)
    probs0, probs1 = [M[i] / M[i].sum() for i in [0, 1]] ←

    if_else = f"""if {feature_name} == 0:
        prediction = {probs0.argmax()}
    else:
        prediction = {probs1.argmax()}
    """.strip()

    if probs0.argmax() == probs1.argmax():
        if_else = f"prediction = {probs0.argmax()}" ←

    accuracies = [probs0.max(), probs1.max()]
    total_accuracy = np.average(accuracies, weights=M.sum(axis=1))
    return if_else, total_accuracy

if_else, accuracy = train_if_else(X_simple, y_simple, feature_name='switch')
print(if_else)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if switch == 0:
    prediction = 0
else:
    prediction = 1
```

This statement is 91% accurate.

Trains an if/else statement on the training set (X, y) and returns the written statement along with the corresponding accuracy. The statement is trained on the feature in $X[:, feature_col]$. The corresponding feature name is stored in `feature_name`.

22.1.1 Training a nested if/else model using two features

Let's return to our system of two light switches connecting to a single staircase bulb. As a reminder, all states of this system are represented by dataset (X, y) , generated in listing 22.1. Our two features, `switch0` and `switch1`, correspond to columns 0 and 1 of matrix `X`. However, the `train_if_else` function can only train on one column at a time. Let's train two separate models: one on `switch0` and another on `switch1`. How well will each of the models perform? We'll find out by outputting their accuracies.

Listing 22.13 Training models on the two-switch system

```
feature_names = [f"switch{i}" for i in range(2)]
for i, name in enumerate(feature_names):
    _, accuracy = train_if_else(X, y, feature_col=i, feature_name=name)
    print(f"The model trained on {name} is {100 * accuracy:.0f}% "
          "accurate.")
```

The model trained on `switch0` is 50% accurate.

The model trained on `switch1` is 50% accurate.

Both models perform terribly! A single `if/else` statement is insufficient to capture the complexity of the problem. What should we do? One approach is to break the problem into parts by training two separate models: Model A considers only those scenarios in which `switch0` is off, and Model B considers all remaining scenarios in which `switch0` is on. Later, we'll combine Model A and Model B into a single coherent classifier.

Let's investigate the first scenario, where `switch0` is off. When it is off, `X[:, 0] == 0`. Hence, we start by isolating a training subset that satisfies this Boolean requirement. We store this training subset in variables `X_switch0_off` and `y_switch0_off`.

Listing 22.14 Isolating a training subset where `switch0` is off

```
is_off = X[:, 0] == 0
X_switch0_off = X[is_off]
y_switch0_off = y[is_off]
print(f"Feature matrix when switch0 is off:\n{X_switch0_off}")
print(f"\nClass labels when switch0 is off:\n{y_switch0_off}")
```

Feature matrix when `switch0` is off:
 $\begin{bmatrix} [0 & 0] \\ [0 & 1] \end{bmatrix}$

All elements of column 0
now equal 0.

Class labels when `switch0` is off:
 $[0 \ 1]$

In the training subset, `switch0` is always off. Hence, `X_switch0_off[:, 0]` always equals zero. This zero column is now redundant, and we can delete the useless column using NumPy's `np.delete` function.

Listing 22.15 Deleting a redundant feature column

```
X_switch0_off = np.delete(X_switch0_off, 0, axis=1) ←
print(X_switch0_off)
```

$\begin{bmatrix} [0] \\ [1] \end{bmatrix}$

The 0 element column
has been deleted.

Running `np.delete(X, r)` returns a copy of X with
row r deleted, and running `np.delete(X, c, axis=1)`
returns a copy of X with column c deleted. Here,
we delete the redundant column 0.

Next, we train an `if/else` model on the training subset. The model predicts bulb activation based on the `switch1` state. These predictions are valid only if `switch0` is off. We store the model in a `switch0_off_model` variable, and we store the model's accuracy in a corresponding `switch0_off_accuracy` variable.

Listing 22.16 Training a model when `switch0` is off

```
results = train_if_else(X_switch0_off, y_switch0_off,
                        feature_name='switch1')
switch0_off_model, off_accuracy = results
print("If switch 0 is off, then the following if/else model is "
      f"\n{100 * off_accuracy:.0f}% accurate.\n\n{switch0_off_model}")
```

If switch 0 is off, then the following if/else model is 100% accurate.

```
if switch1 == 0:
    prediction = 0
else:
    prediction = 1
```

If switch0 is off, then our trained if/else model can predict the bulb state with 100% accuracy. Now, let's train a corresponding model to cover all the cases where switch0 is on. We start by filtering our training data based on the condition $X[:, 0] == 1$.

Listing 22.17 Isolating a training subset where switch0 is on

```
def filter_X_y(X, y, feature_col=0, condition=0):
    inclusion_criteria = X[:, feature_col] == condition
    y_filtered = y[inclusion_criteria]
    X_filtered = X[inclusion_criteria]
    X_filtered = np.delete(X_filtered, feature_col, axis=1)
    return X_filtered, y_filtered
```

Filters the training data based on the feature in the feature_col column of matrix X. Returns a subset of the training data where the feature equals a specified condition value.

A Boolean array where the /th element is True if $X[i][feature_col]$ equals condition

The column feature_col becomes redundant since all the filtered values equal condition. Hence, this column is filtered from the training data.

Next, we train switch0_on_model using the filtered training set.

Listing 22.18 Training a model when switch0 is on

```
results = train_if_else(X_switch0_on, y_switch0_on,
                        feature_name='switch1')
switch0_on_model, on_accuracy = results
print("If switch 0 is on, then the following if/else model is "
      f"{100 * on_accuracy:.0f}% accurate.\n\n{switch0_on_model}")

If switch 0 is on, then the following if/else model is 100% accurate.

if switch1 == 0:
    prediction = 1
else:
    prediction = 0
```

If switch == 0, then switch0_off_model performs with 100% accuracy. In all other cases, the switch1_on_model performs with 100% accuracy. Together, the two models can easily be combined into a single nested if/else statement. Here, we define a combine_if_else function that merges two separate if/else statements; we then apply that function to our two models.

Listing 22.19 Combining separate if/else models

```

def combine_if_else(if_else_a, if_else_b, feature_name='feature'):
    return f"""
if {feature_name} == 0:
{add_indent(if_else_a)} ←
else:
{add_indent(if_else_b)}
""".strip()

def add_indent(if_else):
    return '\n'.join([4 * ' ' + line for line in if_else.split('\n')])

nested_model = combine_if_else(switch0_off_model, switch0_on_model,
                                feature_name='switch0')
print(nested_model)

if switch0 == 0:
    if switch1 == 0:
        prediction = 0
    else:
        prediction = 1
else:
    if switch1 == 0:
        prediction = 1
    else:
        prediction = 0

```

The standard four-space Python indent is added to each statement during nesting.

This helper function helps indent all statements during nesting.

Combines two if/else statements if_else_a and if_else_b into a single nested statement

We've reproduced the nested if/else model from listing 22.2. This model is 100% accurate. We can confirm by taking the weighted average of off_accuracy and on_accuracy. These accuracies correspond to the off/on states of switch0, so their weights should correspond to the off/on counts associated with switch0. The counts equal the lengths of the y_switch0_off and y_switch0_on arrays. Let's take the weighted average and confirm that the total accuracy equals 1.0.

Listing 22.20 Computing total nested accuracy

```

accuracies = [off_accuracy, on_accuracy]
weights = [y_switch0_off.size, y_switch0_on.size]
total_accuracy = np.average(accuracies, weights=weights)
print(f"Our total accuracy is {100 * total_accuracy:.0f}%")

```

Our total accuracy is 100%

We're able to generate a nested two-feature model in an automated manner. Our strategy hinges on the creation of separate training sets. That separation is determined by the on/off states of one of the features. This type of separation is called a *binary split*. Typically, we split training set (X , y) using two parameters:

- Feature i corresponding to column i of X . For example, switch0 in column 0 of X .
- Condition c , where $X[:, i] == c$ is True for some but not all of the data points. For example, condition 0 corresponding to the off state.

A split on feature i and condition c can be carried out as follows:

- 1 Obtain a training subset (X_a, y_a) in which $X_a[:, i] == c$.
- 2 Obtain a training subset (X_b, y_b) in which $X_b[:, i] != c$.
- 3 Delete column i from both X_a and X_b .
- 4 Return the separated subsets (X_a, y_a) and (X_b, y_b) .

NOTE These steps are not intended to run on a continuous feature. Later in this section, we discuss how to transform a continuous feature into a binary variable to carry out the split.

Let's define a `split` function to execute these steps. Then we'll incorporate this function into a systematic training pipeline.

Listing 22.21 Defining a binary split function

```
def split(X, y, feature_col=0, condition=0):
    has_condition = X[:, feature_col] == condition
    X_a, y_a = [e[has_condition] for e in [X, y]]
    X_b, y_b = [e[~has_condition] for e in [X, y]]
    X_a, X_b = [np.delete(e, feature_col, axis=1) for e in [X_a, X_b]]
    return [X_a, X_b, y_a, y_b]

X_a[:,feature_col]
In the second
training set,
never equals
condition.

X_a[ :, feature_col ] always
equals condition.

X_a, X_b, y_a, y_b = split(X, y)
assert np.array_equal(X_a, X_switch0_off)
assert np.array_equal(X_b, X_switch0_on)
```

The split creates two training sets (X_a, y_a) and (X_b, y_b) . In the first training set, $X_a[:, feature_col]$ always equals condition.

By splitting on `switch0`, we were able to train a nested model. Before that split, we first tried training simple `if/else` models. These models performed terribly—we had no choice but to split the training data. But trained nested models should still be compared with simpler models returned by `train_if_else`. If a simpler model shows comparable performance, it should be returned instead.

NOTE A nested two-feature model will never perform worse than a simple model based on a single feature. However, it is possible for the two models to perform equally well. Under these circumstances, it's best to follow Occam's razor: when two competing theories make exactly the same predictions, the simpler one is the better theory.

Let's formalize our process for training two-feature nested models. Given training set (X, y) , we carry out these steps:

- 1 Choose a feature i to split on. Initially, that feature is specified using a parameter. Later, we learn how to choose that feature in an automated manner.
- 2 Attempt to train a simple, single feature model on feature i . If that model performs with 100% accuracy, return it as our output.

Theoretically, we could train two single-feature models on columns 0 and 1 using `train_if_else`. Then we could compare all single-feature models

systematically. However, this approach won't scale when we increase the feature count from two to N .

- 3 Split on feature i using `split`. The function returns two training sets (X_a , y_a) and (X_b , y_b).
- 4 Train two simple models `if_else_a` and `if_else_b` using the training sets returned by `split`. The corresponding accuracies equal `accuracy_a` and `accuracy_b`.
- 5 Combine `if_else_a` and `if_else_b` into a nested `if/else` conditional model.
- 6 Compute the nested model's accuracy using the weighted mean of `accuracy_a` and `accuracy_b`. The weights equal `y_a.size` and `y_b.size`.
- 7 Return the nested model if it outperforms the simple model computed in step 2. Otherwise, return the simple model.

Let's define a `train_nested_if_else` function to execute these steps. The function returns the trained model and that model's accuracy.

Listing 22.22 Training a nested `if/else` model

Trains a nested `if/else` statement on the two-feature training set (X, y) and returns the written statement along with the corresponding accuracy. The statement is trained by splitting on the feature in $X[:, split_col]$. The feature names in the statement are stored in the `feature_names` array.

```
def train_nested_if_else(X, y, split_col=0,
                        feature_names=['feature1', 'feature1']): ←
    split_name = feature_names[split_col]
    simple_model, simple_accuracy = train_if_else(X, y, split_col,
                                                split_name) ←
    if simple_accuracy == 1.0:
        return (simple_model, simple_accuracy) ←
    X_a, X_b, y_a, y_b = split(X, y, feature_col=split_col)
    in_name = feature_names[1 - split_col]
    if_else_a, accuracy_a = train_if_else(X_a, y_a, feature_name=in_name) ←
    if_else_b, accuracy_b = train_if_else(X_b, y_b, feature_name=in_name) ←
    nested_model = combine_if_else(if_else_a, if_else_b, split_name)
    accuracies = [accuracy_a, accuracy_b]
    nested_accuracy = np.average(accuracies, weights=[y_a.size, y_b.size])
    if nested_accuracy > simple_accuracy:
        return (nested_model, nested_accuracy) ←
    return (simple_model, simple_accuracy) ←

    feature_names = ['switch0', 'switch1']
model, accuracy = train_nested_if_else(X, y, feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if switch0 == 0:
    if switch1 == 0:
        prediction = 0
```

The name of the feature located in the inner portion of the nested statement

The feature names in the statement are stored in the `feature_names` array.

Combines the simple models

Trains two simple models

```

        else:
            prediction = 1
    else:
        if switch1 == 0:
            prediction = 1
        else:
            prediction = 0

This statement is 100% accurate.

```

Our function trained a model that's 100% accurate. Given our current training set, that accuracy should hold even if we split on `switch1` instead of `switch0`. Let's verify.

Listing 22.23 Splitting on `switch1` instead of `switch0`

```

model, accuracy = train_nested_if_else(X, y, split_col=1,
                                       feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if switch1 == 0:
    if switch0 == 0:
        prediction = 0
    else:
        prediction = 1
else:
    if switch0 == 0:
        prediction = 1
    else:
        prediction = 0

```

```

This statement is 100% accurate.

```

Splitting on either feature yields the same result. This is true of our two-switch system, but it is not the case for many real-world training sets. It's common for one split to outperform another. In the next subsection, we explore how to prioritize the features during splitting.

22.1.2 Deciding which feature to split on

Suppose we wish to train an `if/else` model that predicts whether it's raining outside. The model returns 1 if it is raining and 0 otherwise. The model relies on the following two features:

- Is the current season autumn? Yes or No?

We'll assume that fall/autumn is the local rainy season and that the feature predicts rain 60% of the time.

- Is it currently wet outside? Yes or No?

Usually it's raining when it's wet. Occasionally, wetness is caused by a sprinkler system on a sunny day; and conditions might appear dry during a drizzly

morning in the forest, if the trees impede the raindrops. We'll assume this feature predicts rain 95% of the time.

Let's simulate the features and class labels through random sampling. We sample across 100 weather observations and store the outputs in training set (`X_rain`, `y_rain`).

Listing 22.24 Simulating a rainy-day training set

```
np.random.seed(1)
y_rain = np.random.binomial(1, 0.6, size=100) ← It rains 60% of the time.
→ is_wet = [e if np.random.binomial(1, 0.95) else 1 - e for e in y_rain]
is_fall = [e if np.random.binomial(1, 0.6) else 1 - e for e in y_rain] ←
X_rain = np.array([is_fall, is_wet]).T → 60% of the time, the
                                         state of autumn equals
                                         the state of rain.
```

95% of the time, the state of wetness equals the state of rain.

Now, let's train a model by splitting on the autumn feature.

Listing 22.25 Training a model with an autumn split

```
feature_names = ['is_autumn', 'is_wet']
model, accuracy = train_nested_if_else(X_rain, y_rain,
                                         feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if is_autumn == 0:
    if is_wet == 0:
        prediction = 0
    else:
        prediction = 1
else:
    if is_wet == 0:
        prediction = 0
    else:
        prediction = 1
```

This statement is 95% accurate.

We trained a nested model that is 95% accurate. What if we split on the wetness feature instead?

Listing 22.26 Training a model with a wetness split

```
model, accuracy = train_nested_if_else(X_rain, y_rain, split_col=1,
                                         feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if is_wet == 0:
    prediction = 0
```

```

else:
    prediction = 1

This statement is 95% accurate.

```

Splitting on wetness yields a simpler (and hence better) model while retaining the previously seen accuracy. Not all splits are equal: splitting on some features leads to preferable results. How should we select the best feature for the split? Naively, we could iterate over all features in `X`, train models by splitting on each feature, and return the simplest model with the highest accuracy. This brute-force approach works when `X.size[1] == 2`, but it won't scale as the feature count increases. Our goal is to develop a technique that can scale to thousands of features, so we need an alternate approach.

One solution requires us to examine the distribution of classes in the training set. Currently, our `y_rain` array contains two binary classes: 0 and 1. The label 1 corresponds to a rainy observation. Therefore, the array sum equals the number of rainy observations. Meanwhile, the array size equals the total number of observations, so `y_rain.sum() / y_rain.size` equals the overall probability of rain. Let's print that probability.

Listing 22.27 Computing the probability of rain

```

prob_rain = y_rain.sum() / y_rain.size
print(f"It rains in {100 * prob_rain:.0f}% of our observations.")

It rains in 61% of our observations

```

It rains in 61% of total observations. How is that probability altered when we split on autumn? Well, the split returns two training sets with two class-label arrays. We'll call these arrays `y_fall_a` and `y_fall_b`. Dividing `y_fall_b.sum()` by the array size returns the likelihood of rain during autumn. Let's print that likelihood and the probability of rain during the other seasons.

Listing 22.28 Computing the probability of rain based on the season

```

y_fall_a, y_fall_b = split(X_rain, y_rain, feature_col=0)[-2:]
for i, y_fall in enumerate([y_fall_a, y_fall_b]):
    prob_rain = y_fall.sum() / y_fall.size
    state = ['not autumn', 'autumn'][i]
    print(f"It rains {100 * prob_rain:.0f}% of the time when it is "
          f"{state}")

It rains 55% of the time when it is not autumn
It rains 66% of the time when it is autumn

```

As expected, we're more likely to see rain during the autumn season, but that likelihood difference is not very large. It rains 66% of the time during autumn, and 55% of the time during the other seasons. It's worth noting that these two probabilities are close to our overall rain likelihood of 61%. If we know that it is autumn, we are slightly

more confident about rain. Still, our confidence increase is not very substantial relative to the original training set, so the split on autumn is not very informative. What if instead we split on wetness? Let's check if that split shifts the observed probabilities.

Listing 22.29 Computing the probability of rain based on wetness

```
y_wet_a, y_wet_b = split(X_rain, y_rain, feature_col=1)[-2:]
for i, y_wet in enumerate([y_wet_a, y_wet_b]):
    prob_rain = y_wet.sum() / y_wet.size
    state = ['not wet', 'wet'][i]
    print(f"It rains {100 * prob_rain:.0f}% of the time when it is "
          f"{state}")
```

```
It rains 10% of the time when it is not wet
It rains 98% of the time when it is wet
```

If we know it's wet outside, then we have nearly perfect confidence in rain. Whenever it is dry, the chance of rain remains at 10%. This percentage is low but very significant to our classifier. We know that dry conditions predict a lack of rain with 90% accuracy.

Intuitively, wetness is a more informative feature than autumn. How should we quantify our intuition? Well, the wetness split returns two class-label arrays whose corresponding rain probabilities are either very low or very high. These extreme probabilities are an indication of class imbalance. As we learned in section 20, an imbalanced dataset has far more of some Class A relative to another Class B. This makes it easier for a model to isolate Class A in the data. By comparison, the autumn split returns two arrays whose likelihoods are in the moderate range of 55 to 66%. The classes in `y_fall_a` and `y_fall_b` are more balanced. Thus, it's not as easy to tell the rain/not-rain classes apart.

When choosing between two splits, we should select the split that yields more imbalanced class labels. Let's figure out how to quantify class imbalance. Generally, imbalance is associated with the shape of the class probability distribution. We can treat this distribution as a vector `v`, where `v[i]` equals the probability of observing Class `i`. A higher value of `v.max()` indicates a greater class imbalance. In our two-class dataset, we can compute `v` as `[1 - prob_rain, prob_rain]`, where `prob_rain` is the probability of rain. This two-element vector can be visualized as a line segment in 2D space, per our discussion in section 12 (figure 22.2).

Such visualizations can prove insightful. We'll now do the following:

- 1 Compute the class distribution vectors for the autumn split using arrays `y_fall_a` and `y_fall_b`.
- 2 Compute the class distribution vectors for the wetness split using arrays `y_wet_a` and `y_wet_b`.
- 3 Visualize all four arrays as line segments in 2D space.

That visualization will reveal how to effectively measure class imbalance (figure 22.3).

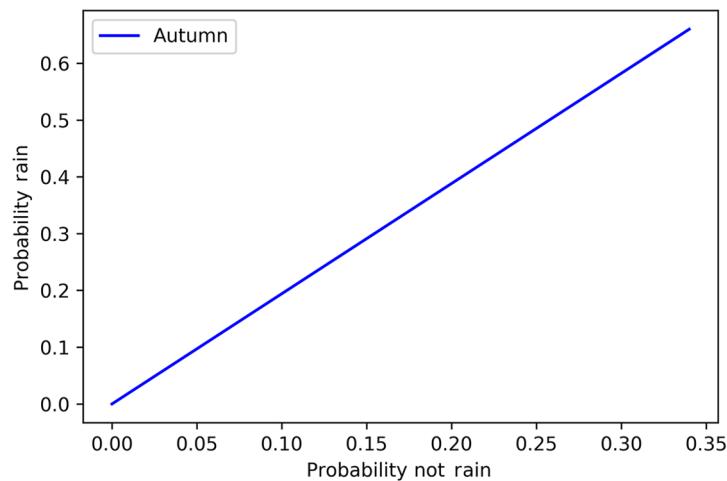


Figure 22.2 The probability distribution over class labels in `y_fall_a` (is autumn) visualized as a 2D line segment. The y-axis represents the probability of rain (0.66), and the x-axis represents the probability of no rain ($1 - 0.66 = 0.36$).

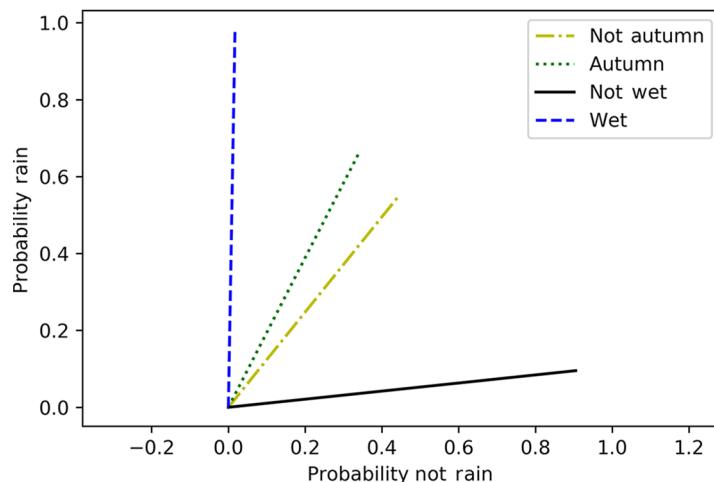


Figure 22.3 A plot of the four vector distributions across each feature split. The wetness vectors are much more imbalanced and thus are positioned closer to the axes. More importantly, the wetness vectors appear longer than the autumn vectors.

Listing 22.30 Plotting the class distribution vectors

```

def get_class_distribution(y):    ← Returns the probability distribution across
    prob_rain = y.sum() / y.size   ← class labels in a binary, two-class system. The
    return np.array([1 - prob_rain, prob_rain])   ← distribution can be treated as a 2D vector.

def plot_vector(v, label, linestyle='--', color='b'):    ← Plots a 2D vector v as a line
    plt.plot([0, v[0]], [0, v[1]], label=label,           ← segment that stretches
              linestyle=linestyle, c=color)               ← from the origin to v

classes = [y_fall_a, y_fall_b, y_wet_a, y_wet_b]    ← Iterates over the four
distributions = [get_class_distribution(y) for y in classes]   ← unique distribution
labels = ['Not Autumn', 'Autumn', 'Not Wet', 'Wet']   ← vectors that result
colors = ['y', 'g', 'k', 'b']   ← from every possible
linestyles = [':', '-.', '--', '-']   ← split and then plots
for tup in zip(distributions, labels, colors, linestyles):   ← these four vectors
    vector, label, color, linestyle = tup
    plot_vector(vector, label, linestyle=linestyle, color=color)

plt.legend()
plt.xlabel('Probability Not Rain')
plt.ylabel('Probability Rain')
plt.axis('equal')
plt.show()

```

In our plot, the two imbalanced wetness vectors skew heavily toward the x- and y-axes. Meanwhile, the two balanced autumn vectors are approximately equidistant from both axes. However, what really stands out is not vector direction but vector length: the balanced autumn vectors are much shorter than the vectors associated with wetness. This isn't a coincidence. Imbalanced distributions are proven to have greater vector magnitudes. Also, as we showed in section 13, the magnitude is equal to the square root of $v @ v$. Hence, the dot product of a distribution vector with itself is greater if that vector is more imbalanced!

Let's demonstrate this property for every 2D vector $v = [1 - p, p]$, where p is the probability of rain. Listing 22.31 plots the magnitude of v across rain likelihoods ranging from 0 to 1. We also plot the square of the magnitude, which is equal to $v @ v$. The plotted values should be maximized when p is very low or very high and minimized when v is perfectly balanced at $p = 0.5$ (figure 22.4).

Listing 22.31 Plotting the distribution vector magnitudes

```

Computes the squares    The probability of rain
of vector magnitudes as    ranges from 0 to 1.0
a simple dot product    (inclusive).    ← Vectors represent all
prob_rain = np.arange(0, 1.001, 0.01)   ← possible two-class
vectors = [np.array([1 - p, p]) for p in prob_rain]   ← distributions, where
magnitudes = [np.linalg.norm(v) for v in vectors]   ← the classes are rain
square_magnitudes = [v @ v for v in vectors]   ← and not rain.
plt.plot(prob_rain, magnitudes, label='Magnitude')   ← Vector magnitudes
                                                               ← computed using NumPy

```

```

plt.plot(prob_rain, square_magnitudes, label='Squared Magnitude',
         linestyle='--')
plt.xlabel('Probability of Rain')
plt.axvline(0.5, color='k', label='Perfect Balance', linestyle=':')
plt.legend()
plt.show()

```

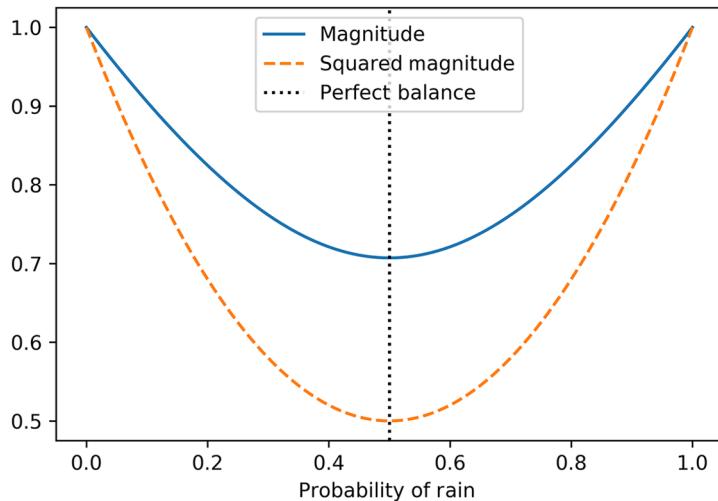


Figure 22.4 A plot of distribution vector magnitudes and squared magnitudes across each distribution vector $[1 - p, p]$. The plotted values are minimized when the vector is perfectly balanced at $p = 0.5$.

The squared magnitude is maximized at 1.0 when v is fully imbalanced at $p = 0.0$ and $p = 1.0$. It's also minimized at 0.5 when v is balanced. Thus, $v @ v$ serves as an excellent metric for class-label imbalance, but data scientists prefer the slightly different metric of $1 - v @ v$. This metric, called the *Gini impurity*, essentially flips the plotted curve: it is minimized at 0 and maximized at 0.5. Let's confirm by plotting the Gini impurity across all values of p (figure 22.5).

NOTE The Gini impurity has a concrete interpretation in probability theory. Suppose that for any data point, we randomly assign a class of i with probability $v[i]$, where v is the vectorized distribution. The probability of choosing a point belonging to Class i is also equal to $v[i]$. Hence, the probability of choosing a point belonging to Class i and correctly labeling that point is equal to $v[i] * v[i]$. It follows that the probability of correctly labeling any point is equal to $\sum(v[i] * v[i])$ for i in $\text{range}(\text{len}(v))$. This simplifies to $v @ v$. Thus, $1 - v @ v$ equals the probability of incorrectly labeling our data. The Gini impurity is equal to the probability of error, which decreases as the data grows more imbalanced.

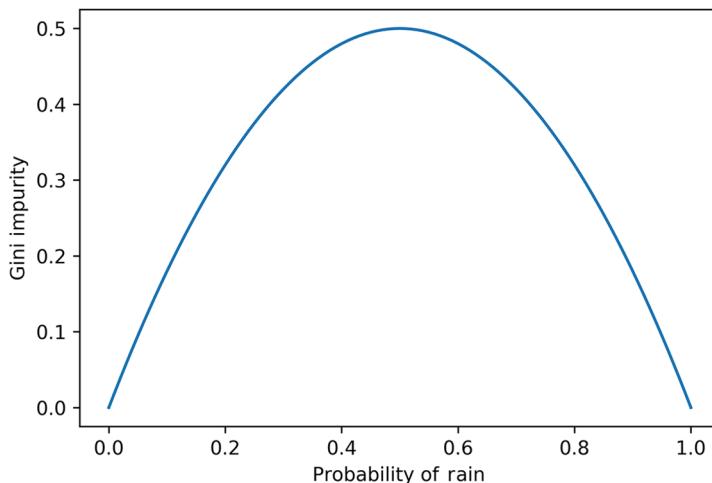


Figure 22.5 A plot of Gini impurities across each distribution vector $[1 - p, p]$. The Gini impurity is maximized when the vector is perfectly balanced at $p = 0.5$.

Listing 22.32 Plotting the Gini impurity

```
gini_impurities = [1 - (v @ v) for v in vectors]
plt.plot(prob_rain, gini_impurities)
plt.xlabel('Probability of Rain')
plt.ylabel('Gini Impurity')
plt.show()
```

Gini impurity is a standard measure of class imbalance. Highly imbalanced datasets are considered more “pure” since labels lean heavily toward one class over any other. When training a nested model, we should split on the feature that minimizes overall impurity. For any split with class labels y_a and y_b , we can compute the impurity like this:

- 1 Compute the impurity of y_a .
 - It’s equal to $1 - v_a @ v_a$, where v_a is the class distribution over y_a .
- 2 Next, we compute the impurity of y_b .
 - It’s equal to $1 - v_b @ v_b$, where v_b is the class distribution over y_b .
- 3 Finally, we take the weighted mean of the two impurities.
 - The weights will equal $y_a.size$ and $y_b.size$, just like in our total accuracy computation.

Let’s compute the impurities associated with autumn and wetness.

Listing 22.33 Computing each feature's Gini impurity

```

The class distribution
vector of y_b
def compute_impurity(y_a, y_b):
    v_a = get_class_distribution(y_a)
    v_b = get_class_distribution(y_b)
    impurities = [1 - v @ v for v in [v_a, v_b]]
    weights = [y_a.size, y_b.size]
    return np.average(impurities, weights=weights)

fall_impurity = compute_impurity(y_fall_a, y_fall_b)
wet_impurity = compute_impurity(y_wet_a, y_wet_b)
print(f"When we split on Autumn, the Impurity is {fall_impurity:.2f}.")
print(f"When we split on Wetness, the Impurity is {wet_impurity:.2f}.")

```

When we split on Autumn, the Impurity is 0.45.
When we split on Wetness, the Impurity is 0.04.

As expected, the impurity is minimized when we split on wetness. This split leads to more imbalanced training data, which simplifies the training of the classifier. Going forward, we will split on features whose Gini impurity is minimized. With this in mind, let's define a `sort_feature_indices` function. The function takes as input a training set (X , y) and returns a list of sorted feature indices based on the impurity associated with each feature split.

Listing 22.34 Sorting features by Gini impurity

```

def sort_feature_indices(X, y):
    feature_indices = range(X.shape[1])
    impurities = []

    for i in feature_indices:
        y_a, y_b = split(X, y, feature_col=i)[-2:]
        impurities.append(compute_impurity(y_a, y_b))

    return sorted(feature_indices, key=lambda i: impurities[i])

indices = sort_feature_indices(X_rain, y_rain)
top_feature = feature_names[indices[0]]
print(f"The feature with the minimal impurity is: '{top_feature}'")

```

The feature with the minimal impurity is: 'is_wet'

Returns the sorted column indices of X . The first column corresponds to the smallest impurity.

The `sort_feature_indices` function will prove invaluable as we train nested `if/else` models with more than two features.

22.1.3 Training if/else models with more than two features

Training a model to predict the current weather is a relatively trivial task. We'll now train a more complicated model that predicts whether it will rain tomorrow. The model relies on the following three features:

- Has it rained at any point today?
If it rained today, then it is very likely to rain tomorrow.
- Is today a cloudy day? Yes or No?
It's more likely to rain on a cloudy day. This increases the chance that it will rain tomorrow.
- Is today an autumn day? Yes or No?

We assume that it's both rainier and cloudier in autumn.

We further assume a complex but realistic interrelationship between the three features, to make the problem much more interesting:

- There's a 25% chance that today is autumn.
- During autumn, it is cloudy 70% of the time. Otherwise, it's cloudy 30% of the time.
- If today is cloudy, there's a 40% of rain at some point in the day. Otherwise, the chance of rain is 5%.
- If it rains today, there's a 50% chance it will rain tomorrow.
- If today is a dry and sunny autumn day, there's a 15% chance it will rain tomorrow. Otherwise, on dry and sunny spring, summer, and winter days, the chance of rain tomorrow falls to 5%.

The following code simulates a training set (`X_rain`, `y_rain`) based on the probabilistic relationships between the features.

Listing 22.35 Simulating a three-feature training set

```
During autumn, it's
cloudy 70% of the time.
Otherwise, it's cloudy
30% of the time.

np.random.seed(0)
def simulate_weather():
    is_fall = np.random.binomial(1, 0.25)
    is_cloudy = np.random.binomial(1, [0.3, 0.7][is_fall])
    rained_today = np.random.binomial(1, [0.05, 0.4][is_cloudy])
    if rained_today:
        rains_tomorrow = np.random.binomial(1, 0.5)
    else:
        rains_tomorrow = np.random.binomial(1, [0.05, 0.15][is_fall])

    features = [rained_today, is_cloudy, is_fall]
    return features, rains_tomorrow
```

```

X_rain, y_rain = [], []
for _ in range(1000):
    features, rains_tomorrow = simulate_weather() ←
        X_rain.append(features)
        y_rain.append(rains_tomorrow)

X_rain, y_rain = np.array(X_rain), np.array(y_rain)

```

Simulates a dataset with 1,000 training examples

The columns in `X_rain` correspond to features '`is_fall`', '`is_cloudy`', and '`rained_today`'. We can sort these features by Gini impurity to measure how well they split the data.

Listing 22.36 Sorting three features by Gini impurity

```

feature_names = ['rained_today', 'is_cloudy', 'is_fall']
indices = sort_feature_indices(X_rain, y_rain)
print(f"Features sorted by Gini Impurity:")
print([feature_names[i] for i in indices])

Features sorted by Gini Impurity:
['is_fall', 'is_cloudy', 'rained_today']

```

Splitting on the autumn feature yields the lowest Gini impurity, and cloudiness ranks second. The rainy feature has the highest Gini impurity: it yields the most balanced datasets and is not a good candidate for splitting.

NOTE The high Gini impurity of the rainy feature may seem surprising. After all, if it rained today, we know it's much more likely to rain tomorrow. Hence, when `X_rain[:, 0] == 1`, the Gini impurity is low. But on a dry day, we have little indication of tomorrow's weather; so when `X_rain[:, 0] == 0`, the Gini impurity is high. There are more dry days than there are rainy days during the year, so the mean Gini impurity is high. In contrast, the autumn feature is much more informative. It provides us with insight into tomorrow's weather on both autumn and non-autumn days.

Given our ranked feature list, how should we train our model? After all, `trained_nested_if_else` is intended to process two features, not three. One intuitive solution is to train the model on just the two top-ranked features. These features lead to a greater training set imbalance, making it easier to decipher between rainy and non-rainy class labels.

Here, we train a two-feature model on just the autumn and cloudiness features. We also set the split column to autumn since autumn has the lowest Gini impurity.

Listing 22.37 Training a model on the two best features

```

skip_index = indices[-1] ← Ignores the final feature, which
                           has the worst Gini impurity
X_subset = np.delete(X_rain, skip_index, axis=1) ←
name_subset = np.delete(feature_names, skip_index) ← The feature subset with
                                                 the two best features

```

```

split_col = indices[0] if indices[0] < skip_index else indices[0] - 1 ←
model, accuracy = train_nested_if_else(X_subset, y_rain,
                                         split_col=split_col,
                                         feature_names=name_subset) ←
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.") | Trains a
                                                               nested model
                                                               on the two
                                                               best features

prediction = 0 | Adjusts the column of the best
This statement is 74% accurate. | split feature relative to the filtered
                                                               index of the worst, deleted feature

```

Our trained model is a frivolously simple. It always predicts no rain, no matter what! This trivial model is only 74% accurate—that accuracy is not disastrous, but we can definitely do better. Ignoring the rainy feature has limited our predictive capacity. We must incorporate all three features to raise the accuracy score. We can incorporate all three features like this:

- 1 Split on the feature with the lowest Gini impurity. This, of course, is autumn.
- 2 Train two nested models using the `train_nested_if_else` function. Model A will consider only those scenarios in which the season is not autumn, and Model B will consider all remaining scenarios in which the season is autumn.
- 3 Combine Model A and Model B into a single coherent classifier.

NOTE These steps are nearly identical to the logic behind the `nested_if_else` function. The main difference is that now we’re expanding that logic to more than two features.

Let’s start by splitting on the autumn feature, whose index is stored in `indices[0]`.

Listing 22.38 Splitting on the feature with the lowest impurity

```
X_a, X_b, y_a, y_b = split(X_rain, y_rain, feature_col=indices[0])
```

Next, let’s train a nested model on `(X_a, y_a)`. This training set contains all of our non-autumn observations.

Listing 22.39 Training a model when the season is not autumn

```

name_subset = np.delete(feature_names, indices[0]) ← Splits on the feature in
split_col = sort_feature_indices(X_a, y_a)[0] ← X_a that yields the best
model_a, accuracy_a = train_nested_if_else(X_a, y_a, ← (lowest) Gini impurity
                                         split_col=split_col,
                                         feature_names=name_subset) ←

print("If it is not autumn, then the following nested model is "
      f"{100 * accuracy_a:.0f}% accurate.\n\n{model_a}") | Trains a nested two-feature
                                                               model on (X_a, y_a)

If it is not autumn, then the following nested model is 88% accurate.

if is_cloudy == 0:
    prediction = 0

```

```

else:
    if rained_today == 0:
        prediction = 0
    else:
        prediction = 1

```

Our trained `model_a` is highly accurate. Now we will train a second `model_b` based on the autumn observations stored in `(X_b, y_b)`.

Listing 22.40 Training a model when the season is autumn

```

split_col = sort_feature_indices(X_b, y_b)[0]           ← Splits on the feature in
model_b, accuracy_b = train_nested_if_else(X_b, y_b,      X_b that yields the best
                                             split_col=split_col,       (lowest) Gini impurity
                                             feature_names=name_subset) ←
print("If it is autumn, then the following nested model is "
      f"{100 * accuracy_b:.0f}% accurate.\n\n{model_b}")

If it is autumn, then the following nested model is 79% accurate.          Trains a
                                                                           nested two-
                                                                           feature model
                                                                           on (X_b, y_b)

if is_cloudy == 0:
    prediction = 0
else:
    if rained_today == 0:
        prediction = 0
    else:
        prediction = 1

```

When it is autumn, `model_b` performs with 79% accuracy. Otherwise, `model_a` performs with 88% accuracy. Let's combine these models into a single nested statement. We use our `combine_if_else` function, which we previously defined for this exact purpose. We also compute the total accuracy, which equals the weighted mean of `accuracy_a` and `accuracy_b`.

Listing 22.41 Combining the models into a nested statement

```

nested_model = combine_if_else(model_a, model_b,
                                 feature_names[indices[0]])
print(nested_model)
accuracies = [accuracy_a, accuracy_b]
accuracy = np.average(accuracies, weights=[y_a.size, y_b.size])
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if is_fall == 0:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1

```

```

else:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1

```

This statement is 85% accurate.

We were able to generate a nested 3-feature model. The process was very similar to how we trained a nested 2-feature model. In this manner, we can extend our logic to train a 4-feature model, or a 10-feature model, or a 100-feature model. In fact, our logic can generalize to training any nested N -feature model. Suppose we're given a training set (X, y) , where X contains N columns. We should be able to easily train a model by executing the following steps:

- 1 If N equals 1, return the simple, non-nested `train_if_else(X, y)` output. Otherwise, go to the next step.
- 2 Sort our N features based on the Gini impurity, from lowest to highest.
- 3 Attempt to train a simpler, $N - 1$ feature model (using the top-ranked features from step 2). If that model performs with 100% accuracy, return it as our output. Otherwise, go to the next step.
- 4 Split on the feature with the smallest Gini impurity. That split returns two training sets (X_a, y_a) and (X_b, y_b) . Each training set contains $N - 1$ features.
- 5 Train two $N - 1$ feature models: `model_a` and `model_b`, using the training sets from the previous step. The corresponding accuracies equal `accuracy_a` and `accuracy_b`.
- 6 Combine `model_a` and `model_b` into a nested `if/else` conditional model.
- 7 Compute the nested model's accuracy using the weighted mean of `accuracy_a` and `accuracy_b`. The weights equal `y_a.size` and `y_b.size`.
- 8 Return the nested model if it outperforms the simpler model computed in step 3. Otherwise, return the simpler model.

Here, we define a recursive `train` function that carries out these steps.

Listing 22.42 Training a nested model with N features

Trains a nested `if/else` statement on the N feature training set (X, y) and returns the written statement along with the corresponding accuracy. The feature names in the statement are stored in the `feature_names` array.

```

def train(X, y, feature_names):
    if X.shape[1] == 1:
        return train_if_else(X, y, feature_name=feature_names[0])

    indices = sort_feature_indices(X, y)      ← Sorts the feature
    X_subset = np.delete(X, indices[-1], axis=1)  indices by Gini impurity

```

```

name_subset = np.delete(feature_names, indices[-1])
simple_model, simple_accuracy = train(X_subset, y, name_subset) ←
if simple_accuracy == 1.0:
    return (simple_model, simple_accuracy) ←
    Tries to train a simpler
    N - 1 feature model to see if it
    performs with 100% accuracy

Splits on
the feature
with the
lowest Gini
impurity

→ split_col = indices[0]
name_subset = np.delete(feature_names, split_col)
X_a, X_b, y_a, y_b = split(X, y, feature_col=split_col)
model_a, accuracy_a = train(X_a, y_a, name_subset) ←
model_b, accuracy_b = train(X_b, y_b, name_subset)
accuracies = [accuracy_a, accuracy_b] ←
Trains two simpler
N - 1 feature models
on the two training sets
returned after the split

Combines
the simpler
models
→ nested_model = combine_if_else(model_a, model_b, feature_names[split_col])
if total_accuracy > simple_accuracy:
    return (nested_model, total_accuracy)

return (simple_model, simple_accuracy)

```

model, accuracy = train(X_rain, y_rain, feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

```

if is_fall == 0:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1
else:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1

```

This statement is 85% accurate.

The branching `if/else` statements in our trained output resemble the branches of a tree. We can make the resemblance more explicit by visualizing the output as a *decision tree diagram*. Decision trees are special network structures used to symbolize `if/else` decisions. Features are nodes in the network, and conditions are edges. The `if` condition branches to the right of the feature nodes, and the `else` condition branches to the left. Figure 22.6 represents our rain-prediction model using a decision tree diagram.

Any nested `if/else` statement can be visualized as a decision tree, so trained `if/else` conditional classifiers are referred to as *decision tree classifiers*. Trained decision tree classifiers have been in common use since the 1980s. Numerous strategies exist for training these classifiers effectively, all of which have the following properties in common:

- The N feature training problem is simplified down to multiple $N - 1$ feature subproblems by splitting on one of the features.
- The split is carried out by choosing the feature that yields the highest class imbalance. This is commonly done using the Gini impurity, although alternative metrics do exist.
- Caution is taken to avoid needlessly complex if/else statements if simpler statements work equally well. This process is called *pruning*, since excessive if/else branches are pruned out.

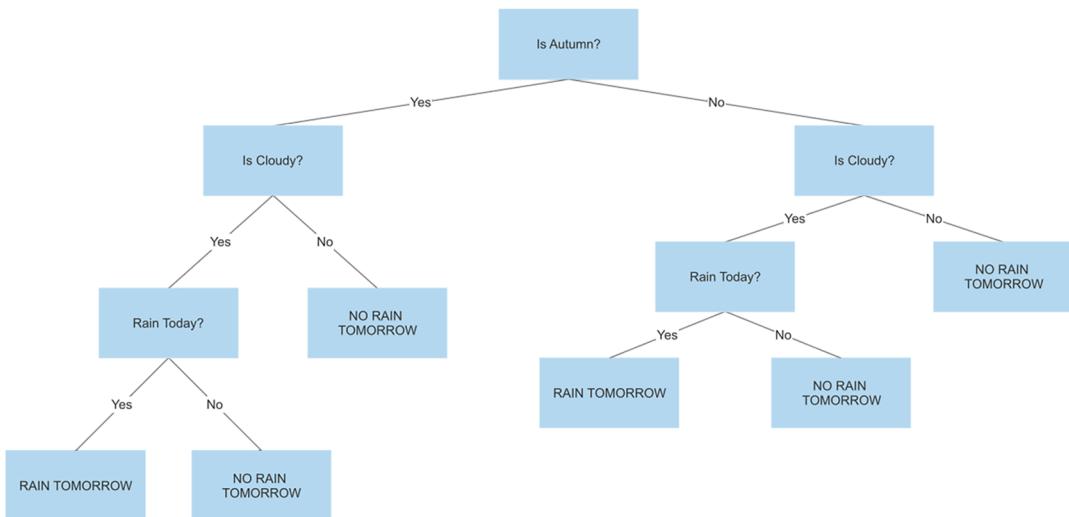


Figure 22.6 Visualizing the rain-prediction model using a decision tree diagram. The diagram is a network. The network nodes represent the model's features, such as “Is Autumn?”. The edges represent the conditional if/else statements. For instance, if it's autumn, the diagram's Yes edge branches left; otherwise, the No edge branches right.

Scikit-learn includes a highly optimized decision tree implementation. We explore it in the next subsection.

22.2 Training decision tree classifiers using scikit-learn

In scikit-learn, decision tree classification is carried out by the `DecisionTreeClassifier` class. Let's import that class from `sklearn.tree`.

Listing 22.43 Importing scikit-learn's `DecisionTreeClassifier` class

```
from sklearn.tree import DecisionTreeClassifier
```

Next, we initialize the class as `clf`. Then we train `clf` on the two-switch system introduced at the beginning of the section. That training set is stored in parameters `(X, y)`.

Listing 22.44 Initializing and training a decision tree classifier

```
clf = DecisionTreeClassifier()
clf.fit(X, y)
```

We can visualize the trained classifier using a decision tree diagram. Scikit-learn includes a `plot_tree` function, which uses Matplotlib to carry out that visualization. Calling `plot_tree(clf)` plots the trained decision tree diagram. The feature names and class names in that plot can be controlled using the `feature_names` and `class_names` parameters.

Let's import `plot_tree` from `sklearn.tree` and visualize `clf` (figure 22.7). In the plot, the feature names equal `Switch0` and `Switch1`, and the class labels equal the two bulb states: `Off` and `On`.

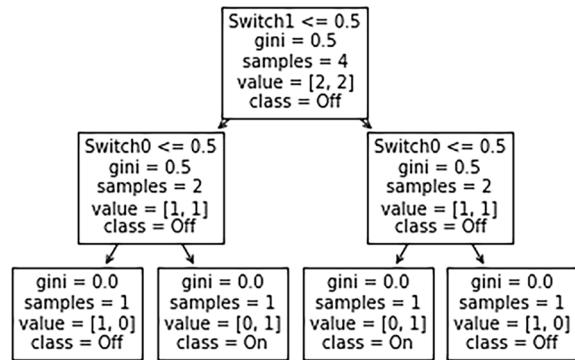


Figure 22.7 A plot of the two-switch system's decision tree diagram. Each top node contains a feature name along with additional statistics such as the Gini impurity and the dominant class. The bottom nodes contain the final predicted bulb classifications.

Listing 22.45 Displaying a trained decision tree classifier

```
from sklearn.tree import plot_tree
feature_names = ['Switch0', 'Switch1']
class_names = ['Off', 'On']
plot_tree(clf, feature_names=feature_names, class_names=class_names)
plt.show()
```

The visualized diagram tracks the class distribution at each conditional position in the tree. It also tracks the associated Gini impurity, as well as the dominant class. Such visualizations can be useful, but these tree plots can get unwieldy when the total feature count is large. That is why scikit-learn provides an alternative visualization function: `export_text` allows us to display the tree using a simplified text-based diagram. Calling `export_text(clf)` returns a string. Printing that string reveals a tree composed of | and - characters. The feature names in that text tree can be specified with

the `feature_names` parameter; but due to the limited nature of the output, we can't print the class names. Let's import `export_text` from `sklearn.tree` and then visualize our tree as a simple string.

Listing 22.46 Displaying a decision tree classifier as a string

```
from sklearn.tree import export_text
text_tree = export_text(clf, feature_names=feature_names)
print(text_tree)

|--- Switch0 <= 0.50
|   |--- Switch1 <= 0.50
|   |   |--- class: 0
|   |--- Switch1 >  0.50
|   |   |--- class: 1
|--- Switch0 >  0.50
|   |--- Switch1 <= 0.50
|   |   |--- class: 1
|   |--- Switch1 >  0.50
|       |--- class: 0
```

In the text, we clearly see the branching logic. Initially, the data is split using `Switch0`. The branch selection depends on whether `Switch0 <= 0.50`. Of course, because `Switch0` is either 0 or 1, that logic is identical to `Switch0 == 0`. Why does the tree use an inequality when the simple `Switch0 == 0` statement should suffice? The answer has to do with how `DecisionTreeClassifier` handles continuous features. Thus far, all of our features have been Booleans; but in most real-world problems, the features are numeric. Fortunately, any numerical feature can be transformed into a Boolean feature. We simply need to run `feature >= thresh`, where `thresh` is some numeric threshold. In scikit-learn, decision trees scan for this threshold automatically.

How should we select the optimal threshold for splitting a numeric feature? It's easy; we just need to choose the threshold that minimizes the Gini impurity. Suppose that we're examining a dataset that's driven by a single numeric feature. In that data, the class always equals 0 when the feature is less than 0.7, or 1 otherwise. Hence, `y = (v >= 0.7).astype(int)`, where `v` is the feature vector. By applying a threshold of 0.7, we can perfectly separate our class labels. Splitting on that threshold leads to a Gini impurity of 0.0, so we can isolate the threshold by computing the Gini impurity across a range of possible threshold values. Then we can select the value at which the impurity is minimized. Listing 22.47 samples a feature vector from a normal distribution, sets `y` to equal `(feature >= 0.7).astype(int)`, computes the impurity across a range of thresholds, and plots the results (figure 22.8). The minimal impurity appears at the threshold of 0.7.

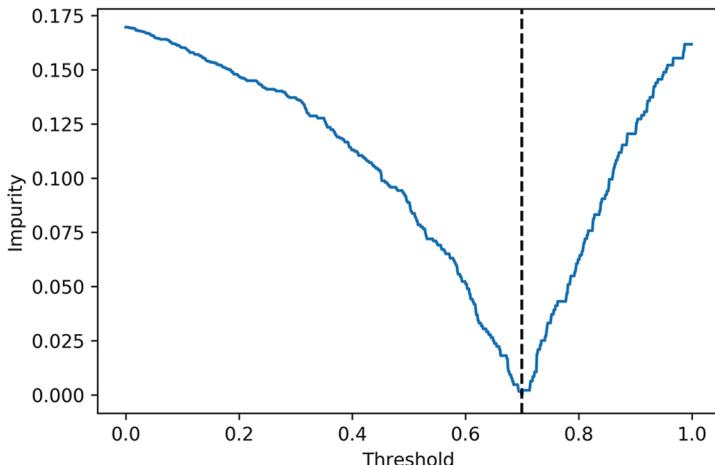


Figure 22.8 A plot of Gini impurities across each possible threshold of a feature. The Gini impurity is minimized at a threshold of 0.7. Thus, we can convert the numeric feature f into a binary feature $f \geq 0.7$.

Listing 22.47 Choosing a threshold by minimizing the Gini impurity

```

np.random.seed(1)
feature = np.random.normal(size=1000)
y = (feature >= 0.7).astype(int)
thresholds = np.arange(0.0, 1, 0.001)
gini_impurities = []
for thresh in thresholds:
    y_a = y[feature <= thresh]
    y_b = y[feature >= thresh]
    impurity = compute_impurity(y_a, y_b)
    gini_impurities.append(impurity)

best_thresh = thresholds[np.argmin(gini_impurities)]
print(f"impurity is minimized at a threshold of {best_thresh:.02f}")

plt.plot(thresholds, gini_impurities)
plt.axvline(best_thresh, c='k', linestyle='--')
plt.xlabel('Threshold')
plt.ylabel('impurity')
plt.show()

```

Randomly samples a numeric feature from a normal distribution

Class labels are 0 when the feature falls below a threshold of 0.7 and 1 otherwise.

Iterates over thresholds ranging from 0 to 1.0

At every threshold, we do a split and calculate the resulting Gini impurity.

Chooses the threshold at which the Gini impurity is minimized. That threshold should be equal to 0.7.

Impurity is minimized at a threshold of 0.70

In this manner, scikit-learn obtains inequality thresholds for all features used to train `DecisionTreeClassifier`, so the classifier can derive conditional logic from numeric data. Let's now train `clf` on the numeric wine data introduced in the previous section. After training, we visualize the tree.

NOTE As a reminder, the wine dataset contains three classes of wine. Thus far, we've only trained decision trees on two-class systems. However, our branching if/else logic can easily be extended to predict more than two classes. Consider, for instance, the statement `0 if x == 0 else 1 if y == 0 else 2.` The statement returns 0 if `x == 0`. Otherwise, the statement returns 1 if `y == 0` and 2 if `y != 0`. It's straightforward to incorporate this added conditional logic into our classifier.

Listing 22.48 Training a decision tree on numeric data

```

np.random.seed(0)
from sklearn.datasets import load_wine
X, y = load_wine(return_X_y=True)
clf.fit(X, y)

feature_names = load_wine().feature_names
text_tree = export_text(clf, feature_names=feature_names)
print(text_tree)

--- proline <= 755.00
|--- od280/od315_of_diluted_wines <= 2.11
|   |--- hue <= 0.94
|   |   |--- flavanoids <= 1.58
|   |   |   |--- class: 2
|   |   |   |--- flavanoids >  1.58
|   |   |   |--- class: 1
|   |--- hue >  0.94
|   |   |--- color_intensity <= 5.82
|   |   |   |--- class: 1
|   |   |--- color_intensity >  5.82
|   |   |   |--- class: 2
|--- od280/od315_of_diluted_wines >  2.11
|--- flavanoids <= 0.80
|   |--- class: 2
|--- flavanoids >  0.80
|   |--- alcohol <= 13.17
|   |   |--- class: 1
|   |--- alcohol >  13.17
|   |   |--- color_intensity <= 4.06
|   |   |   |--- class: 1
|   |   |   |--- color_intensity >  4.06
|   |   |   |--- class: 0
--- proline > 755.00
|--- flavanoids <= 2.17
|   |--- malic_acid <= 2.08
|   |   |--- class: 1
|   |--- malic_acid >  2.08
|   |   |--- class: 2
|--- flavanoids >  2.17
|   |--- magnesium <= 135.50
|   |   |--- class: 0
|   |--- magnesium >  135.50
|   |   |--- class: 1

```

The printed tree is larger than any tree or conditional statement seen thus far. The tree is large because it is deep. In machine learning, tree depth equals the number of nested if/else statements required to capture the logic within the tree. For instance, our single switch example required a single if/else statement. It therefore had a depth of 1. Meanwhile, our two-switch system had a depth of 2. Also, our three-feature weather predictor had a depth of 3. Our wine predictor is even deeper, making it more difficult to follow the logic. In scikit-learn, we can limit the depth of a trained tree using the `max_depth` hyperparameter. For example, running `DecisionTreeClassifier(max_depth=2)` will create a classifier whose depth cannot exceed two nested statements. Let's demonstrate by training a limited depth classifier on our wine data.

Listing 22.49 Training a tree with limited depth

```
clf = DecisionTreeClassifier(max_depth=2)
clf.fit(X, y)
text_tree = tree.export_text(clf, feature_names=feature_names)
print(text_tree)

--- proline <= 755.00
|   --- od280/od315_of_diluted_wines <= 2.11
|   |   --- class: 2
|   |   --- od280/od315_of_diluted_wines >  2.11
|   |   --- class: 1
|--- proline >  755.00
|   --- flavanoids <= 2.17
|   |   --- class: 2
|   |   --- flavanoids >  2.17
|   |   --- class: 0
```

The printed tree is two if/else statements deep. The outer statement is determined by the proline concentration: if the proline concentration is greater than 755, flavanoids are used to identify the wine.

NOTE The dataset has *flavonoids* misspelled as *flavanoids*.

Otherwise, the OD280 / OD315 of diluted wines is utilized for class determination. Based on the output, we can fully comprehend the working logic in the model. Furthermore, we can infer the relative importance of features that are driving class prediction:

- Proline is the most important feature. It appears at the top of the tree and therefore has the lowest Gini impurity. Splitting on that feature must thus lead to the most imbalanced data. In an imbalanced dataset, it is much easier to isolate one class over another, so knowing the proline concentration allows us to more easily separate the different classes of wine.

This is consistent with our linear model trained in section 21, where proline coefficients yielded the most noticeable signal.

- Flavonoids and OD280 / OD315 are also important drivers of prediction (although not as important as proline).
- The remaining 10 features are less relevant.

The depth at which a feature appears in the tree is an indicator of its relative importance. That depth is determined by the Gini impurity. Hence, the Gini impurity can be used to compute an importance score. The importance scores across all features are stored in the `feature_importances_` attribute of `clf`. Listing 22.50 prints `clf.feature_importances_`.

NOTE More precisely, scikit-learn computes feature importance by subtracting the Gini impurity of the feature split from the Gini impurity of the previous split. For instance, in the wine tree, the impurity of the flavonoids at depth 2 is subtracted from the impurity of proline at depth 1. After the subtraction, the importance is weighted by the fraction of training samples represented during the split.

Listing 22.50 Printing the feature importances

```
print(clf.feature_importances_)

[0.          0.          0.          0.          0.          0.
 0.117799    0.          0.          0.          0.          0.39637021
 0.48583079]
```

In the printed array, the importance of feature `i` is equal to `feature_importances_[i]`. Most of the features receive a score of 0 because they are not represented in the trained tree. Let's rank the remaining features based on their importance score.

Listing 22.51 Ranking relevant features by importance

```
for i in np.argsort(clf.feature_importances_)[-1:-1]:
    feature = feature_names[i]
    importance = clf.feature_importances_[i]
    if importance == 0:
        break

    print(f'{feature} has an importance score of {importance:.2f}')

'proline' has an importance score of 0.49
'od280/od315_of_diluted_wines' has an importance score of 0.40
'flavonoids' has an importance score of 0.12
```

Among our features, proline is ranked as most important. It's followed by OD280 / OD315 and the flavonoids.

Tree-based feature ranking can help draw meaningful insights from our data. We'll emphasize this point by exploring the serious problem of cancer diagnosis.

22.2.1 Studying cancerous cells using feature importance

An identified tumor could be cancerous. The tumor needs to be examined under a microscope to determine if it is malignant (cancerous) or benign (noncancerous). Zooming in on the tumor reveals individual cells. Each cell has a multitude of measurable features, including these:

- Area
- Perimeter
- Compactness (the ratio of the squared perimeter to the area)
- Radius (a cell is not perfectly round, so the radius is computed as the mean distance from the center to the perimeter)
- Smoothness (variations in the distance from the cell center to the perimeter)
- Concave points (the number of inward curves on the perimeter)
- Concavity (the average inward angle of the concave points)
- Symmetry (if one side of the cell resembles the other)
- Texture (the standard deviation of color shades in the cell image)
- Fractal dimension (the “wrigginess” of the perimeter, based on the number of separate straight-ruler measurements required to measure the wiggly border)

Imaging technology allows us to compute these features for each individual cell. However, the tumor biopsy will reveal dozens of cells beneath the microscope (figure 22.9), so the individual features must somehow be aggregated together. The simplest way to aggregate the features is to compute their mean and standard deviation. We can also store the most extreme value computed across each cell: for instance, we can record the largest concavity measured across the cells. Informally, we’ll refer to this statistic as the *worst concavity*.

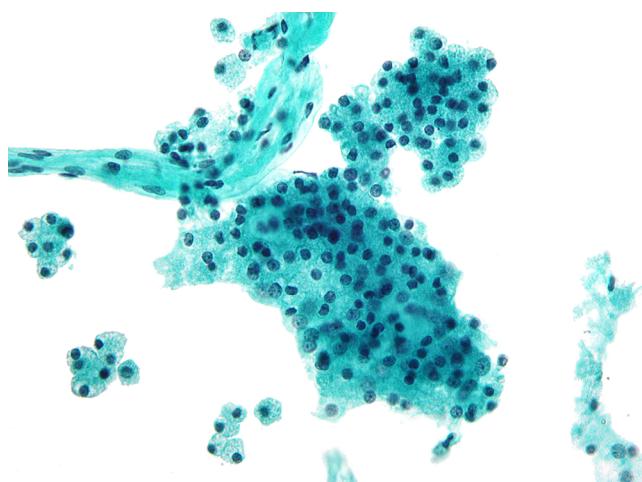


Figure 22.9 Dozens of tumor cells seen through a microscope. Each cell has 10 different measurable features. We can aggregate these features across cells using three different statistics, so we obtain 30 features total for determining whether the tumor is malignant or benign.

NOTE Usually, these features are not computed on the cell itself. Instead, they are computed on the nucleus of the cell. The nucleus is an enclosed, circular structure in the center of the cell that is easily visible through a microscope.

The three different aggregations across 10 measured features lead to 30 features total. Which features are most important for determining cancer-cell malignancy?. We can find out. Scikit-learn includes a cancer-cell dataset: let's import it from `sklearn.datasets` and print the feature names and class names.

Listing 22.52 Importing scikit-learn's cancer-cell dataset

```
from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()
feature_names = data.feature_names
num_features = len(feature_names)
num_classes = len(data.target_names)
print(f"The cancer dataset contains the following {num_classes} classes:")
print(data.target_names)
print(f"\nIt contains these {num_features} features:")
print(feature_names)

The cancer dataset contains the following 2 classes:
['malignant' 'benign']

It contains these 30 features:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

The dataset contains 30 different features. Let's rank them by importance and output the ranked features along with their importance scores. We ignore features whose importance scores are close to zero.

Listing 22.53 Ranking tumor features by importance

```
X, y = load_breast_cancer(return_X_y=True)
clf = DecisionTreeClassifier()
clf.fit(X, y)
for i in np.argsort(clf.feature_importances_)[-1:-1:-1]:
    feature = feature_names[i]
    importance = clf.feature_importances_[i]
    if round(importance, 2) == 0:
        break
    print(f"'{feature}' has an importance score of {importance:.2f}")
```

```
'worst radius' has an importance score of 0.70
'worst concave points' has an importance score of 0.14
'worst texture' has an importance score of 0.08
'worst smoothness' has an importance score of 0.01
'worst concavity' has an importance score of 0.01
'mean texture' has an importance score of 0.01
'worst area' has an importance score of 0.01
'mean concave points' has an importance score of 0.01
'worst fractal dimension' has an importance score of 0.01
'radius error' has an importance score of 0.01
'smoothness error' has an importance score of 0.01
'worst compactness' has an importance score of 0.01
```

The three top-ranking features are *worst radius*, *worst concave points*, and *worst texture*. Neither the mean nor the standard deviation drives the malignancy of the tumor; instead, it is the presence of a few extreme outliers that determines the cancer diagnosis. Even one or two irregularly shaped cells can indicate malignancy. Among the top-ranking features, *worst radius* particularly stands out: it has an importance score of 0.70. The next-highest importance score is 0.14. This difference suggests that the radius of the largest cell is an extremely important indicator of cancer. We can check this hypothesis by plotting histograms of the *worst-radius* measurements across the two classes (figure 22.10).

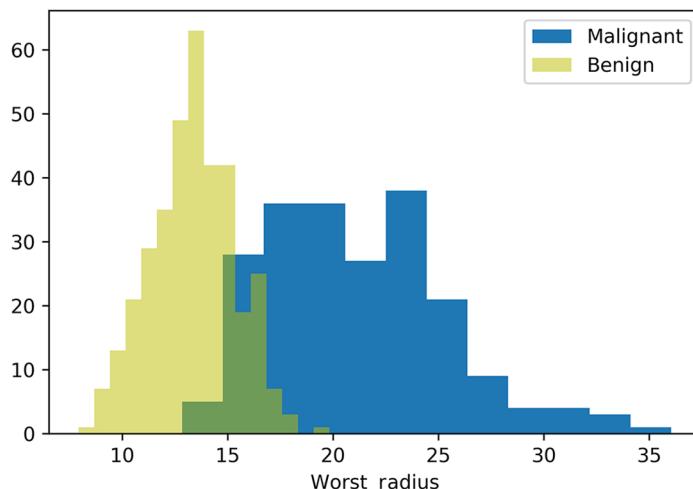


Figure 22.10 A histogram of worst-radius measurements across cancerous and noncancerous tumors. That radius is noticeably greater when the tumor is malignant and not benign.

Listing 22.54 Plotting two worst radius histograms

```
index = clf.feature_importances_.argmax()
plt.hist(X[y == 0][:, index], label='Malignant', bins='auto')
plt.hist(X[y == 1][:, index], label='Benign', color='y', bins='auto',
         alpha=0.5)
```

```
plt.xlabel('Worst Radius')
plt.legend()
plt.show()
```

The histogram reveals an enormous separation between malignant and benign worst-radius measurements. In fact, the presence of any cell radius greater than 20 units is a sure-fire indication of malignancy.

By training a decision tree, we gained insights into medicine and biology. Generally, decision trees are very useful tools for comprehending signals in complex datasets. The trees are very interpretable; their learned logical statements can easily be probed by data science. Furthermore, decision trees offer additional benefits:

- Decision tree classifiers are very quick to train. They are orders of magnitude faster than KNN classifiers. Also, unlike linear classifiers, they are not dependent on repeated training iterations.
- Decision tree classifiers don't depend on data manipulation before training. Logistic regression requires us to standardize the data before training; decision trees do not require standardization. Also, linear classifiers cannot handle categorical features without pretraining transformations, but a decision tree can handle these features directly.
- Decision trees are not limited by the geometric shape of the training data. In contrast, as shown in figure 22.1, KNN and linear classifiers cannot handle certain geometric configurations.

All these benefits come at a price: trained decision tree classifiers sometimes perform poorly on real-world data.

22.3 Decision tree classifier limitations

Decision trees learn the training data well—sometimes too well. Certain trees simply memorize the data without yielding any useful real-world insights. There are serious limits to rote memorization. Imagine a college student who is studying for a physics final. The previous year's exam is available online and includes written answers to all the questions. The student memorizes last year's exam. Given last year's questions, the student can easily recite the answers. The student is feeling confident—but on the day of the final, disaster strikes! The questions on the final are slightly different. Last year's exam asked for the velocity of a tennis ball dropped from 20 feet, but this exam asks for the velocity of a billiard ball dropped from 50 feet. The student is at a loss. They learned the answers but not the general patterns driving those answers; they can't do well on the exam because their knowledge does not generalize.

Overmemorization limits the usefulness of our trained models. In supervised machine learning, this phenomenon is referred to as *overfitting*. An overfitted model corresponds too closely to the training data, so it may fail to predict accurately on new observations. Decision tree classifiers are particularly prone to overfitting since they

can memorize the training data. For instance, our cancer detector `clf` has perfectly memorized the training set (X, y) . We can confirm by outputting the accuracy with which `clf.predict(X)` corresponds to y .

Listing 22.55 Checking the accuracy of the cancer-cell model

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(clf.predict(X), y)
print("Our classifier has memorized the training data with "
      f"{100 * accuracy:.0f}% accuracy.")
```

Our classifier has memorized the training data with 100% accuracy.

The classifier can identify any training example with 100% accuracy, but this does not mean it can generalize as well to real-world data. We can better gauge the classifier's true accuracy using cross-validation. Listing 22.56 splits (X, y) into training set $(X_{\text{train}}, y_{\text{train}})$ and validation set $(X_{\text{test}}, y_{\text{test}})$. We train `clf` to perfectly memorize $(X_{\text{train}}, y_{\text{train}})$ and then check how well the model can generalize to data that it has not encountered before. We do this by computing the model's accuracy on the validation set.

Listing 22.56 Checking model accuracy with cross-validation

```
np.random.seed(0)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, )
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
accuracy = accuracy_score(clf.predict(X_test), y_test)
print(f"The classifier performs with {100 * accuracy:.0f}% accuracy on "
      "the validation set.")
```

The classifier performs with 90% accuracy on the validation set.

The classifier's true accuracy is at 90%. That accuracy is decent, but we can definitely do better. We need a way to improve the performance by limiting overfitting in the tree. This can be done by training multiple decision trees at once using a technique called *random forest classification*.

Relevant scikit-learn decision tree classifier methods

- `clf = DecisionTreeClassifier()`—Initializes a decision tree classifier
- `clf = DecisionTreeClassifier(max_depth=x)`—Initializes a decision tree classifier with a maximum depth of x
- `clf.feature_importances_`—Accesses the feature importance scores of a trained decision tree classifier
- `plot_tree(clf)`—Plots a decision tree diagram associated with tree `clf`

(continued)

- `plot_tree(clf, feature_names=x, class_names=y)`—Plots a decision tree diagram with customized feature names and class labels
- `export_text(clf)`—Represents the decision tree diagram as a simple string
- `export_text(clf, feature_names=x)`—Represents the decision tree diagram as a simple string with customized feature names

22.4 Improving performance using random forest classification

Sometimes, in human affairs, the aggregated viewpoint of a crowd outperforms all individual predictions. In 1906, a crowd gathered at the Plymouth country fair to guess the weight of a 1,198-pound ox. Each person present wrote down their best guess, and the median of these guesses was tallied. The final median estimate of 1,207 pounds was within 1% of the actual weight. This aggregated triumph of collective intelligence is called the *wisdom of the crowd*.

Modern democratic institutions are built on the wisdom of the crowd. In democratic nations, the people come together and vote on the future of their country. Usually, the voters have incredibly diverse political views, opinions, and life experiences. But somehow, their accumulated choices average out into a decision that can benefit their country in the long run. This democratic process is partially dependent on the diversity of the populace. If everyone has the exact same opinion, everyone is prone to the same errors—but a diversity of views helps limit those errors. The crowd tends to make its best decisions when its members think in different ways.

The wisdom of the crowd is a natural phenomenon. It's seen not just in people but also in animals. Bats, fish, birds, and even flies can optimize their behavior when surrounded by other members of their species. The phenomenon can also be observed in machine learning. A crowd of decision trees can sometimes outperform a single tree; however, for this to happen, the inputs into each tree must be diverse.

Let's explore the wisdom of the crowd by initializing 100 decision trees. A large collection of trees is (not surprisingly) referred to as a *forest*. Hence, we store our trees in a `forest` list.

Listing 22.57 Initializing a 100-tree forest

```
forest = [DecisionTreeClassifier() for _ in range(100)]
```

How should we train the trees in the forest? Naively, we could train each individual tree on our cancer training set (`X_train`, `y_train`). However, we would end up with 100 trees that memorized the exact same data. The trees would therefore make identical predictions, so the key element of diversity would be missing from the forest. Without diversity, the wisdom of the crowd cannot be applied. What should we do?

One solution is to randomize our training data. In section 7, we studied a technique called bootstrapping with replacement. In this technique, the contents of an N -element dataset are sampled repeatedly. The elements are sampled with replacement, which means duplicate elements are allowed. Through sampling, we can generate a new N -element dataset whose contents differ from the original data. Let's bootstrap our training data to randomly generate a new training set (`X_train_new`, `y_train_new`).

Listing 22.58 Randomly sampling a new training set

```
Applies bootstrapping to
training set (X, y) to generate
a brand-new training set
np.random.seed(1)
def bootstrap(X, y):
    num_rows = X.shape[0]
    indices = np.random.choice(range(num_rows), size=num_rows,
                               replace=True)
    X_new, y_new = X[indices], y[indices]
    return X_new, y_new

Samples random indices of data
points in (X, y). The sampling is
carried out with replacement,
so certain indices may be
sampled twice.
X_train_new, y_train_new = bootstrap(X_train, y_train)
assert X_train.shape == X_train_new.shape
assert y_train.size == y_train_new.size
assert not np.array_equal(X_train, X_train_new)
assert not np.array_equal(y_train, y_train_new)

>Returns a random training set
based on the sampled indices
The bootstrapped
training set is as large as
the original training set.

The bootstrapped training set
is not equal to the original set.
```

Now, let's run our `bootstrap` function 100 times to generate 100 different training sets.

Listing 22.59 Randomly sampling 100 new training sets

```
np.random.seed(1)
features_train, classes_train = [], []
for _ in range(100):
    X_train_new, y_train_new = bootstrap(X_train, y_train)
    features_train.append(X_train_new)
    classes_train.append(y_train_new)
```

Across our 100 training sets, the data may be different, but all the features are the same. However, we can increase the overall diversity by randomizing the features in `features_train`. In general, the wisdom of the crowd works best when different individuals pay attention to diverging features. For instance, in a democratic election, the top priorities of urban voters might not overlap with those of rural voters. Urban voters might focus on housing policies and crime, and rural voters might focus on crop tariffs and property taxes. These diverging priorities can lead to a consensus that benefits both urban and rural voters in the long run.

More concretely, in supervised machine learning, feature diversity can limit overfitting. Consider, for example, our cancer dataset. As we have seen, “worst radius” is an incredibly impactful feature. Thus, all trained models where that feature is included will

rely on the radius as a crutch. But on certain rare occasions, a tumor might be cancerous even if the radius is low. Conformist models will mislabel that tumor if they all rely on the same features. Imagine, however, if we train some models without including the radius in their feature sets: these models will be forced to search for alternate, subtle patterns of cell malignancy and will be more resilient to fluctuating real-world observations. Individually, each model might not perform as well because its feature set is limited. Collectively, models should perform better than each individual tree.

We'll aim to train the trees in the forest on random samples of features in `features_train`. Currently, each feature matrix holds 30 cancer-related measurements. We need to lower the feature count in each random sample from 30 to a smaller number. What is an appropriate sample count? Well, it has been shown that the square root of the total feature count is usually a good choice for the sample size. The square root of 30 equals approximately 5, so we'll set our sample size to 5.

Let's iterate over `features_train` and filter each feature matrix to five random columns. We also track the indices of the randomly chosen features for later use during validation.

Listing 22.60 Randomly sampling the training features

We randomly sample 5 of 30 feature columns for each tree. Sampling is carried out without replacement because a duplicate feature will not yield novel signals during training.

```
np.random.seed(1)
sample_size = int(X.shape[1] ** 0.5) ←
assert sample_size == 5 ←
feature_indices = [np.random.choice(range(30), 5, replace=False) ←
                   for _ in range(100)] →
```

The feature sample size is equal to approximately the square root of the total feature count.

Given 30 features total, we expect the sample size to equal 5.

```
for i, index_subset in enumerate(feature_indices):
    features_train[i] = features_train[i][:, index_subset]
```

```
for index in [0, 99]: ←
    index_subset = feature_indices[index]
    names = feature_names[index_subset]
    print(f"\nRandom features utilized by Tree {index}:")
    print(names)
```

Prints the randomly sampled feature names in the very first and very last feature subsets

```
Random features utilized by Tree 0:
['concave points error' 'worst texture' 'radius error'
 'fractal dimension error' 'smoothness error'] ←
```

Five randomly sampled features do not include the worst-radius measurement.

```
Random features utilized by Tree 99:
['mean smoothness' 'worst radius' 'fractal dimension error'
 'worst concave points' 'mean concavity'] ←
```

Five randomly sampled features include the impactful worst-radius measurement.

We've randomized each of our 100 feature matrices by both row (data point) and column (feature). The training data for every tree is very diverse. Let's train each *i*th tree in `forest` on training set (`features_train[i]`, `classes_train[i]`).

Listing 22.61 Training the trees in the forest

```
for i, clf_tree in enumerate(forest):
    clf_tree.fit(features_train[i], classes_train[i])
```

We've trained every tree in the forest. Now, let's put the trained trees to a vote. What is the class label of the data point at `X_test[0]`? We can check using the wisdom of the crowd. Here, we iterate across every trained `clf_tree` in the forest. For every *i*th iteration, we do the following:

- 1** Utilize the tree to predict the class label at `X_test[0]`. As a reminder, each *i*th tree in `forest` depends on a random subset of features. The chosen feature indices are stored in `feature_indices[i]`. Hence, we need to filter the `X_test[0]` by the chosen indices before making the prediction.
- 2** Record the prediction as the *vote* of the tree at index *i*.

Once all the trees have voted, we tally the 100 votes and select the class label that has received the plurality of the votes.

NOTE This process is very similar to the KNN plurality voting that we utilized in section 20.

Listing 22.62 Using tree voting to classify a data point

```
from collections import Counter
feature_vector = X_test[0]
votes = []
for i, clf_tree in enumerate(forest):
    index_subset = feature_indices[i]
    vector_subset = feature_vector[index_subset]
    prediction = clf_tree.predict([vector_subset])[0]
    votes.append(prediction)
```

```
Counts all votes
Iterates over 100 trained trees
Adjusts the columns in feature_vector to correspond with the five random feature indices associated with each tree
Each tree casts a vote by returning a class-label prediction.
```

```
class_to_votes = Counter(votes)
for class_label, votes in class_to_votes.items():
    print(f"We counted {votes} votes for class {class_label}.")
```

```
top_class = max(class_to_votes.items(), key=lambda x: x[1])[0]
print(f"\nClass {top_class} has received the plurality of the votes.")
```

We counted 93 votes for class 0.
We counted 7 votes for class 1.

Class 0 has received the plurality of the votes.

93% of the trees voted for Class 0. Let's check if this majority vote is correct.

Listing 22.63 Checking the true class of the predicted label

```
true_label = y_test[0]
print(f"The true class of the data-point is {true_label}.")
```

The true class of the data-point is 0.

The forest has successfully identified the point at `X_test[0]`. Now, we will use voting to identify all points in the `X_test` validation set and utilize `y_test` to measure our prediction accuracy.

Listing 22.64 Measuring the accuracy of the forest model

```

predictions = []
for i, clf_tree in enumerate(forest):
    index_subset = feature_indices[i]
    prediction = clf_tree.predict(X_test[:,index_subset])
    predictions.append(prediction)

predictions = np.array(predictions)
y_pred = [Counter(predictions[:,i]).most_common()[0][0] for i in range(y_test.size)]
accuracy = accuracy_score(y_pred, y_test)
print("The forest has predicted the validation outputs with "
      f"{100 * accuracy:.0f}% accuracy")

The forest has predicted the validation outputs with 96% accuracy

```

Our randomly generated forest has predicted the validation outputs with 96% accuracy. It outperformed our single trained decision tree, whose accuracy hovered at 90%. By using the wisdom of the crowd, we have managed to improve performance. In the process, we have also trained a *random forest classifier*: a collection of trees whose training inputs are randomized to maximize diversity. Random forest classifiers are trained in the following manner:

- 1 Initialize N decision trees. The number of trees is a hyperparameter. Generally, more trees lead to higher accuracy, but using too many trees increases the classifier's running time.
- 2 Generate N random training sets by sampling with replacement.
- 3 Choose $N^{**} 0.5$ feature columns at random for each of our N training sets.
- 4 Train all the decision trees across the N random training sets.

After training, each tree in the forest casts a vote on how to label inputted data. These votes are tallied, and the class with the most votes is outputted by the classifier.

Random forest classifiers are very versatile and are not prone to overfitting. Scikit-learn, of course, includes a random forest implementation.

22.5 Training random forest classifiers using scikit-learn

In scikit-learn, random forest classification is carried out by the `RandomForestClassifier` class. Let's import that class from `sklearn.ensemble`, initialize the class, and train it using `(X_train, y_train)`. Finally, we check the classifier's performance using the validation set `(X_test, y_test)`.

Listing 22.65 Training a random forest classifier

```
np.random.seed(1)
from sklearn.ensemble import RandomForestClassifier
clf_forest = RandomForestClassifier()
clf_forest.fit(X_train, y_train)
y_pred = clf_forest.predict(X_test)
accuracy = accuracy_score(y_pred, y_test)
print("The forest has predicted the validation outputs with "
      f"{100 * accuracy:.0f}% accuracy")
```

This result is slightly higher than our earlier result of 96% due to random fluctuations as well as additional optimizations provided by scikit-learn.

The forest has predicted the validation outputs with 97% accuracy

By default, scikit-learn's random forest classifier utilizes 100 decision trees. However, we can specify a lower or higher count using the `n_estimators` parameter. The following code reduces the number of trees to 10 by running `RandomForestClassifier(n_estimators=10)`. Then we recompute the accuracy.

Listing 22.66 Training a 10-tree random forest classifier

```
np.random.seed(1)
clf_forest = RandomForestClassifier(n_estimators=10)
clf_forest.fit(X_train, y_train)
y_pred = clf_forest.predict(X_test)
accuracy = accuracy_score(y_pred, y_test)
print("The 10-tree forest has predicted the validation outputs with "
      f"{100 * accuracy:.0f}% accuracy")
```

The 10-tree forest has predicted the validation outputs with 97% accuracy

Even with the lower tree count, the total accuracy remains very high. Sometimes, 10 trees are more than sufficient to train a very accurate classifier.

Each of the 10 trees in `clf_forest` is assigned a random subset of five features. Every feature in the subset contains its own feature importance score. Scikit-learn can average all these scores across all the trees, and the aggregated averages can be accessed by calling `clf_forest.feature_importances_`. Let's utilize the `feature_importances_` attribute to print the top three features in the forest.

Listing 22.67 Ranking the random forest features

```
for i in np.argsort(clf_forest.feature_importances_)[-1:-1:-1][3]:
    feature = feature_names[i]
    importance = clf_forest.feature_importances_[i]
    print(f'{feature} has an importance score of {importance:.2f}')

'worst perimeter' has an importance score of 0.20
'worst radius' has an importance score of 0.16
'worst area' has an importance score of 0.16
```

The worst-radius feature continues to rank high, but its ranking is now on par with worst area and worst perimeter. Unlike our decision tree, the random forest does not

over-rely on any individual inputted feature. This gives the random forest more flexibility in handling fluctuating signals in new data. The classifier's versatile nature makes it a popular choice when training on medium-sized datasets.

NOTE Random forest classifiers function very well on multifeature datasets with hundreds or thousands of points. However, once the dataset size enters the millions, the algorithm can no longer scale. When processing exceedingly large datasets, more powerful deep learning techniques are required. None of the problems in this book fall in the scope of that requirement.

Relevant scikit-learn random forest classifier methods

- `clf = RandomForestClassifier()`—Initializes a random forest classifier
- `clf = RandomForestClassifier(n_estimators=x)`—Initializes a random forest classifier in which the number of trees is set to `x`
- `clf.feature_importances_`—Accesses the feature importance scores of a trained random forest classifier

Summary

- Certain classification problems can be handled with nested `if/else` statements but not with KNN or logistic regression classifiers.
- We can train a single-feature `if/else` model by maximizing accuracy across the co-occurrence counts between each feature state and each class label.
- We can train a two-feature nested `if/else` model by doing a *binary split* on one of the features. Splitting on the feature returns two different training sets. Each training set is associated with a unique split-feature state. The training sets can be used to compute two single-feature models, and then we can combine the models into a nested `if/else` statement. The nested model's accuracy is equal to the weighted mean of the simpler model accuracies.
- The choice of feature for the binary split can impact the quality of the model. Generally, the split leads to superior results if it generates imbalanced training data. A training set's imbalance can be captured using its class-label distribution. More imbalanced training sets have a higher distribution vector magnitude. Hence, imbalanced datasets yield a higher value for $v @ v$, where v is the distribution vector. Additionally, the value $1 - v @ v$ is referred to as the *Gini impurity*. Minimizing the Gini impurity minimizes the training set imbalance, so we should always split on the feature that yields the minimal Gini impurity.
- We can extend two-feature model training to handle N features. We train an N -feature model by splitting on the feature with the minimum Gini impurity. Then we train two simpler models, each of which handles $N - 1$ features. The two simpler models are then combined into a more complex nested model whose accuracy is equal to the weighted mean of the simpler model accuracies.

- The branching `if/else` statements in the trained conditional models resemble the branches of a tree. We can make the resemblance more explicit by visualizing the output as a *decision tree diagram*. Decision trees are special network structures used to symbolize `if/else` decisions. Any nested `if/else` statement can be visualized as a decision tree, so trained `if/else` conditional classifiers are referred to as *decision tree classifiers*.
- Decision tree *depth* equals the number of nested `if/else` statements required to capture the logic in the tree. Limiting the depth can yield more interpretable diagrams.
- The depth at which a feature appears in the tree is an indicator of its relative importance. That depth is determined by the Gini impurity, so the Gini impurity can be used to compute an importance score.
- Overmemorization limits the usefulness of our trained models. In supervised machine learning, this phenomenon is referred to as *overfitting*. An overfitted model corresponds too closely to the training data, so it may fail to accurately predict on new observations. Decision tree classifiers are particularly prone to overfitting since they can memorize the training data.
- We can limit overfitting by training multiple decision trees in parallel. This collection of trees is called a *forest*. The collective wisdom of the forest can outperform an individual tree, but this requires us to introduce diversity to the forest. We can add diversity by generating random training sets with randomly chosen features. Then every tree in the forest is trained on a diverging training set. After training, each tree in the forest casts a vote on how to label inputted data. This voting-based ensemble model is referred to as a *random forest classifier*.

Case study 5 solution

This section covers

- Cleaning data
- Exploring networks
- Feature engineering
- Optimizing machine learning models

FriendHook is a popular social networking app designed for college campuses. Students can connect as friends in the FriendHook network. A recommendation engine emails users weekly with new friend suggestions based on their existing connections; students can ignore these recommendations, or they can send out friend requests. We have been provided with one week's worth of data pertaining to friend recommendations and student responses. That data is stored in the friendhook/Observations.csv file. We're provided with two additional files: friendhook/Profiles.csv and friendhook/Friendships.csv, containing user profile information and the friendship graph, respectively. The user profiles have been encrypted to protect student privacy. Our goal is to build a model that predicts user behavior in response to the friend recommendations. We will do so by following these steps:

- 1 Load the three datasets containing the observations, user profiles, and friendship connections.

- 2 Train and evaluate a supervised model that predicts behavior based on network features and profile features. We can optionally split this task into two subtasks: training a model using network features, and then adding profile features and evaluating the shift in model performance.
- 3 Check to ensure that the model generalizes well to other universities.
- 4 Explore the inner workings of our model to gain better insights into student behavior.

WARNING Spoiler alert! The solution to case study 5 is about to be revealed. We strongly encourage you to try to solve the problem before reading the solution. The original problem statement is available for reference at the beginning of the case study.

23.1 Exploring the data

Let's separately explore the Profiles, Observations, and Friendships tables. We will clean and adjust the data in these tables if required.

23.1.1 Examining the profiles

We start by loading the Profiles table into Pandas and summarizing the table's contents.

Listing 23.1 Loading the Profiles table

```
import pandas as pd
def summarize_table(df):
    n_rows, n_columns = df.shape
    summary = df.describe()
    print(f"The table contains {n_rows} rows and {n_columns} columns.")
    print("Table Summary:\n")
    print(summary.to_string())

df_profile = pd.read_csv('friendhook/Profiles.csv')
summarize_table(df_profile)
```

We'll reuse this summarize function on the other two tables in our dataset.

The table contains 4039 rows and 6 columns.
Table Summary:

	Profile_ID	Sex	Relationship_Status	Dorm	Major	Year
count	4039	4039	3631	4039	4039	4039
unique	4039	2	3	15	30	4
top	b90a1222d2b2	e807eb960650	ac0b88e46e20	a8e6e404d1b3	141d4cdd5aaf	c1a648750a4b
freq	1	2020	1963	2739	1366	1796

The table contains 4,039 distinct profiles distributed across two different sexes. The most frequent sex is mentioned in 2,020 of 4,039 profiles, so we can infer that the profiles represent an equal distribution between males and females. Furthermore, the profiles capture a student body distribution that's spread across 30 majors and 15 dormitories. Suspiciously, the most frequently mentioned dorm contains over 2,700 students. This number seems large, but a quick Google search reveals that large on-campus student complexes are not uncommon. For instance, the 17-story Sandburg Residence Hall at the University of Wisconsin-Milwaukee can house 2,700 students. These numbers may also represent students in the *Off-Campus Housing* category. The count can be explained by a multitude of hypotheses—but going forward, we should consider the various driving factors behind the numbers we observe. Rather than blindly crunching numbers, we should keep in mind that our data is derived from real-world behaviors and physical constraints of university students.

There is one anomaly in the table summary, in the *Relationship Status* column. Pandas has detected three *Relationship Status* categories across 3,631 of 4,039 table rows. The remaining 400 or so rows are null—they don't contain any assigned relationship status. Let's count the number of empty rows.

Listing 23.2 Counting empty *Relationship Status* profiles

```
is_null = df_profile.Relationship_Status.isnull()  
num_null = df_profile[is_null].shape[0]  
print(f'{num_null} profiles are missing the Relationship Status field.')  
  
408 profiles are missing the Relationship Status field.
```

408 profiles are missing a value in the *Relationship_Status* field. This makes sense: as stated in the problem statement, the *Relationship_Status* field is optional. It appears that one-tenth of the students refused to specify that field. But we cannot continue our analysis with empty values in our data; we need to either delete the empty rows or replace the empty fields with some other value. Deleting the empty rows is not a good option—we'd be throwing out potentially valuable information in the other columns. Instead, we can treat the lack of status as a fourth *unspecified* relationship status category. To do so, we should assign these rows a category ID. What ID value should we choose? Before we answer the question, let's examine all unique IDs in the *Relationship Status* column.

Listing 23.3 Checking unique *Relationship Status* values

```
unique_ids = set(df_profile.Relationship_Status.values)  
print(unique_ids)  
  
{'9cea719429e9', nan, '188f9a32c360', 'ac0b88e46e20'}
```

As expected, the *Relationship Status* values are composed of three hash codes and an empty nan. The hash codes are encrypted versions of the three possible status categories:

Single, In a Relationship, and It's Complicated. Of course, we cannot know which category is which. All we can determine is whether two profiles fall in the same status category. Our aim is to eventually use this information in a trained machine learning model. However, the scikit-learn library is unable to process hash codes or null values: it can only process numbers, so we need to convert the categories to numeric values. The simplest solution would be to assign each category a number between 0 and 4. Let's execute this assignment. We start by generating a dictionary mapping between each category and number.

Listing 23.4 Mapping Relationship Status values to numbers

```
import numpy as np
category_map = {'9cea719429e9': 0, np.nan: 1, '188f9a32c360': 2,
                'ac0b88e46e20': 3} ←
{'9cea719429e9': 0, nan: 1, '188f9a32c360': 2, 'ac0b88e46e20': 3}
```

Normally, we would generate this map automatically by executing `category_map = {id_: i for i, id_ in enumerate(unique_ids)}`, but the order of numeric assignments could vary based on Python versioning. So, we manually set the mappings to ensure consistent outputs for all readers of this section.

Next, we replace the contents of the *Relationship Status* column with the appropriate numeric values.

Listing 23.5 Updating the Relationship Status column

```
nums = [category_map[hash_code]
        for hash_code in df_profile.Relationship_Status.values]
df_profile['Relationship_Status'] = nums
print(df_profile.Relationship_Status)

0      0
1      3
2      3
3      3
4      0
..
4034    3
4035    0
4036    3
4037    3
4038    0
Name: Relationship_Status, Length: 4039, dtype: int64
```

We've transformed *Relationship Status* into a numeric variable, but the remaining five columns in the table still contain hash codes. Should we also replace these hash codes with numbers? Yes! Here's why:

- As previously mentioned, scikit-learn cannot process strings or hashes. It only takes numeric values as input.

- For humans, reading hash codes is more mentally taxing than reading numbers. Thus, replacing the multicharacter codes with shorter numbers will make it easier for us to explore the data.

With this in mind, let's create a category mapping between hash codes and numbers in each column. We track the category mappings in each column with a `col_to_mapping` dictionary. We also use the mappings to replace all hash codes with numbers in `df_profile`.

Listing 23.6 Replacing all *Profile* hash codes with numeric values

```
col_to_mapping = {'Relationship_Status': category_map}

for column in df_profile.columns:
    if column in col_to_mapping:
        continue

    unique_ids = sorted(set(df_profile[column].values))
    category_map = {id_: i for i, id_ in enumerate(unique_ids)}
    col_to_mapping[column] = category_map
    nums = [category_map[hash_code]
            for hash_code in df_profile[column].values]
    df_profile[column] = nums

head = df_profile.head()
print(head.to_string(index=False))
```

Profile_ID	Sex	Relationship_Status	Dorm	Major	Year
2899	0		0	5	13
1125	0		3	12	6
3799	0		3	12	29
3338	0		3	4	25
2007	1		0	12	2

Sorting the IDs helps ensure consistent outputs for all readers, independent of their Python versioning. Note that we can only sort the hash code IDs if no nan values are present among the hash codes; otherwise, the sorting will cause an error.

We've finished tweaking `df_profile`. Now let's turn our attention to the table of experimental observations.

23.1.2 Exploring the experimental observations

We start by loading the `Observations` table into Pandas and summarizing the table's contents.

Listing 23.7 Loading the *Observations* table

```
df_obs = pd.read_csv('friendhook/Observations.csv')
summarize_table(df_obs)
```

The table contains 4039 rows and 5 columns.
Table Summary:

	Profile_ID	Selected_Friend	Selected_Friend_of_Friend	Friend_Request_Sent	Friend_Request_Accepted
count	4039	4039	4039	4039	4039
unique	4039	2219	2327	2	2
top	b90a1222d2b2	89581f99fa1e	6caa597f13cc	True	True
freq	1	77	27	2519	2460

The five table columns all consistently show 4,039 filled rows. There are no empty values in the table. This is good—but the column names are hard to read. The names are very descriptive but also very long. We should consider shortening some of the names to ease our cognitive load. Let's briefly discuss the various columns and whether some renaming would be appropriate:

- *Profile_ID*—The ID of the user who received the friend recommendation. This name is short and straightforward. It also corresponds to the *Profile_ID* column in `df_profile`. We should keep this name as is.
- *Selected_Friend*—An existing friend of the user in the *Profile_ID* column. We can simplify this column name to just *Friend*.
- *Selected_Friend_of_Friend*—A randomly chosen friend of *Selected_Friend* who was not yet a friend of *Profile_ID*. In our analysis, this random friend-of-a-friend was emailed as a *friend recommendation* for the user. We can rename this column *Recommended_Friend* or possibly *FoF*. Let's call the column *FoF*, because this acronym is memorable and short.
- *Friend_Request_Sent*—This Boolean column is `True` if a user sent a friend request to the suggested friend of a friend or `False` otherwise. Let's shorten the column name to just *Sent*.
- *Friend_Request_Accepted*—This Boolean column is only `True` if a user sent a friend request and that request was accepted. We can shorten the column name to *Accepted*.

Based on our discussion, we need to rename four of the five columns. Let's rename the columns and regenerate the summary.

Listing 23.8 Renaming the observation columns

```
new_names = {'Selected_Friend': 'Friend',
             'Selected_Friend_of_Friend': 'FoF',
             'Friend_Request_Sent': 'Sent',
             'Friend_Request_Accepted': 'Accepted'}
df_obs = df_obs.rename(columns=new_names)
summarize_table(df_obs)
```

The table contains 4039 rows and 5 columns.

Table Summary:

	Profile_ID	Friend	FoF	Sent	Accepted
count	4039	4039	4039	4039	4039
unique	4039	2219	2327	2	2
top	b90a1222d2b2	89581f99fa1e	6caa597f13cc	True	True
freq	1	77	27	2519	2460

In the updated table, the statistics are clearer. The observations contain 2,219 unique *Friend* IDs and 2,327 unique *FoF* IDs out of 4,039 samples total. This means, on average, each *Friend* and *FoFID* is utilized approximately twice. No single profile ID dominates our data, which is reassuring. This will allow us to more easily design a robust predictive model, as opposed to a model that is driven by a single profile signal and thus more susceptible to overtraining.

Further examination reveals that approximately 62% (2,519) of the friend suggestions led to a friend request being sent. This is very promising; the friend-of-a-friend suggestions are quite effective. Furthermore, approximately 60% (2,460) of sampled instances led to a friend request being accepted; the sent friend requests are ignored or rejected just 2% ($2519 - 2460 = 50$) of the time. Of course, our numbers assume that there are no observations where *Sent* is False and *Accepted* is True. This scenario is not possible because a friend request cannot be accepted if it has not yet been sent. Still, as a sanity check, let's test the integrity of the data by confirming that the scenario does not take place.

Listing 23.9 Ensuring that *Sent* is True for all accepted requests

```
condition = (df_obs.Sent == False) & (df_obs.Accepted == True)
assert not df_obs[condition].shape[0]
```

Based on our observations, user behavior follows three possible scenarios:

- A user rejects or ignores the friend recommendation listed in the *FoF* column. This occurs in 38% of instances.
- A user sends a friend request based on the recommendation, and the friend request is accepted. This occurs in 62% of instances.
- A user sends a friend request based on the recommendation, and the friend request is rejected or ignored. This scenario is rare, occurring in just 1.2% of total instances.

Each of these three scenarios represents three categories of user behavior. Hence, we can encode this categorical behavior by assigning numbers 0, 1, and 2 to behavior patterns *a*, *b*, and *c*. Here, we carry out the categorical assignments and store them in a *Behavior* column.

Listing 23.10 Assigning classes of behavior to the user observations

```
behaviors = []
for sent, accepted in df_obs[['Sent', 'Accepted']].values:
    behavior = 2 if (sent and not accepted) else int(sent) * int(accepted) ←
        behaviors.append(behavior)
df_obs['Behavior'] = behaviors
```

Python treats Boolean values True and False as simple integer values 1 and 0, respectively. So, this arithmetic operation returns either 0, 1, or 2 based on our behavior definitions.

Additionally, we must transform the profile IDs in the first three columns from hash codes to numeric IDs consistent with `df_profile.Profile_ID`. The following code utilizes the mapping stored in `col_to_mapping['Profile_ID']` for this purpose.

Listing 23.11 Replacing all *Observation* hash codes with numeric values

```
for col in ['Profile_ID', 'Friend', 'FoF']:
    nums = [col_to_mapping['Profile_ID'][hash_code]
            for hash_code in df_obs[col]]
    df_obs[col] = nums

head = df_obs.head()
print(head.to_string(index=False))

Profile_ID  Friend   FoF    Sent  Accepted  Behavior
2485       2899    2847  False   False      0
2690       2899    3528  False   False      0
3904       2899    3528  False   False      0
709        2899    3403  False   False      0
502        2899    345   True    True      1
```

`df_obs` now aligns with `df_profile`. Only a single data table remains unanalyzed. Let's explore the friendship linkages in the `Friendships` table.

23.1.3 Exploring the *Friendships* linkage table

We start by loading the `Friendships` table into Pandas and summarizing the table's contents.

Listing 23.12 Loading the *Friendships* table

```
df_friends = pd.read_csv('friendhook/Friendships.csv')
summarize_table(df_friends)
```

The table contains 88234 rows and 2 columns.
Table Summary:

	Friend_A	Friend_B
count	88234	88234
unique	3646	4037

	Friend_A	Friend_B
top	89581f99fa1e	97ba93d9b169
freq	1043	251

There are over 88,000 friendship links in this social network. The social network is quite dense, with an average of approximately 22 friends per FriendHook profile. One social butterfly in the network (*89581f99fa1e*) has more than 1,000 friends. However, an exact friend count cannot be gauged because the two columns in the network are not symmetric. In fact, we cannot even validate whether all 4,039 profiles are appropriately represented in the table.

To carry out a more detailed analysis, we should load the friendship data into a NetworkX graph. Listing 23.13 computes the social graph. We represent the node IDs with the numeric values mapped from the hash codes in the columns. After we compute the graph, we count the number of nodes in `G.nodes`.

Listing 23.13 Loading the social graph into NetworkX

```
import networkx as nx
G = nx.Graph()
for id1, id2 in df_friends.values:
    node1 = col_to_mapping['Profile_ID'][id1]
    node2 = col_to_mapping['Profile_ID'][id2]
    G.add_edge(node1, node2)

nodes = list(G.nodes)
num_nodes = len(nodes)
print(f"The social graph contains {num_nodes} nodes.")

The social graph contains 4039 nodes.
```

Let's try to gain more insights into the graph structure by visualizing it with `nx.draw` (figure 23.1). Note that the graph is rather large, so visualization might take 10 to 30 seconds of running time to complete.

Listing 23.14 Visualizing the social graph

```
import matplotlib.pyplot as plt
np.random.seed(0)
nx.draw(G, node_size=5)
plt.show()
```

Tightly clustered social groups are clearly visible in the network. Let's extract these groups using Markov clustering and then count the number of clusters.

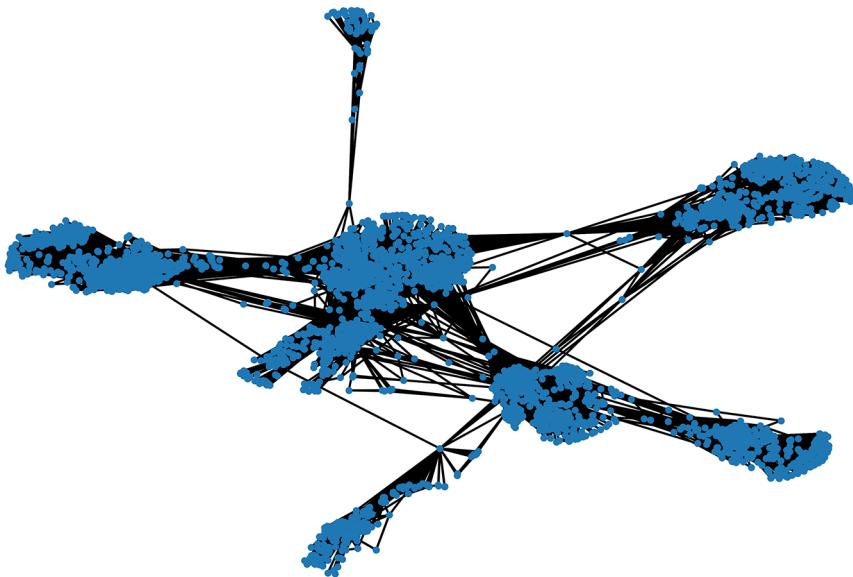


Figure 23.1 The university's visualized social graph. Tight social-group clusters are clearly visible; these can be extracted using Markov clustering.

Listing 23.15 Finding social groups using Markov clustering

```
import markov_clustering as mc
matrix = nx.toSciPySparse_matrix(G)
result = mc.run_mcl(matrix)
clusters = mc.get_clusters(result)
num_clusters = len(clusters)
print(f"{num_clusters} clusters were found in the social graph.")

10 clusters were found in the social graph.
```

Ten clusters were found in the social graph. Let's visualize these clusters by coloring each node based on cluster ID. To start, we need to iterate over clusters and assign a `cluster_id` attribute to every node.

Listing 23.16 Assigning cluster attributes to nodes

```
for cluster_id, node_indices in enumerate(clusters):
    for i in node_indices:
        node = nodes[i]
        G.nodes[node]['cluster_id'] = cluster_id
```

Next, we color the nodes based on their cluster attribute assignment (figure 23.2).

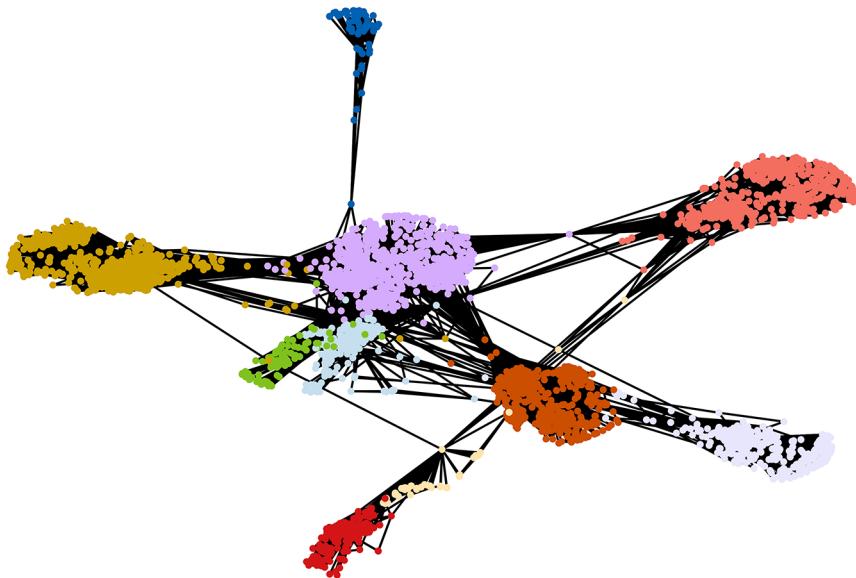


Figure 23.2 The university's visualized social graph. Tight social-group clusters have been identified using Markov clustering. The nodes in the graph are colored based on their cluster ID.

Listing 23.17 Coloring the nodes by cluster assignment

```
np.random.seed(0)
colors = [G.nodes[n]['cluster_id'] for n in G.nodes]
nx.draw(G, node_size=5, node_color=colors, cmap=plt.cm.tab20)
plt.show()
```

The cluster colors clearly correspond to tight social groups. Our clustering has been effective, so the assigned `cluster_id` attributes should be helpful during the model-building process. Similarly, it might be useful to store all five profile features as attributes in the student nodes. Let's iterate over the rows in `df_profile` and store each column value in its corresponding node.

Listing 23.18 Assigning profile attributes to nodes

```
attribute_names = df_profile.columns
for attributes in df_profile.values:
    profile_id = attributes[0]
    for name, att in zip(attribute_names[1:], attributes[1:]):
        G.nodes[profile_id][name] = att

first_node = nodes[0]
print(f"Attributes of node {first_node}:")
print(G.nodes[first_node])
```

```
Attributes of node 2899:  
{'cluster_id': 0, 'Sex': 0, 'Relationship_Status': 0, 'Dorm': 5,  
 'Major': 13, 'Year': 2}
```

We have finished exploring our input data. Now we'll train a model that predicts user behavior. We'll start by constructing a simple model that only utilizes network features.

23.2 Training a predictive model using network features

Our goal is to train a supervised machine learning model on our dataset to predict user behavior. Currently, all possible classes of behavior are stored in the *Behavior* columns of `df_obs`. Our three behavior class labels are 0, 1, and 2. As a reminder, the Class 2 label occurs in just 50 of the 4,039 sampled instances: Class 2 is very imbalanced relative to the other class labels. There is a case to be made for removing these 50 labeled examples from our training data. For the time being, let's leave in these examples to see what happens; later, we will remove them if necessary. For now, we assign our training class label array to equal the `df_obs.Behavior` column.

Listing 23.19 Assigning the class-label array `y`

```
y = df_obs.Behavior.values  
print(y)  
  
[0 0 0 ... 1 1 1]
```

Now that we have class labels, we need to create a feature matrix `x`. Our goal is to populate this matrix with features arising from the social graph structure. Later, we'll add additional features from the student profiles, so we don't need to assemble the feature matrix all at once. We will build up the matrix slowly, adding new features in batches to better understand the impact of these features on model performance. With this in mind, let's create an initial version of `x` and populate it with some very basic features. The simplest question we can ask about any FriendHook user is this: how many friends does the user have? That value equals the edge count associated with the user's node in the social graph. In other words, the friend count of user `n` is equal to `G.degree(n)`. Let's make this count the very first feature in the matrix. We'll iterate over all the rows in `df_obs` and assign an edge count to each profile referenced in each row. As a reminder, every row contains three profiles: *Profile_ID*, *Friend*, and *FoF*. We'll calculate the friend count for each profile, creating features *Profile_ID_Edge_Count*, *Friend_Edge_Count*, and *FoF_Edge_Count*.

NOTE It's not always easy to come up with a good, consistent feature name. Rather than choosing *FoF_Edge_Count*, we could have chosen *FoF_Friend_Count* as our name. However, maintaining consistency would have forced us to include a *Friend_Friend_Count* feature as well, leading to a very awkward feature name. Alternatively, we could have named our three features *Profile_Degree*, *Friend_Degree*, and *FoF_Degree*. These names would be short and informative, but

it's worth remembering that one of our profile features pertains to college majors. In the context of college, both degrees and majors have a nearly identical definition, so a degree-based naming convention could cause confusion down the line. That is why we're sticking with an *Edge_Count* suffix.

Let's generate a matrix of 3-by-4,039 edge count features. We need a way to track these features along with the associated feature names. We also need a way to easily update features and their names with additional inputs. One straightforward solution is to store the features in a df_features Pandas table. That table will allow us to access the feature matrix via df_features.values. Let's compute df_features to create an initial version of our feature matrix.

Listing 23.20 Creating a feature matrix from edge counts

```
cols = ['Profile_ID', 'Friend', 'FoF']
features = {f'{col}_Edge_Count': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        degree = G.degree(node) ←
        features[feature_name].append(degree)
df_features = pd.DataFrame(features)
X = df_features.values
```

As a reminder, the node's degree equals the edge count of that node. Hence, G.degree(n) returns the friend count associated with user n.

We have an initial training set in place. Let's check the quality of the signal in that set by training and testing a simple model. We have multiple possible models to choose from. One sensible choice is a decision tree classifier; decision trees can handle non-linear decision boundaries and are easily interpretable. On the downside, they are prone to overtraining, so cross-validation will be required to measure model performance appropriately. Listing 23.21 trains a decision tree on a subset of (X, y) and evaluates the results on the remaining data. During the evaluation, we should keep in mind that our Class 2 labels are highly imbalanced. Thus, the f-measure metric will provide a more reasonable assessment of performance than simple accuracy.

NOTE Through the remainder of this section, we repeatedly train and test our classifier models. Listing 23.21 defines an evaluate function for this purpose that takes as input a training set (X, y) and a model type preset to DecisionTreeClassifier. The function then splits X, y into training and test sets, trains the classifier, and computes an f-measure using the test set. Finally, it returns both the f-measure and the classifier for evaluation.

Listing 23.21 Training and evaluating a decision tree classifier

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
```

```

→ def evaluate(X, y, model_type=DecisionTreeClassifier, **kwargs):
    np.random.seed(0)
    X_train, X_test, y_train, y_test = train_test_split(X, y) ←
    clf = model_type(**kwargs)
    clf.fit(X_train, y_train) ← Trains the model
    pred = clf.predict(X_test)
    f_measure = f1_score(pred, y_test, average='macro') ←
    return f_measure, clf

```

Splits (X, y) into training and test sets

```

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:.2f}")

```

The f-measure is 0.37

We'll repeatedly utilize this function over the remainder of this section. It trains a classifier on a subsample of data in (X, y) . The classifier type is specified using the `model_type` parameter: here, the parameter is preset to a decision tree classifier. Additional classifier hyperparameters can be specified using `**kwargs`. After training, the classifier's performance is evaluated using a retained subset of the data.

Computes an f-measure. The `average='macro'` parameter is required because three class labels are present in the training data.

Our f-measure is terrible! Clearly, the edge count by itself is not sufficient for predicting user behavior. Perhaps a more sophisticated measure of node centrality is required. Earlier, we learned how the PageRank centrality measure can be more informative than edge count. Would adding PageRank values to our training set improve model performance? Let's find out.

Listing 23.22 Adding PageRank features

```

node_to_pagerank = nx.pagerank(G)
features = {f'{col}_PageRank': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        pagerank = node_to_pagerank[node]
        features[feature_name].append(pagerank)

def update_features(new_features):
    for feature_name, values in new_features.items():
        df_features[feature_name] = values
    return df_features.values

```

This function is utilized repeatedly. It updates the `df_features` Pandas table with novel features in the `new_features` dictionary.

```

X = update_features(features)
f_measure, clf = evaluate(X, y)

```

Returns the altered feature matrix

```

print(f"The f-measure is {f_measure:.2f}")

```

The f-measure is 0.38

The f-measure remains approximately the same. Basic centrality measures are insufficient. We need to expand X to include the social groups uncovered by Markov clustering. After all, two people in the same social group are more likely to be friends. How do we incorporate these social groups into the feature matrix? Well, naively, we could

assign the `cluster_id` attribute of each referenced node as our social group feature. However, this approach has a serious downside: our current cluster IDs are only relevant to the specific social graph in `G`. They are not at all relevant to any other college network. In other words, a model trained on the cluster IDs in `G` is not applicable to some other college graph in `G_other`. This won't do! One of our goals is to construct a model that is generalizable across other colleges. Thus, we need a more nuanced solution.

One alternate approach is just to consider the following binary question: are two people in the same social group? If they are, then perhaps they are more likely to eventually become friends on FriendHook. We can make this binary comparison between each pair of profile IDs in a single row of observations. More precisely, we can ask the following:

- Does the user in the `Profile_ID` column fall in the same social group as the friend in the `Friend` column? We'll name this feature `Shared_Cluster_id_f`.
- Does the user in the `Profile_ID` column fall in the same social group as the friend of a friend in the `FoF` column? We'll name this feature `Shared_Cluster_id_fof`.
- Does the friend in the `Friend` column fall in the same social group as the friend of a friend in the `FoF` column? We'll name this feature `Shared_Cluster_f_fof`.

Let's answer these three questions by adding the three additional features. Then we test whether these features yield improved model performance.

Listing 23.23 Adding social group features

```
features = {f'Shared_Cluster_{e}': []
            for e in ['id_f', 'id_fof', 'f_fof']}

i = 0
for node_ids in df_obs[cols].values:
    c_id, c_f, c_fof = [G.nodes[n]['cluster_id']
                         for n in node_ids]
    features['Shared_Cluster_id_f'].append(int(c_id == c_f))
    features['Shared_Cluster_id_fof'].append(int(c_id == c_fof))
    features['Shared_Cluster_f_fof'].append(int(c_f == c_fof))

X = update_features(features)
f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:.2f}")

The f-measure is 0.43
```

Our f-measure has improved from 0.38 to 0.43. Performance is still poor, but the social group inclusion has led to a slight enhancement of our model. How important are the new social group features relative to the model's current performance? We can check using the `feature_importance_` attribute of our trained classifier.

Listing 23.24 Ranking features by their importance score

```

def view_top_features(clf, feature_names):
    for i in np.argsort(clf.feature_importances_)[-1:-10:-1]:
        feature_name = feature_names[i]
        importance = clf.feature_importances_[i]
        if not round(importance, 2):
            break
    print(f'{feature_name}: {importance:0.2f}')
feature_names = df_features.columns
view_top_features(clf, feature_names)

Shared_Cluster_id_fof: 0.18
FoF_PageRank: 0.17
Profile_ID_PageRank: 0.17
Friend_PageRank: 0.15
FoF_Edge_Count: 0.12
Profile_ID_Edge_Count: 0.11
Friend_Edge_Count: 0.10

```

Sorts the features based on importance score

Prints the top features along with their importance scores in the classifier based on order of importance

Features with an importance score of less than 0.01 are not displayed.

The *Shared_Cluster_id_foff* feature is the most important feature in the model. In other words, the social group overlap between the user and the friend of a friend is the most important predictor of a future online friendship. However, the PageRank features also rank highly on the list, which indicates that social graph centrality plays some role in friendship determination. Of course, our model's performance is still poor, so we should be cautious with our inferences about how the features drive predictions. Instead, we should focus on improving model performance. What other graph-based features could we utilize? Perhaps the network cluster size can impact the predictions. We can find out, but we should be careful about trying to keep our model generalizable. Cluster size can inexplicably take the place of a cluster ID, making the model very specific to the university. Let's explore how this could occur.

Suppose our dataset has two social clusters, A and B. The clusters contain 110 and 115 students, respectively. Thus, their sizes are nearly identical and should not drive prediction. Now, let's further suppose the students in Cluster A are more likely to become FriendHook friends than students in Cluster B. Our model would pick up on this during training and associate a size of 110 with a propensity for friendship. Essentially, it would treat the size like a cluster ID! This could cause trouble in the future if the model encountered a brand-new cluster with size 110.

So should we ignore cluster size altogether? Not necessarily. We are scientists, and we wish to honestly explore how cluster size impacts model prediction. But we should be very cautious: if cluster size has minimal impact on model quality, we should delete it from our features. However, if the size drastically improves model prediction, we will cautiously reevaluate our options. Let's test what happens when we add cluster size to our list of features.

Listing 23.25 Adding cluster-size features

```

cluster_sizes = [len(cluster) for cluster in clusters]
features = {f'{col}_Cluster_Size': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        c_id = G.nodes[node]['cluster_id']
        features[feature_name].append(cluster_sizes[c_id])

X = update_features(features)
f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:.2f}")

```

The f-measure is 0.43

The cluster size did not improve the model. As a precaution, let's delete it from our feature set.

Listing 23.26 Deleting cluster-size features

```

Deletes all feature names in df_features that match the regex
regular expression. It is utilized elsewhere in this section.

import re
def delete_features(df_features, regex=r'Cluster_Size'):

    df_features.drop(columns=[name for name in df_features.columns
                               if re.search(regex, name)], inplace=True)
    return df_features.values

X = delete_features(df_features)

```

← Returns the altered feature matrix

The f-measure remains at 0.43. What else can we do? Perhaps we should try thinking outside the box. In what ways can social connections drive real-world behavior? Are there additional, problem-specific signals we can harness? Yes! Consider the following scenario. Suppose we analyze a student named Alex, whose node ID in network G is n . Alex has 50 FriendHook friends, who are accessible through $G[n]$. We randomly sample two of the friends in $G[n]$. Their node IDs are a and b . We then check if a and b are friends. They are! It seems that a is in $list(G[n])$. We then repeat this 100 times. In 95% of sampled instances, a is a friend of b . Basically, there's a 95% likelihood that any pair of Alex's friends are also friends with each other. We'll refer to this probability as the *friend-sharing likelihood*. Now, Mary is new to FriendHook. She just joined and added Alex as her friend. We can be fairly confident that Mary will also connect with Alex's friends—though of course this is not guaranteed. But a friend-sharing likelihood of 0.95 gives us more confidence than a likelihood of 0.10.

Let's try incorporating this likelihood into our features. We start by computing the likelihood for every node in G . We store the node-to-likelihood mapping in a `friend_sharing_likelihood` dictionary.

Listing 23.27 Computing friend-sharing likelihoods

```

friend_sharing_likelihood = {}
for node in nodes:
    neighbors = list(G[node])
    friendship_count = 0
    total_possible = 0
    for i, node1 in enumerate(neighbors[:-1]):
        for node2 in neighbors[i + 1:]:
            if node1 in G[node2]:
                friendship_count += 1
    total_possible += 1

prob = friendship_count / total_possible if total_possible else 0
friend_sharing_likelihood[node] = prob

```

Tracks the count of shared friendships across neighbors

Tracks the total possible shared friendships. Note that with a bit of graph theory, we could prove this value always equals $\text{len(neighbors)} * (\text{len(neighbors} - 1))$.

Checks if two neighbors are friends

Next, we generate a friend-sharing likelihood feature for each of our three profile IDs. After adding the features, we reevaluate the trained model's performance.

Listing 23.28 Adding friend-sharing likelihood features

```

features = {f'{col}_Friend_Sharing_Likelihood': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        sharing_likelihood = friend_sharing_likelihood[node]
        features[feature_name].append(sharing_likelihood)

X = update_features(features)
f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:.0.2f}")

```

The f-measure is 0.49

Performance has increased from 0.43 to 0.49! It's still not great, but it's progressively getting better. How does the friend-sharing likelihood compare to other features in the model? Let's find out.

Listing 23.29 Ranking features by their importance score

```

feature_names = df_features.columns
view_top_features(clf, feature_names)

Shared_Cluster_id_fof: 0.18
Friend_Friend_Sharing_Likelihood: 0.13
FoF_PageRank: 0.11
Profile_ID_PageRank: 0.11
Profile_ID_Friend_Sharing_Likelihood: 0.10
FoF_Friend_Sharing_Likelihood: 0.10
FoF_Edge_Count: 0.08
Friend_PageRank: 0.07
Profile_ID_Edge_Count: 0.07
Friend_Edge_Count: 0.06

```

One of our new friend-sharing features ranks quite highly: it's in second place, between `Shared_Cluster_id_fof` and `FoF_PageRank`. Our outside-the-box thinking has improved the model. But the model is incomplete. An f-measure of 0.49 is not acceptable; we need to do better. It's time to move beyond network structure. We need to incorporate features from the profiles stored in `df_profiles`.

23.3 Adding profile features to the model

Our aim is to incorporate the profile attributes `Sex`, `Relationship_Status`, `Major`, `Dorm`, and `Year` into our feature matrix. Based on our experience with the network data, there are three ways in which we can do this:

- *Exact value extraction*—We can store the exact value of the profile feature associated with each of the three profile ID columns in `df_obs`. This is analogous to how we utilized the exact values of edge counts and PageRank outputs from the network.

Example feature: The relationship status of the friend of a friend in `df_obs`.

- *Equivalence comparison*—Given a profile attribute, we can carry out a pairwise comparison of the attribute across all three profile ID columns in `df_obs`. For each comparison, we return a Boolean feature demarcating whether the attribute is equal in the two columns. This is analogous to how we checked whether a profile pair belonged to the same social group.

Example feature: Do a particular user and a friend of a friend live in the same dorm? Yes or no?

- *Size*—Given a profile attribute, we can return the number of profiles that share that attribute. This is analogous to the attempted inclusion of social group size in our model.

Example feature: The number of students residing in a particular dorm.

Let's utilize exact value extraction to expand our feature matrix. Which of our five attributes are good candidates for this technique? Well, the categorical values of `Sex`, `Relationship_Status`, and `Year` are not college-dependent; they should remain consistent across all colleges and universities. This is not the case for `Dorm`—dormitory names will change in other college networks. Our goal is to train a model that can be applied to other social graphs, so the `Dorm` attribute is not a valid feature for exact value extraction.

What about the `Major` attribute? Here, the situation is trickier. Certain majors like biology and economics are shared by most colleges and universities. Other majors, like civil engineering, might appear at more technically oriented schools but not on a liberal arts college curriculum. And certain rare majors like bagpiping or astrobiology are specific to a few niche schools. Thus, we can expect some consistency across majors but not total consistency. A model harnessing the exact values of the majors will therefore be partially reusable; potentially, that partial signal could boost predictive power at some schools, but this would happen at the expense of other schools. Is

the trade-off worth it? Perhaps. The answer is not immediately clear. For the time being, let's see how well we can train our model without relying on the added crutch of Major values. If we find ourselves unable to train an adequate model, we will revisit our decision.

Let's now apply exact value extraction to Sex, Relationship_Status, and Year and then check for improvements in our model.

Listing 23.30 Adding exact value profile features

```
attributes = ['Sex', 'Relationship_Status', 'Year']
for attribute in attributes:
    features = {f'{col}_{attribute}_Value': [] for col in cols}
    for node_ids in df_obs[cols].values:
        for node, feature_name in zip(node_ids, features.keys()):
            att_value = G.nodes[node][attribute]
            features[feature_name].append(att_value)

X = update_features(features)

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:.2f}")
```

The f-measure is 0.74

Wow! The f-measure dramatically increased from 0.49 to 0.74! The profile features have provided a very valuable signal, but we can still do better. We need to incorporate information from the Major and Dorm attributes. Equivalence comparison is an excellent way to do this. The question of whether two students share the same major or dorm is independent of their university. Let's apply the equivalence comparison to the Major and Dorm attributes and then recompute the f-measure.

Listing 23.31 Adding equivalence comparison profile features

```
attributes = ['Major', 'Dorm']
for attribute in attributes:
    features = {f'Shared_{attribute}_{e}': []
                for e in ['id_f', 'id_fof', 'f_fof']}

    for node_ids in df_obs[cols].values:
        att_id, att_f, att_fof = [G.nodes[n][attribute]
                                  for n in node_ids]
        features[f'Shared_{attribute}_id_f'].append(int(att_id == att_f))
        features[f'Shared_{attribute}_id_fof'].append(int(att_id == att_fof))
        features[f'Shared_{attribute}_f_fof'].append(int(att_f == att_fof))

X = update_features(features)

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:.2f}")
```

The f-measure is 0.82

The f-measure has risen to 0.82. Incorporating the Major and Dorm attributes has improved model performance. Now let's consider adding Major and Dorm size into the mix: we can count the number of students associated with each major and dorm and include this count as one of our features. But we need to be careful; as we previously discussed, our trained model can cheat by utilizing size as a substitute for a category ID. For instance, as we've previously seen, our largest dormitory holds over 2,700 students. Thus, we can easily identify that dorm based on its size alone. We must be cautious going forward. Let's see what happens when we incorporate Major and Dorm size into our features. If there's little impact on performance, we'll delete the features from our model. Otherwise, we'll reevaluate our options.

Listing 23.32 Adding size-related profile features

```
from collections import Counter

for attribute in ['Major', 'Dorm']:
    counter = Counter(df_profile[attribute].values)
    att_to_size = {k: v
                   for k, v in counter.items()} ←
    features = {f'{col}_{attribute}_Size': [] for col in cols}
    for node_ids in df_obs[cols].values:
        for node, feature_name in zip(node_ids, features.keys()):
            size = att_to_size[G.nodes[node][attribute]]
            features[feature_name].append(size)

X = update_features(features)

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:.2f}")

The f-measure is 0.85
```

Tracks the number of times each attribute appears in our dataset

Performance has increased from 0.82 to 0.85. The introduction of size has impacted our model. Let's dive deeper into that impact. We start by printing out the feature importance scores.

Listing 23.33 Ranking features by their importance score

```
feature_names = df_features.columns.values
view_top_features(clf, feature_names)

FoF_Dorm_Size: 0.25
Shared_Cluster_id_fof: 0.16
Shared_Dorm_id_fof: 0.05
FoF_PageRank: 0.04
Profile_ID_Major_Size: 0.04
FoF_Major_Size: 0.04
FoF_Edge_Count: 0.04
Profile_ID_PageRank: 0.03
Profile_ID_Friend_Sharing_Likelihood: 0.03
```

```

Friend_Friend_Sharing_Likelihood: 0.03
Friend_Edge_Count: 0.03
Shared_Major_id_fof: 0.03
FoF_Friend_Sharing_Likelihood: 0.02
Friend_PageRank: 0.02
Profile_ID_Dorm_Size: 0.02
Profile_ID_Edge_Count: 0.02
Profile_ID_Sex_Value: 0.02
Friend_Major_Size: 0.02
Profile_ID_Relationship_Status_Value: 0.02
FoF_Sex_Value: 0.01
Friend_Dorm_Size: 0.01
Profile_ID_Year_Value: 0.01
Friend_Sex_Value: 0.01
Shared_Major_id_f: 0.01
Friend_Relationship_Status_Value: 0.01
Friend_Year_Value: 0.01

```

The feature importance scores are dominated by two features: *FoF_Dorm_Size* and *Shared_Cluster_id_fof*. These two features have importance scores of 0.25 and 0.16, respectively. All other feature scores fall below 0.01.

The presence of *FoF_Dorm_Size* is a bit concerning. As we've discussed, a single dorm dominates 50% of the network data. Is our model simply memorizing that dorm based on its size? We can find out by visualizing a trained decision tree. For simplicity's sake, we limit the tree to a depth of 2, to limit our output to just those decisions driven by the two most dominant features.

Listing 23.34 Displaying the top branches of the tree

```

from sklearn.tree import export_text
clf_depth2 = DecisionTreeClassifier(max_depth=2)
clf_depth2.fit(X, y)
text_tree = export_text(clf_depth2, feature_names=list(feature_names))
print(text_tree)

```

The `export_text` function has trouble taking NumPy arrays as input, so we convert `feature_names` to a list.

Friend-of-a-friend dorm size is less than 279.
Under these circumstances, the most likely class label is 0 (friend suggestion ignored).

Friend-of-a-friend dorm size is ≥ 279 .

Friend of a friend and user are not in the same social group. The most likely class label is 1.

Friend of a friend and user share the same social group. The most likely class label is 1 (FriendHook connection established).

According to the tree, the most important signal is whether *FoF_Dorm_Size* is less than 279. If the friend of a friend's dormitory holds fewer than 279 students, then the FoF

and the user are unlikely to become FriendHook friends. Otherwise, they are more likely to connect if they already share the same social group (`Shared_Cluster_id_fof > 0.50`). This begs the question, how many dorms contain at least 279 students? Let's check.

Listing 23.35 Checking dorms with at least 279 students

```
counter = Counter(df_profile.Dorm.values)
for dorm, count in counter.items():
    if count < 279:
        continue

    print(f"Dorm {dorm} holds {count} students.")

Dorm 12 holds 2739 students.
Dorm 1 holds 413 students.
```

Just two of the 15 dorms contain more than 279 FriendHook-registered students. Essentially, our model relies on the two most populous dorms to make its decisions. This puts us in a bind: on the one hand, the observed signal is very interesting; FriendHook connections are more likely to occur in some dorms than others. Dorm size plays a factor in these connections. This insight could allow FriendHook developers to better understand user behavior, and perhaps this understanding will lead to better user engagement. We are better off having gained this knowledge. However, our current model has a serious downside.

Our model focuses mostly on the two largest dorms in the data. This focus may not generalize to other college campuses. For instance, consider a campus whose dormitories are smaller and hold 200 students at most. The model will completely fail to predict user behavior in this instance.

NOTE Theoretically, this situation can be avoided if we divide the dorm size by the total student count. This will ensure that the dorm-size feature always lies between 0 and 1.

More worryingly, we're dealing with the very real possibility that our model simply picked up on behavior that's unique to just these two specific dorms. This is exactly the type of scenario we were asked to avoid in the problem statement. What should we do?

Unfortunately, there is no explicit correct answer. Sometimes data scientists are forced to make difficult decisions, where each decision carries risks and trade-offs. We can keep our feature list as is to maintain high model performance, but we run the risk of not being able to generalize to other schools. Alternatively, we can remove size-related features and keep our model generalizable at the expense of overall performance.

Perhaps there's a third option: we can try deleting the size-related features while also adjusting our choice of classifier. There is a slight chance that we'll achieve com-

parable performance without relying on dorm size. This is unlikely but still worth trying. Let's assign a copy of the current feature matrix to variable `X_with_sizes` (in case we need it later) and then delete all size-related features from matrix `X`. We'll then look for other ways to boost our f-measure beyond 0.82.

Listing 23.36 Deleting all size-related features

```
X_with_sizes = X.copy()
X = delete_features(df_features, regex=r'_Size')
```

23.4 Optimizing performance across a steady set of features

In section 22, we learned how random forest models tend to outperform decision trees. Will switching the model type from a decision tree to a random forest improve performance? Let's find out.

Listing 23.37 Training and evaluating a random forest classifier

```
from sklearn.ensemble import RandomForestClassifier
f_measure, clf = evaluate(X, y, model_type=RandomForestClassifier)
print(f"The f-measure is {f_measure:.2f}")
```

The f-measure is 0.75

Oh no! The performance has actually gotten worse! How can this be? Well, it's an established fact that random forests usually outperform decision trees, but this does not guarantee that random forests will always perform better. In certain training instances, decision trees are superior to random forests. This appears to be one such instance. For our particular dataset, we cannot improve predictive performance by switching to a random forest model.

NOTE In supervised machine learning, there's a well-established theorem known as the *No Free Lunch Theorem*. In layman's terms, the theorem states the following: it is impossible for a certain training algorithm to always outperform all other algorithms. In other words, we cannot rely on a single algorithm for every type of training problem. An algorithm that works great most of the time will not work great all of the time. Random forests perform well on most problems but not on all problems. In particular, random forests perform poorly when prediction depends on just one or two inputted features. Random feature sampling can dilute that signal and worsen the quality of predictions.

Switching the type of model has not helped. Perhaps instead we can boost performance by optimizing on the hyperparameters. In this book, we've focused on a single decision tree hyperparameter: maximum depth. Currently, maximum depth is set to `None`. This means the tree's depth is not restricted. Will limiting the depth improve

our predictions? Let's quickly check using a simple grid search. We scan across the `max_depth` parameter values that range from 1 to 100 and settle on the depth that optimizes performance.

Listing 23.38 Optimizing maximum depth using a grid search

```
from sklearn.model_selection import GridSearchCV
np.random.seed(0)

hyperparams = {'max_depth': list(range(1, 100)) + [None]}
clf_grid = GridSearchCV(DecisionTreeClassifier(), hyperparams,
                       scoring='f1_macro', cv=2) ←
clf_grid.fit(X, y)
best_f = clf_grid.best_score_
best_depth = clf_grid.best_params_['max_depth']
print(f"A maximized f-measure of {best_f:.2f} is achieved when "
      f"max_depth equals {best_depth}")

A maximized f-measure of 0.84 is achieved when max_depth equals 5
```

By passing `cv=2`, we carry out two-fold cross-validation to be more consistent with our current random splitting of (X, y) into training and test datasets. Note that the grid search could split our data slightly differently, leading to fluctuations in the f-measure value.

Setting `max_depth` to 5 improves the f-measure from 0.82 to 0.84. This level of performance is comparable with our dorm-size-dependent model. Thus, we have achieved performance parity without relying on dorm size. Of course, the story is not over: we cannot make a fair comparison without first running a grid search on the size-inclusive `X_with_sizes` feature matrix. Will optimizing on `X_with_sizes` yield an even better classifier? Let's find out.

NOTE A curious reader may wonder whether our random forest output could be improved by running a grid search on the number of trees. In this particular instance, the answer is no. Altering the tree count from 100 to some other number will not significantly improve performance.

Listing 23.39 Applying a grid search to size-dependent training data

```
np.random.seed(0)
clf_grid.fit(X_with_sizes, y)
best_f = clf_grid.best_score_
best_depth = clf_grid.best_params_['max_depth']
print(f"A maximized f-measure of {best_f:.2f} is achieved when "
      f"max_depth equals {best_depth}")

A maximized f-measure of 0.85 is achieved when max_depth equals 6
```

The grid search did not improve performance on `X_with_sizes`. Thus, we can conclude that with the right choice of maximum depth, the size-dependent and size-independent models perform with approximately equal quality, and we can train a generalizable,

size-independent model without sacrificing performance. That's great news! Let's train a decision tree on `X` using a `max_depth` of 5 and then explore the real-world implications of our model.

Listing 23.40 Training a decision tree with `max_depth` set to 5

```
clf = DecisionTreeClassifier(max_depth=5)
clf.fit(X, y)
```

23.5 Interpreting the trained model

Let's print our model's feature importance scores.

Listing 23.41 Ranking features by their importance score

```
feature_names = df_features.columns
view_top_features(clf, feature_names)

Shared_Dorm_id_fof: 0.42
Shared_Cluster_id_fof: 0.29
Shared_Major_id_fof: 0.10
Shared_Dorm_f_fof: 0.06
Profile_ID_Relationship_Status_Value: 0.04
Profile_ID_Sex_Value: 0.04
Friend_Edge_Count: 0.02
Friend_PageRank: 0.01
Shared_Dorm_id_f: 0.01
```

Only nine important features remain. The top four features pertain to shared dorms, social groups, and majors. They are followed by features demarcating the category of a user's *Sex* and *Relationship Status*. Simple network features like edge count and PageRank appear at the very bottom of the list. Interestingly enough, our friend-sharing likelihood feature doesn't even make the list! This abandoned feature required effort and imagination to implement. It was satisfying to see the f-measure rise by 0.06 units once the friend-sharing likelihood was added. But in the end, that effort did not matter. With enough additional features, the friend-sharing likelihood was rendered irrelevant. Such experiences can sometimes feel frustrating. Unfortunately, feature selection is still less of a science and more of an art; it is difficult to know in advance which features to use and which features to avoid. We cannot know how a feature will integrate holistically into a model until we actually train the model. This does not mean we shouldn't get creative—creativity usually pays off. As scientists, we should experiment! We should try to use every possible signal at our disposal until adequate performance is achieved.

Let's return to our top features. Only three features have an importance score at or above 0.10: `Shared_Dorm_id_fof`, `Shared_Cluster_id_fof`, and `Shared_Major_id_fof`. Thus, the model is primarily driven by the following three questions:

- Do the user and the friend of a friend share a dormitory? Yes or no?
- Do the user and the friend of a friend share a social group? Yes or no?
- Do the user and the friend of a friend share a major? Yes or no?

Intuitively, if the answers to all three questions are yes, then the user and the friend of a friend are more likely to connect on FriendHook. Let's test this intuition by displaying the tree. We'll limit the tree's depth to 3 to simplify our output while ensuring that the top three features are represented appropriately.

Listing 23.42 Displaying the top branches of the tree

```
clf_depth3 = DecisionTreeClassifier(max_depth=3)
clf_depth3.fit(X, y)
text_tree = export_text(clf_depth3,
                      feature_names=list(feature_names))
print(text_tree)
```

The user and the friend of a friend do not share a dormitory.

The user and the friend of a friend do not share a social group. Under these circumstances, the friend suggestion is ignored (Class 0 dominates).

The user and the friend of a friend share a dormitory.

In this branch, Class 2 dominates, and the Sex feature drives Class 2 prediction. We'll soon investigate this unexpected result.

The user and the friend of a friend share a social group. Under these circumstances, the FriendHook connection is likely (Class 1 dominates).

As expected, dorm and social-group sharing primarily drive the model's predictions. If the user and friend of a friend share both a dorm and a social group, they are more likely to connect. If they share neither a dorm nor a social group, they are less likely to connect. Additionally, individuals can connect if they fall in the same social group and share the same major, even if not the same dorm.

NOTE The text representation of the tree lacks the exact count of class labels at each tree branch. As we discussed in section 22, we can produce these counts by calling `plot_tree(clf_depth3, feature_names=list(feature_names))`. For brevity's sake, we don't generate the tree plot, but you're encouraged to try this visualization. In the visualized tree statistics, you should see that the user and the *FoF* share both a cluster and a dorm in 1,635 instances; 93% of these instances represent Class 1 labels. Also, you'll observe that the user and the *FoF* share neither cluster nor dorm in 356 instances; 97%

of these instances represent Class 0 labels. Thus, social group and dorm sharing are strong predictors of user behavior.

We are nearly ready to deliver to our employer a model based on social groups, dorms, and majors. The model's logic is very straightforward: users who share social groups and living spaces or study schedules are more likely to connect. There's nothing surprising about that. What is surprising is how the *Sex* feature drives Class 2 label prediction. As a reminder, the Class 2 label corresponds to a rejected FriendHook request. According to our tree, rejection is more likely when

- The users share a dorm but are not in the same social group.
- The request sender is of a certain specific sex.

Of course, we know that Class 2 labels are fairly sparse in our data. They occur just 1.2% of the time. Perhaps the model's predictions are caused by random noise arising from the sparse sampling. We can find out. Let's quickly check how well we predict rejection. We'll execute `evaluate` on `(X, y_reject)` where `y_reject[i]` equals 2 if `y[i]` equals 2, and equals 0 otherwise. In other words, we'll evaluate a model that only predicts rejection. If that model's f-measure is low, then our predictions are driven primarily by random noise.

Listing 23.43 Evaluating a rejection classifier

```
y_reject = y * (y == 2)
f_measure, clf_reject = evaluate(X, y_reject, max_depth=5)
print(f"The f-measure is {f_measure:.02f}")
```

The f-measure is 0.97

Wow, the f-measure is actually very high! We can predict rejection very well, despite the sparsity of data. What features drive rejection? Let's check by printing the new feature importance scores.

Listing 23.44 Ranking features by their importance score

```
view_top_features(clf_reject, feature_names)

Profile_ID_Sex_Value: 0.40
Profile_ID_Relationship_Status_Value: 0.24
Shared_Major_id_fof: 0.21
Shared_Cluster_id_fof: 0.10
Shared_Dorm_id_fof: 0.05
```

Interesting! Rejection is primarily driven by the user's `Sex` and `Relationship_Status` attributes. Let's visualize the trained tree to learn more.

Listing 23.45 Displaying the rejection-predicting tree

```
text_tree = export_text(clf_reject,
                      feature_names=list(feature_names))
print(text_tree)
```

```

--- Shared_Cluster_id_fof <= 0.50      ←
|--- Shared_Major_id_fof <= 0.50
|   |--- Shared_Dorm_id_fof <= 0.50
|   |   |--- class: 0
|   |--- Shared_Dorm_id_fof >  0.50
|   |   |--- Profile_ID_Relationship_Status_Value <= 2.50
|   |   |   |--- class: 0
|   |   |--- Profile_ID_Relationship_Status_Value >  2.50      ←
|   |   |   |--- Profile_ID_Sex_Value <= 0.50
|   |   |   |   |--- class: 0
|   |   |   |--- Profile_ID_Sex_Value >  0.50
|   |   |   |   |--- class: 2
|--- Shared_Major_id_fof >  0.50
|   |--- Profile_ID_Sex_Value <= 0.50
|   |   |--- class: 0
|   |--- Profile_ID_Sex_Value >  0.50
|   |   |--- Profile_ID_Relationship_Status_Value <= 2.50
|   |   |   |--- class: 0
|   |   |--- Profile_ID_Relationship_Status_Value >  2.50
|   |   |   |--- class: 2
--- Shared_Cluster_id_fof >  0.50      ←
|--- class: 0

```

The user and the friend of a friend do not share a social group.

The user's relationship status is equal to 3. If the user's sex is equal to 1, rejection (Class 2) is likely, so rejection is dependent on a user's sex and relationship status.

The user and the friend of a friend share a social group. Under such circumstances, rejection is unlikely.

According to the tree, rejection is likely to occur under the following circumstances:

- The users do not belong to the same social group.
- The users share either a dorm or a major.
- The sender's sex is Category 1.
- The sender's relationship status is Category 3. According to the tree, the status category must be greater than 2.5. However, the maximum value of `df_Profile.Relationship_Status` is 3.

Essentially, individuals with *Sex* Category 1 and *Relationship Status* Category 3 are sending friend requests to people outside their social group. These friend requests are likely to get rejected. Of course, we cannot precisely identify the categories that lead to rejection, but as scientists, we can still speculate. Given what we know about human nature, it wouldn't be surprising if this behavior is driven by single men. Perhaps men are trying to connect with women outside their social group to get a date; if so, their requests are likely to be rejected. Again, all this is speculation, but this hypothesis is worth discussing with the product managers at FriendHook. If our hypothesis is correct, then certain changes should be introduced to the product. More steps could be taken to limit unwanted dating requests. Alternatively, new product changes could be added that make it easier for single people to connect.

23.5.1 Why are generalizable models so important?

In this case study, we have agonized over keeping our model generalizable. A model that does not generalize beyond the training set is worthless, even if the performance score seems high. Unfortunately, it's hard to know whether a model can generalize

until it's tested on external data. But we can try to remain aware of hidden biases that won't generalize well to other datasets. Failure to do so can yield serious consequences. Consider the following true story.

For many years, machine learning researchers have tried to automate the field of radiology. In radiology, trained doctors examine medical images (such as X-rays) to diagnose disease. This can be treated as a supervised learning problem in which the images are features and the diagnoses are class labels. By the year 2016, multiple radiology models were published in the scientific literature. Each published model was supposed to be highly accurate based on internal evaluation. That year, leading machine learning researchers publicly declared that "we should stop training radiologists" and that "radiologists should be worried about their jobs." Four years later, the negative publicity had led to a worldwide radiologist shortage—medical students were reluctant to enter a field that seemed destined for full automation. But by 2020, the promise of automation had failed to materialize. Most of the published models performed very poorly on new data. Why? Well, it turns out that imaging outputs differ from hospital to hospital. Different hospitals use slightly different lighting and different settings on their imaging machines. Thus, a model trained at Hospital A could not generalize well to Hospital B. Despite their seemingly high performance scores, the models were not fit for generalized use. The machine learning researchers had been too optimistic; they failed to take into account the biases inherent in their data. These failures inadvertently led to a crisis in the medical community. A more thoughtful evaluation of generalizability could have prevented this from happening.

Summary

- Superior machine learning algorithms do not necessarily work in every situation. Our decision tree model outperformed our random forest model, even though random forests are considered superior in the literature. We should never blindly assume that a model will always work well in every possible scenario. Instead, we should intelligently calibrate our model choice based on the specifics of the problem.
- Proper feature selection is less of a science and more of an art. We cannot always know in advance which features will boost a model's performance. However, the commonsense integration of diverse and interesting features into our model should eventually improve prediction quality.
- We should pay careful attention to the features we feed into our model. Otherwise, the model may not generalize to other datasets.
- Proper hyperparameter optimization can sometimes significantly boost a model's performance.
- Occasionally, it seems like nothing is working and our data is simply insufficient. However, with grit and perseverance, we can eventually yield meaningful resources. Remember, a good data scientist should never give up until they have exhausted every possible avenue of analysis.

index

Symbols

'&' operator 252

A

<a> tags 389
accuracy score 530
add_edges_from method 464
add_node method 455–456
add_nodes_from method 464
add_random_edge method 470
adjacency matrices 454, 489
algorithms 439
DBSCAN 185–190
clustering based on non-Euclidean distance 187–190
comparing K-means and 186–187
K-means 174–181
clustering using scikit-learn 175–177
optimal K using elbow method 177–181
KNN 544–546
all-by-all similarities 274, 285–287
alternative hypothesis 116
@ operator 261, 284, 286
apt-get method 199
argmax method 43
argsort method 358
arithmetic mean 82

arrays, NumPy 38–50
binning similar points in histogram plots 41–43
computing histograms 49–50
deriving probabilities from histograms 43–46
shrinking range of high confidence interval 46–49
arrowsize parameter 457
assign method 146
astype method 552
automated learning of logical rules 587–614
deciding which feature to split on 599–607
training if/else models with more than two features 608–614
training nested if/else model using two features 593–599
AWS (Amazon Web Services) 433
ax.add_features method 204
ax.coastlines() method 203, 210
AxesSubplot object 378
axes variable 378
axis array 324
ax.scatter() method 206–208, 210
ax.set_extent method 207–208
ax.set_global() method 206, 210
ax.stock_img() method 203–204, 210

B

bag-of-words technique 257
base environment 200
Beautiful Soup 394–401
Bernoulli distribution 84
bias 553, 557
biased coin 7–8, 36–38
biased deck of cards 51–53
bias_shift 557, 560
bigrams 376
binary splits 596
bin_edges array 42–43, 50, 127
binning 41–43
binomial distribution 36
bins parameter 41
bin_width 44
<body> tag 387–388, 392
Bonferroni correction 123
bootstrap function 627
bootstrapping with replacement 124–132
brew install proj geos function 199
bs class 394
buckles 454

C

capitalization inconsistency 251
card game case study
optimizing strategies for 10-card deck 64–68
overview 2
predicting red cards in shuffled deck 59–64
problem statement 1

Cartopy 198–210
 manually installing 199
 utilizing Conda package manager 199–201
 visualizing maps 201–210
`cartopy.crs` module 201
`cartopy.feature` module 204
 case studies
 assessing online ad clicks for significance
 41 shades of blue 162
 computing p-values from differences in means 157–160
 dataset description 70
 determining statistical significance 161–162
 overview 70
 problem statement 69–70
 processing ad-click table in Pandas 155–157
 finding winning strategy in card game
 optimizing strategies for 10-card deck 64–68
 overview 2
 predicting red cards in shuffled deck 59–64
 problem statement 1
 predicting future friendships from social network data
 adding profile features to model 652–657
 data 635–645
 dataset description 447–449
 interpreting trained Model 659–663
 optimizing performance across steady set of features 657–659
 overview 449
 problem statement 445–447
 training predictive model using network features 645–652
 tracking disease outbreaks using news headlines
 extracting insights from location clusters 238–243
 extracting locations from headline data 227
 overview 166
 problem statement 165

visualizing and clustering extracted location data 233–237
 using online job postings to improve data science resume
 clustering skills in relevant job postings 422–443
 extracting skill requirements from job posting data 405–412
 filtering jobs by relevance 412–422
 overview 247
 problem statement 245–247
`cdf` method 127
 centrality
 clustering data into groups 168–174
 mean as measure of 76–85
 measuring using traffic simulations 486–488
 central limit theorem 49
 determining mean and variance of population through random sampling 103–107
 making predictions using mean and variance 107–113
 computing area beneath normal curve 109–112
 interpreting computed probability 112–113
 manipulating normal distribution using SciPy 95–103
 class attribute 392, 398
 classes 520
 classifiers 520
`clf` 540–542, 574, 615, 617, 620, 625
`clf.feature_importances_` 620, 625, 632
`clf_grid.best_estimator_.attribute` 543
`clf_grid.best_params_` 543–544
`clf_grid.best_score_` 543–544
`clf_grid` object 544
`clf_grid.predict` 544
`clf.intercept_` 574, 577, 579
`clf.n_neighbors` attribute 540
`clf.predict` method 540
`clf.weights` attribute 540
`cloud_generator` object 373
 cluster_groups list 425
`cluster_id` attributes 516, 643–644, 648
 clustering
 2D data in one dimension 293–309
 4D data in two dimensions 315–323
 analyzing clusters using Pandas 191–192
 DBSCAN 185–190
 clustering based on non-Euclidean distance 187–190
 comparing K-means and 186–187
 K-means 174–181
 clustering using scikit-learn 175–177
 optimal K using elbow method 177–181
 skills in relevant job postings 422–443
 analyzing 700 most relevant postings 440–443
 exploring clusters at alternative values of K 436–439
 grouping job skills into 15 clusters 425–431
 investigating soft-skill clusters 434–436
 investigating technical skill clusters 431–434
 texts by topic 363–372
 using centrality 168–174
 using density 181–185
 visualizing text clusters 372–383
`cluster_model` object 176, 309, 363
 clusters array 181, 190
 clusters attribute 643
 clusters list 512
`cluster_to_image` 426
`cluster_to_image` function 425
`cmap` parameter 150, 516
 coastlines method 201
 coefficients 553, 579–582
 coin flips
 comparing multiple distributions 26–32
 plotting probabilities 22–32
 simulations for random 34–38

Colab (Google Colaboratory) 345
 color_func parameter 376
 color parameter 26–27
 col_to_mapping dictionary 638
 columns
 modifying 145–148
 retrieving 141–143
 combined_if_else function 611
 combine_if_else functions 595
 communities 499
 community detection 499
 community structure 499
 compare_flow_distributions()
 503
 components matrix 313
 compute_centrality function 238
 compute_event_probability
 function 8–10, 51–52
 compute_high_confidence_
 interval 50
 compute_p_value function 125,
 131
 compute_stop_likelihoods
 function 497
 conda activate base
 function 200
 conda activate new_env
 function 200
 conda create -n new_env
 function 200
 conda deactivate command 200
 conda info function 200
 conda install -c anaconda
 ipykernel 200
 conda install -c conda-forge
 cartopy 199
 Conda package manager
 199–201
 confidence intervals
 shrinking range of high 46–49
 using histograms and NumPy
 arrays 38–53
 binning similar points in
 histogram plots 41–43
 computing histograms
 49–50
 deriving probabilities from
 histograms 43–46
 shrinking range of high
 confidence interval
 46–49
 confusion matrix 528
 confusion_matrix function
 536–537

connected components 510
 cosine similarities 272
 cosine_similarities 285, 359, 414
 cosine_similarity_matrix 363
 CountVectorizer 343, 348, 356
 covariance matrix 324, 327
 cov_matrix 325, 327–331
 CSR (compressed sparse row)
 345
 csr_matrix 345
 csr module 208
 cumulative distribution
 function 117
 cv parameter 543

D

dart function 170
 data 410
 dimension reduction of
 matrix data
 clustering 2D data in one
 dimension 293–309
 clustering 4D data in two
 dimensions 315–323
 computing principal compo-
 nents without
 rotation 323–336
 dimension reduction using
 PCA and scikit-learn
 309–315
 efficient dimension reduc-
 tion using SVD and
 scikit-learn 336–338
 downloading and parsing
 online 401–403
 relationships between proba-
 bility and 72–76
 saving and loading table
 data 148–149
 data analysts 411
 data dredging 121–124
 data fishing 123
 data frame 138
 DataFrameGroupBy object
 192
 DataFrame objects 138, 237
 data science resume case study
 clustering skills in relevant job
 postings 422–443
 analyzing 700 most relevant
 postings 440–443
 exploring clusters at alter-
 native values of K
 436–439

grouping job skills into 15
 clusters 425–431
 investigating soft-skill
 clusters 434–436
 investigating technical skill
 clusters 431–434
 extracting skill requirements
 from job posting
 data 405–412
 filtering jobs by relevance
 412–422
 overview 247
 problem statement 245–247
 data scientists 411
 datasets, large text
 clustering texts by topic
 363–372
 computing similarities across
 large document
 datasets 358–362
 loading online forum discus-
 sions using scikit-
 learn 341–343
 ranking words by both post
 frequency and
 count 350–358
 vectorizing documents using
 scikit-learn 343–350
 visualizing text clusters
 372–383
 DBSCAN (density-based spatial
 clustering of applications
 with noise) 185–190, 363
 clustering based on non-
 Euclidean distance
 187–190
 comparing K-means and
 186–187
 DecisionTreeClassifier 614,
 616–617, 646
 decision tree classifiers
 limitations of 624–626
 training using scikit-learn
 614–624
 decision tree diagrams 613
 decompose method 399, 410
 degree method 484
 degree of centrality 484
 dense_region_cluster 184
 density 44, 49, 181–185
 describe method 156, 228, 405,
 422
 detect_setosa function 319
 df.values 141
 diacritics 219

die rolls
 analyzing multiple 10–11
 computing probabilities using weighted sample spaces 11–12
 simulations for random 34–38

dimension reduction of matrix data
 clustering 2D data in one dimension 293–309
 clustering 4D data in two dimensions 315–323
 computing principal components without rotation 323–336

dimension reduction using PCA and scikit-learn 309–315

efficient dimension reduction using SVD and scikit-learn 336–338

directed distance 566

directed edges 453

directed graphs 453

disease outbreaks case study
 extracting insights from location clusters 238–243
 extracting locations from headline data 227
 overview 166
 problem statement 165
 visualizing and clustering extracted location data 233–237

dispersion 85–92

<div> tag 391–392

divergence 115–120

document frequency 350

dot product 260

downloading online data 401–403

draw() method 457

draw function 466

dynamic graph theory techniques
 community detection using Markov clustering 498–512

computing travel probabilities using matrix multiplication 489–498

computing PageRank centrality using NetworkX 496–498

deriving PageRank centrality from probability theory 492–496

uncovering central nodes based on expected traffic in a network 483–488

uncovering friend groups in social networks 513–517

E

edge_exists function 453

edges 452, 454

edge weights 478

eigenvalue 326–327, 332

eigen_vec 326–327

eigenvectors 325, 332

elbow method 177–181

element_tag 400–401

else condition 613

end method 221

eps parameter 185, 187, 190, 236–237, 363–364

equidistant cylindrical projection 201

equivalence comparison 652

error matrix 528

Euclidean distance 170

euclidean_distances function 523

Euclidean norm 267

evaluate functions 646

event condition 4, 16

events 4, 16

exact value extraction 652

explained_variance_ratio_attribute 310, 316, 337

export_text function 615–616

extremeness 130

extremes, evaluating using interval analysis 13–15

F

false negatives 532

false positives 531

feature_importance_attribute 648

feature_importances_attribute 620, 631

feature_names parameter 615–616

features 520

features module 204, 207

feature vector 616

fetch_20newsgroups 341

figure variable 378

find_all method 397–398

find method 395, 398

find_top_principal_components function 333

first principal component 311

first principal direction 306, 315

fit method 540

fit_transform method 309, 316, 320, 338, 348

fit_words method 373

flags parameter 222

flow 500

flow_matrix 500, 505–507, 509

f-measure 533

force-directed layout 461

forests 626

for loops 282, 284

friend-of-a-friend recommendation algorithm 446

G

G.add_edges_from method 461

G.add_node method 456

G.add_nodes_from method 459–460

Gamma distribution 102

Gaussian distribution 49

G.draw() method 457, 459

G.edges() method 507

generalizable models 662–663

generic_sample_space variable 8

geographic location
 great-circle distance 195–198

location tracking using GeoNamesCache 211–221

accessing city information 215–219

accessing country information 212–214

limitations of GeoNames-Cache library 219–221

matching location names in text 221–224

plotting maps using Cartopy 198–210

manually installing GEOS and Cartopy 199

utilizing Conda package manager 199–201

visualizing maps 201–210

GeoNamesCache 211–221
 accessing city information 215–219
 accessing country information 212–214
 limitations of library 219–221
G
 GeonamesCache location-tracking object 211
 GeonamesCache object 219
 GEOS, installing 199
 Gini impurity 605
 G.nodes 456, 483, 500, 509, 642
 graph clustering 499
 graphs 451
 graph theory dynamic techniques community detection using Markov clustering 498–512 computing travel probabilities using matrix multiplication 489–498 uncovering central nodes based on expected traffic in a network 483–488 uncovering friend groups in social networks 513–517 ranking websites by popularity 452–464 utilizing undirected graphs to optimize travel time between towns 465–480 computing fastest travel time between nodes 473–480 modeling complex network of towns and counties 467–472 great-circle distance 195–198 great_circle_distance function 195–197 GridSearchCV class 542–543 grid searches 539–544 groupby method 192, 238

H

<h1> tag 392
 <h2> tag 391–392
 harmonic mean 533
 hash code 448
 <head> tag 387–388, 392
 head element 388

heatmap 149
 histograms 38–50
 binning similar points in histogram plots 41–43
 computing 49–50
 deriving probabilities from 43–46
 shrinking range of high confidence interval 46–49
 href attribute 390, 398
 HTML (Hypertext Markup Language) 385
 exploring for skill descriptions 406–412
 parsing using BeautifulSoup Soup 394–401
 structure of documents 386–393
 <html> tag 386–387, 392
 html_contents 386–387, 393, 400, 406
 hyperlinks 389
 hyperparameters 537
 hypothesis testing, statistical assessing divergence between sample mean and population mean 115–120
 bootstrapping with replacement 124–132
 data dredging 121–124
 permutation testing 132–135

I

id attributes 389, 401
 IDFs (inverse document frequencies) 354
 if/else model 593–594, 596–597, 599, 607
 training models with more than two features 608–614
 training nested model using two features 593–599
 Image objects 374
 import cartopy 200
 in_degree method 463
 inertia 177
 _inertia attribute 177
 inflation 503
 installing GEOS and Cartopy 199
 intertools.product function 9
 interval ranges 13–15
 IPython.core.display 386

is_in_interval function 13
 isin method 144
 itertools.permutations function 54
 itertools.product function 9

J

Jaccard similarity 255–257

K

K-D trees 545
 KElbowVisualizer class 367
 K-means 174–181
 clustering using scikit-learn 175–177
 optimal K using elbow method 177–181
 KNeighborsClassifier 539, 542, 544
 KNeighborsRegressor classes 540
 KNN (K-nearest neighbors) 544
 limitations of KNN algorithm 544–546
 optimizing performance 537–539
 k-NN (K-nearest neighbor graph) 524
 K principal directions 333
 Kth principal component 315
 Kth principal direction 315

L

L2 norm 267
 lambda expressions 10
 LambertConformal class 208
 Lambert conformal conic projection 208
 law of large numbers 31
 learning rate 555
 tag 390–392, 408
 likelihoods, bin_edges 127
 likelihoods array 44, 127
 likelihoods variable 49
 linear_classifier functions 553
 linear classifiers improving perceptron performance through standardization 562–565
 improving with logistic regression 565–573
 limitations of 582–584

linear classifiers (*continued*)
 linearly separating customers by size 549–554
 measuring feature importance with coefficients 579–582
 using scikit-learn 574–579
 linear decision boundaries 551, 583
 linestyle parameter 27
 loading table data 148–149
 location tracking 211–221
 accessing city information 215–219
 accessing country information 212–214
 limitations of GeoNames- Cache library 219–221
 logistic curve 568
 logistic regression 565, 572–573
 logisticRegression class 574, 576
 logistic regression classifiers 572, 574
 Log-normal distribution 103
 lower string method 251
 LSA (latent semantic analysis) 360
 lxml library 394

M

machine learning, network- driven supervised basics of supervised machine learning 519–527
 limitations of KNN algorithm 544–546
 measuring predicted label accuracy 527–537
 optimizing KNN performance 537–539
 running grid search using scikit-learn 539–544
 magnitude 267
 map extent 207
 marker parameter 27
 Markov clustering 498–512
 markov_clustering module 512
 Markov matrix 490
 markup languages 385
 Match object 221, 223
 Matplotlib basic plots 17–21
 coin-flip probabilities 22–32

matrices 277
 dimension reduction of matrix data
 clustering 2D data in one dimension 293–309
 clustering 4D data in two dimensions 315–323
 computing principal components without rotation 323–336
 dimension reduction using PCA and scikit- learn 309–315
 efficient dimension reduction using SVD and scikit-learn 336–338
 multiplication 274–287
 basic matrix operations 277–285
 computational limits of 287–290
 computing all-by-all matrix similarities 285–287
 computing travel probabilities using 489–498
 matrix deflation 330
 matrix multiplication 274, 283
 max_depth parameter 658–659
 MCL (Markov clustering) 506
 mean as measure of centrality 76–85
 comparing when population parameters are unknown 132–135
 computing p-values from differences in 157–160
 determining through random sampling 103–107
 making predictions using 107–113
 computing area beneath normal curve 109–112
 interpreting computed probability 112–113
 mean method 76, 82, 100–101, 121
 median 78
 metric_function 190
 metric parameter 187
 MiniBatchKMeans 365–366
 mini-batch K-means 365
 min output 140
 multi-class linear models 576–579
 multiple word clouds 377–383
 multivariate normal distribution 169

N

n_components parameter 360
 N decision trees 630
 N-dimensional dataset 333
 nearest_bulls_eye function 170–171
 N-element dataset 627
 nested_if_else function 610
 n_estimators parameter 631
 network analysis ranking websites by popularity 452–464
 social network analysis community detection using Markov clustering 498–512
 computing travel probabilities using matrix multiplication 489–498
 uncovering central nodes based on expected traffic in a network 483–488
 uncovering friend groups in social networks 513–517
 utilizing undirected graphs to optimize travel time between towns 465–480
 computing fastest travel time between nodes 473–480
 modeling complex network of towns and counties 467–472
 network-driven supervised machine learning basics of supervised machine learning 519–527
 limitations of KNN algorithm 544–546
 measuring predicted label accuracy 527–537
 optimizing KNN performance 537–539
 running grid search using scikit-learn 539–544
 network theory 451
 NetworkX 455–464, 496–498

news headlines case study
 extracting insights from location clusters 238–243
 extracting locations from headline data 227
 overview 166
 problem statement 165
 visualizing and clustering extracted location data 233–237
N-feature model 612
n-gram 376
ngram_range parameter 376
NLP (natural language processing) analysis
 clustering texts by topic 363–372
 computing similarities across large document datasets 358–362
 loading online forum discussions using scikit-learn 341–343
 ranking words by both post frequency and count 350–358
 vectorizing documents using scikit-learn 343–350
 visualizing text clusters 372–383
n_neighbors parameter 542
 no-binary command 199
 node centrality 484
 node_color parameter 471
 node ranking
 community detection using Markov clustering 498–512
 computing travel probabilities using matrix multiplication 489–498
 computing PageRank centrality using NetworkX 496–498
 deriving PageRank centrality from probability theory 492–496
 uncovering central nodes based on expected traffic in a network 483–488
 uncovering friend groups in social networks 513–517
 nodes 452
 node_size parameter 458
 No Free Lunch Theorem 657

non-Euclidean distance 187–190
nonlinear classifiers
 automated learning of logical rules 587–614
 deciding which feature to split on 599–607
 training if/else models with more than two features 608–614
 training nested if/else model using two features 593–599
decision tree classifier
 limitations 624–626
random forest classification
 improving performance using 626–630
 training using scikit-learn 630–632
 training decision tree classifiers using scikit-learn 614–624
nontrivial probabilities 8–12
 analyzing family with four children 8–10
 analyzing multiple die rolls 10–11
 computing die-roll probabilities using weighted sample spaces 11–12
non_zero_indices 347, 349–350, 356
normal curves
 comparing two sampled 99–103
 computing area beneath 109–112
normal distribution 49, 94–103
normalization 264, 268–272
normalized_tanimoto function 270, 286
normalized vector 268
normalize function 361
normal_likelihoods 109
norm function 270
np.arccos function 273
np.array_equal method 38
np.average function 497
np.average method 82–83, 89
np.delete function 594
np.flatnonzero function 346
np.histogram function 49–50
np.hstack method 133
np_matrix 352
np.mean method 82
np.radians function 197
np.random.choice module 104
np.random.permutation function 54
np.random.randint module 51, 103
np.random.seed(0) method 34, 368
np.random.shuffle function 54, 133
np.std method 91
np.trapz method 109
null hypothesis 116
numeric intervals 16
numeric values, replacing words with 257–262
NumPy
 analyzing biased deck of cards using confidence intervals 51–53
 computing confidence intervals using arrays 38–50
 binning similar points in histogram plots 41–43
 computing histograms 49–50
 deriving probabilities from histograms 43–46
 shrinking range of high confidence interval 46–49
matrix operations
 NumPy matrix arithmetic operations 277–279
 NumPy matrix products 282–285
 NumPy matrix row and column operations 279–282
 simulating random coin flips and die rolls using 34–38
 using permutations to shuffle cards 54–56
nx.DiGraph() 455
nx.DiGraph class 455
nx.disjoint_union function 469
nx.draw 458, 466, 516, 642
nx.Graph() 465
nx.shortest_path_length 510

O

Observation hash codes 641
 one-hot encoding 583

online ad clicks case study
41 shades of blue 162
computing p-values from differences in means 157–160
dataset description 70
determining statistical significance 161–162
overview 70
problem statement 69–70
processing ad-click table in Pandas 155–157
online job postings case study
clustering skills in relevant job postings 422–443
analyzing 700 most relevant postings 440–443
exploring clusters at alternative values of K 436–439
grouping job skills into 15 clusters 425–431
investigating soft-skill clusters 434–436
investigating technical skill clusters 431–434
extracting skill requirements from job posting data 405–412
filtering jobs by relevance 412–422
overview 247
problem statement 245–247
orthographic projection 201
out-degree 462
overfitting 624
oversampling 121–124

P

<p> tag 388–389, 392
PageRank centrality 491
computing using NetworkX 496–498
deriving from probability theory 492–496
Pandas
analyzing clusters using 191–192
analyzing tables
exploring tables 138–141
modifying table rows and columns 145–148
retrieving table columns 141–143

retrieving table rows 143–145
saving and loading table data 148–149
storing tables using basic Python 138
visualizing tables using Seaborn 149–152
processing ad-click table in 155–157
parameter sweeps 539
parsing
online data 401–403
using BeautifulSoup 394–401
patches 47
path length 478
paths 478
PatternObject 223
PCA (principal component analysis) 309
dimension reduction using 309–315
limitations of 320–323
pca.components attribute 311
pca_model 309
pca_object 310–311, 317, 320
pd.csv method 149
pd.describe() 139
pdf method 127
pd.read_csv method 148
percent_relevant_tiles function 418
perceptron class 574
perceptron performance 562–565
perceptron training algorithms 561
permutations
testing 132–135
to shuffle cards 54–56
p-hacking 123
pip install command 198
pip install geonamescache 211
pip install lxml 394
pip install
markov_clustering 512
pip install matplotlib 17, 199
pip install networkx 455
pip install numpy 34
pip install pandas 138
pip install scikit-learn 175
pip install scipy 72
pip install seaborn 149
pip install Unidecode 220
pip install wordcloud 373
pip install yellowbrick 367
pip package-management system 198
PlateCarree class 201, 208, 210
plate carrée projection 201
plotting
maps using Cartopy 198–210
manually installing GEOS and Cartopy 199
utilizing Conda package manager 199–201
visualizing maps 201–210
probabilities
basic plots 17–21
coin-flip probabilities 22–32
plot_tree function 615
plt.figure function 201
plt.fill_between method 19, 24–25
plt.hist method 41–42, 44, 47, 49–50
plt.imshow function 374
plt.legend() method 27
plt.plot method 18–19, 24–27
plt.scatter method 19, 24, 26–27
plt.show() function 18
plt.subplots 378–379
plt.text function 372
plt.xlabel method 21
plt.ylabel method 21
Poisson distribution 102
power iteration 327, 329–336
power_iteration function 329–330, 332
precision 531
predicted label accuracy 527–537
predictions
determining mean and variance of population through random sampling 103–107
manipulating normal distribution using SciPy 95–103
using mean and variance 107–113
predictions array 576
predict method 177
predict_proba method 541
_pred prediction array 540
print command 149

probabilities
 analysis using SciPy
 mean as measure of
 centrality 76–85
 relationships between data
 and probability 72–76
 variance as measure of
 dispersion 85–92
 computing using Python
 equation-free approach for
 measuring uncertainty
 in outcomes 4–8
 nontrivial probabilities
 8–12
 over interval ranges 13–15
 deriving from histograms
 43–46
 plotting using Matplotlib
 basic plots 17–21
 coin-flip probabilities 22–32
 probability density function 97
 probability mass function 73
 probability theory 3
 projection 201, 314–315
 pruning 614
 punctuation inconsistency 251
 p-values 119, 157–160
 Python
 computing probabilities using
 equation-free approach for
 measuring uncertainty
 in outcomes 4–8
 nontrivial probabilities
 8–12
 over interval ranges 13–15
 storing tables using basic 138

R

random_drive functions 486
 random forest classification 625
 improving performance
 using 626–630
 training using scikit-
 learn 630–632
 RandomForestClassifier
 class 630
 random sampling 103–107
 random_state parameter 373
 random_variable.cdf
 method 127
 random_variable.mean()
 function 128
 random_variable object 128,
 135

random_variable.pdf method
 127
 random_variable.sf method 127
 random_vector 327–328
 random walks 486
 rank_words_by_tfidf 371–372
 rank_words function 410
 re.compile 223
 reduce_dimensions 333–335
 regressors 520
 regular expressions (regex)
 221
 re.IGNORECASE 222
 relative likelihood variable 30,
 44
 re library 221
 remove_edge_from method
 507
 reproduced_data array 306
 re.search function 221–223
 ridiculous_measure metric 189
 rotation
 computing principal compo-
 nents without 323–336
 reducing dimensions
 using 297–309
 rotation matrix 298
 rows
 modifying 145–148
 retrieving 143–145
 run_mcl 509, 512
 rv_mean 130

S

sample space 4, 16
 sample_space function 4, 6, 9,
 11–12, 56
 saving table data 148–149
 scikit-learn
 clustering using 175–177
 computing TFIDF vectors
 with 356–358
 dimension reduction
 using 309–315
 efficient dimension reduction
 using 336–338
 grid search using 539–544
 linear classifiers 574–579
 loading online forum discus-
 sions using 341–343
 prediction measurement
 functions 536–537
 training decision tree classifi-
 ers using 614–624

training random forest
 classification 630–632
 vectorizing documents
 using 343–350

SciPy
 determining mean and vari-
 ance of population
 through random
 sampling 103–107
 manipulating normal
 distribution 95–103
 mean as measure of
 centrality 76–85
 relationships between data
 and probability
 using 72–76
 using mean and
 variance 107–113
 computing area beneath
 normal curve 109–112
 interpreting computed
 probability 112–113
 variance as measure of
 dispersion 85–92
 scipy.spatial.distance.euclidean
 function 170
 scipy.stats.gamma.pdf
 distribution 102
 scipy.stats.lognorm.pdf
 distribution 103
 Seaborn 149–152
 search method 223
 second principal
 component 313
 second principal direction 306
 self-loops 454
 SEM (standard error of the
 mean) 108, 116, 121
 sf method 127
 shortest_path function 479
 shortest_path_length
 function 478
 shortest path length
 problem 478
 shuffling cards 54–56
 significance level 119
 silhouette score 180
 similarities 277–280, 282
 simple text comparison 250–262
 Jaccard similarity 255–257
 replacing words with numeric
 values 257–262
 simulations 34–38
 single text cluster 368–372
 skew 126

skills 410
 sklearn.cluster 185
 sklearn.datasets 315, 341, 622
 sklearn.decomposition 309, 336
 sklearn.ensemble 630
 sklearn.feature_extraction 343
 sklearn.linear_model 574
 sklearn.metrics module 536
 sklearn.metrics.silhouette_score
 method 180
 sklearn.preprocessing 351, 361, 583
 sklearn.tree 614–616
 sns.heatmap method 149–152
 social network analysis
 community detection using
 Markov clustering 498–512
 computing travel probabilities using matrix multiplication 489–498
 computing PageRank centrality using NetworkX 496–498
 deriving PageRank centrality from probability theory 492–496
 uncovering central nodes
 based on expected traffic in a network 483–488
 uncovering friend groups in social networks 513–517
 social network data case study
 adding profile features to model 652–657
 data 635–645
 experimental observations 638–641
 friendships linkage table 641–645
 profiles 635–638
 dataset description 447–449
 friendships table 449
 observations table 448–449
 profiles table 447–448
 interpreting trained model 659–663
 optimizing performance across steady set of features 657–659
 overview 449
 problem statement 445–447
 introducing friend-of-a-friend recommendation algorithm 446

predicting user behavior 446–447
 training predictive model using network features 645–652
 soup object 394–395, 399, 402, 408
 sparse matrices 345
 spatial density, grouping data based on 185–190
 spherical coordinates 195
 split method 250
 standard deviation 91
 standardization 558, 562–565
 StandardScaler class 575–576
 standard_scaler object 576
 statistical hypothesis test 115
 statistics
 analysis using SciPy
 mean as measure of centrality 76–85
 relationships between data and probability 72–76
 variance as measure of dispersion 85–92
 determining statistical significance of online ad clicks 161–162
 hypothesis testing
 assessing divergence
 between sample mean and population mean 115–120
 bootstrapping with replacement 124–132
 data dredging 121–124
 permutation testing 132–135
 stats.binom.mean method 83–84
 stats.binom.pmf method 73–75
 stats.binom_test method 72–73
 stats.binom.var method 90
 stats module 72
 stats.norm.cdf method 117, 568
 stats.norm.pdf method 97
 stats.norm.sf method 110
 stats.poisson.pmf distribution 102
 stats.rv_histogram method 127, 131, 135
 stats.skew method 126
 std method 100–101
 stemming 371

stochastic flow 500
 subplots 377–383
 sum of squares 87
 supervised machine learning, network-driven
 basics of supervised machine learning 519–527
 limitations of KNN
 algorithm 544–546
 measuring predicted label accuracy 527–537
 optimizing KNN
 performance 537–539
 running grid search using scikit-learn 539–544
 survival function 110
 SVD (singular value decomposition) 323, 336–338, 341
 svd_object 336–337

T

tables
 exploring using Pandas 138–141
 modifying table rows and columns 145–148
 retrieving table columns 141–143
 retrieving table rows 143–145
 saving and loading data 148–149
 storing tables using basic Python 138
 visualizing using Seaborn 149–152
 Tag class 395
 tag.decompose() 399
 tag_object.decompose() 401
 Tag objects 396, 399–401
 tag_object.text 401
 Tanimoto similarity 261
 tanimoto_similarity
 function 261–262
 term-frequency vector 262
 text 398
 extracting from web pages
 downloading and parsing
 online data 401–403
 parsing HTML using BeautifulSoup 394–401
 structure of HTML documents 386–393

text (*continued*)
 matrix multiplication
 274–287
 basic matrix operations
 277–285
 computational limits
 of 287–290
 computing all-by-all matrix
 similarities 285–287
 NLP analysis of large text
 datasets
 clustering texts by
 topic 363–372
 computing similarities
 across large document
 datasets 358–362
 loading online forum dis-
 cussions using scikit-
 learn 341–343
 ranking words by both post
 frequency and
 count 350–358
 vectorizing documents
 using scikit-learn
 343–350
 visualizing text
 clusters 372–383
 simple text comparison
 250–262
 Jaccard similarity 255–257
 replacing words with
 numeric values 257–262
 vectorizing texts using word
 counts 262–274
 using normalization to
 improve TF vector
 similarity 264–272
 using unit vector dot prod-
 ucts to convert between
 relevance metrics
 272–274
 text vectorization 257
 TF (term-frequency) vectors
 343
 TFIDF (term frequency-inverse
 document frequency) 355
 tfidf_matrix 356, 360, 370, 413
 tfidf_vector 356
 TfidfVectorizer class 356, 383
 tfidf_vectorizer.get_feature_
 names() 356
 TfidfVectorizer object 358
 tf_matrix 345, 349, 352
 tf_vector 349, 356
 time module 284

<title> tag 386–387, 392–393
 to_image methods 374
 tokenization 251
 training
 linear classifiers
 improving perceptron
 performance through
 standardization
 562–565
 improving with logistic
 regression 565–573
 limitations of 582–584
 linearly separating custom-
 ers by size 549–554
 measuring feature impor-
 tance with coefficients
 579–582
 using scikit-learn 574–579
 nonlinear classifiers
 automated learning of logi-
 cal rules 587–614
 decision tree classifier
 limitations 624–626
 random forest
 classification 626–632
 training decision tree classi-
 fiers using scikit-learn
 614–624
 train_logistic function 571–572
 train_nested_if_else function
 598, 610
 train_test_split function
 521–522
 transition matrix 490
 transition_matrix 491, 493, 500
 transition_vector 490
 transpose operation 145
 trapezoidal rule 109
 true positives 530
 TruncatedSVD 336, 338, 360

U

 tag 391–392
 uncertainty 4–8
 undirected graphs 464–480
 computing fastest travel time
 between nodes 473–480
 modeling complex network
 of towns and counties
 467–472
 unidecode output 220
 unions 253
 unit vector dot products
 272–274

unit_vectors 277, 279–280, 282,
 285–286
 unsupervised machine learning
 algorithms 433
 urllib module 401
 urlopen function 401
 user behavior, predicting
 446–447

V

variance 76, 88
 as measure of dispersion
 85–92
 determining through random
 sampling 103–107
 making predictions using
 107–113
 computing area beneath
 normal curve 109–112
 interpreting computed
 probability 112–113
 testing hypothesis with
 unknown 124–132
 var method 89
 var variance 103
 vectorizer object 343, 350,
 358
 vectorizing texts 262–274
 using normalization to
 improve TF vector
 similarity 264–272
 using unit vector dot prod-
 ucts to convert between
 relevance metrics
 272–274
 virtual environments 199
 visualizing maps 201–210

W

web pages
 extracting text from
 downloading and parsing
 online data 401–403
 parsing HTML using Beau-
 tiful Soup 394–401
 structure of HTML
 documents 386–393
 ranking by popularity
 452–464
 weighted_mean 83
 weighted_variance function
 89–90
 where parameter 19, 21

wisdom of the crowd 626
WordCloud class 373–374, 376
wordcloud_image 373–374, 378
word clouds 372
word counts 262–274
 using normalization to
 improve TF vector
 similarity 264–272

using unit vector dot products
 to convert between relevance metrics 272–274
worst concave points 623
worst concavity 621
worst radius feature 623–624,
 627, 631
worst texture feature 623

X

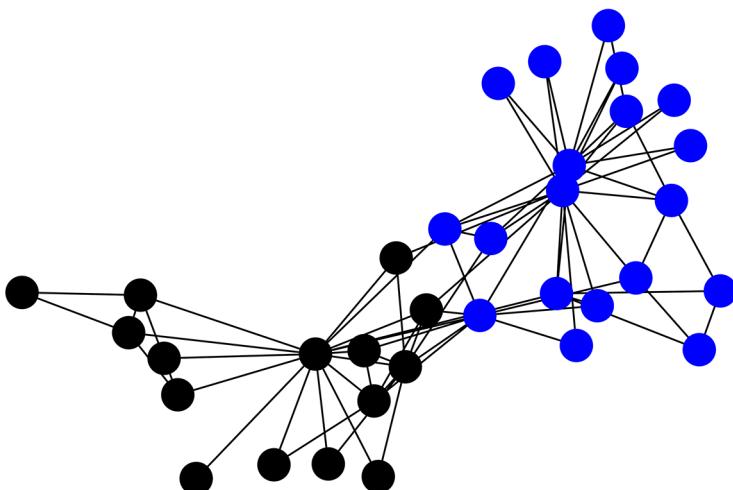
xticklabels parameter 152

Y

y class-label array 540, 645
yticklabels parameter 152

Core Python libraries inside the book

Python library	Use case	First introduced
Matplotlib	Plot visualization	Section 2
NumPy	Large-scale numeric computing	Section 3
SciPy	Statistical analysis	Section 5
Pandas	Tabular data analysis	Section 8
Seaborn	Data visualization	Section 8
Scikit-Learn	Machine learning	Section 10
Cartopy	Map visualization	Section 11
GeoNamesCache	Location name retrieval	Section 11
WordCloud	Text visualization	Section 15
Beautiful Soup	HTML parsing	Section 16
NetworkX	Network analysis	Section 18



A social network is visualized using the NetworkX graph analysis library (see section 18).

Data Science Bookcamp

Leonard Apeltsin

A data science project has a lot of moving parts, and it takes practice and skill to get all the code, algorithms, datasets, formats, and visualizations working together harmoniously. This unique book guides you through five realistic projects, including tracking disease outbreaks from news headlines, analyzing social networks, and finding relevant patterns in ad click data.

Data Science Bookcamp doesn't stop with surface-level theory and toy examples. As you work through each project, you'll learn how to troubleshoot common problems like missing data, messy data, and algorithms that don't quite fit the model you're building. You'll appreciate the detailed setup instructions and the fully explained solutions that highlight common failure points. In the end, you'll be confident in your skills because you can see the results.

What's Inside

- Web scraping
- Organize datasets with clustering algorithms
- Visualize complex multi-variable datasets
- Train a decision tree machine learning algorithm

For readers who know the basics of Python. No prior data science or machine learning skills required.

Leonard Apeltsin is the Head of Data Science at Anomaly, where his team applies advanced analytics to uncover healthcare fraud, waste, and abuse.

Register this print book to get free access to all ebook formats.
Visit <https://www.manning.com/freebook>

“Valuable and accessible...
a solid foundation for anyone
aspiring to be a data scientist.”

—Amaresh Rajasekharan
IBM Corporation

“Really good introduction
of statistical data science
concepts. A must-have for
every beginner!”

—Simone Sguazza
University of Applied Sciences and
Arts of Southern Switzerland

“A full-fledged tutorial in
data science including
common Python libraries
and language tricks!”

—Jean-François Morin
Laval University

“This book is a complete
package for understanding
how the data science process
works end to end.”

—Ayon Roy, Internshala

Free eBook

See first page

ISBN: 978-1-61729-625-3



9 781617 296253

55999



MANNING

\$59.99 / Can \$79.99 [INCLUDING eBOOK]