CHAPTER 20

Clustering

Where we such clusters had
As made us nobly wild, not mad
—Robert Herrick

Most of the algorithms in this book are what's known as *supervised learning* algorithms, in that they start with a set of labeled data and use that as the basis for making predictions about new, unlabeled data. Clustering, however, is an example of *unsupervised learning*, in which we work with completely unlabeled data (or in which our data has labels but we ignore them).

The Idea

Whenever you look at some source of data, it's likely that the data will somehow form *clusters*. A dataset showing where millionaires live probably has clusters in places like Beverly Hills and Manhattan. A dataset showing how many hours people work each week probably has a cluster around 40 (and if it's taken from a state with laws mandating special benefits for people who work at least 20 hours a week, it probably has another cluster right around 19). A dataset of demographics of registered voters likely forms a variety of clusters (e.g., "soccer moms," "bored retirees," "unemployed millennials") that pollsters and political consultants consider relevant.

Unlike some of the problems we've looked at, there is generally no "correct" clustering. An alternative clustering scheme might group some of the "unemployed millennials" with "grad students," and others with "parents' basement dwellers." Neither scheme is necessarily more correct—instead, each is likely more optimal with respect to its own "how good are the clusters?" metric.

Furthermore, the clusters won't label themselves. You'll have to do that by looking at the data underlying each one.

The Model

For us, each `input` will be a vector in d-dimensional space, which, as usual, we will represent as a list of numbers. Our goal will be to identify clusters of similar inputs and (sometimes) to find a representative value for each cluster.

For example, each input could be a numeric vector that represents the title of a blog post, in which case the goal might be to find clusters of similar posts, perhaps in order to understand what our users are blogging about. Or imagine that we have a picture containing thousands of `(red, green, blue)` colors and that we need to screen-print a 10-color version of it. Clustering can help us choose 10 colors that will minimize the total "color error."

One of the simplest clustering methods is k-means, in which the number of clusters k is chosen in advance, after which the goal is to partition the inputs into sets $S_1, ..., S_k$ in a way that minimizes the total sum of squared distances from each point to the mean of its assigned cluster.

There are a lot of ways to assign n points to k clusters, which means that finding an optimal clustering is a very hard problem. We'll settle for an iterative algorithm that usually finds a good clustering:

1. Start with a set of k-means, which are points in d-dimensional space.
2. Assign each point to the mean to which it is closest.
3. If no point's assignment has changed, stop and keep the clusters.
4. If some point's assignment has changed, recompute the means and return to step 2.

Using the `vector_mean` function from Chapter 4, it's pretty simple to create a class that does this.

To start with, we'll create a helper function that measures how many coordinates two vectors differ in. We'll use this to track our training progress:

```python
from scratch.linear_algebra import Vector

def num_differences(v1: Vector, v2: Vector) -> int:
    assert len(v1) == len(v2)
    return len([x1 for x1, x2 in zip(v1, v2) if x1 != x2])

assert num_differences([1, 2, 3], [2, 1, 3]) == 2
assert num_differences([1, 2], [1, 2]) == 0
```

We also need a function that, given some vectors and their assignments to clusters, computes the means of the clusters. It may be the case that some cluster has no points assigned to it. We can't take the mean of an empty collection, so in that case we'll just randomly pick one of the points to serve as the "mean" of that cluster:

```
from typing import List
from scratch.linear_algebra import vector_mean

def cluster_means(k: int,
                  inputs: List[Vector],
                  assignments: List[int]) -> List[Vector]:
    # clusters[i] contains the inputs whose assignment is i
    clusters = [[] for i in range(k)]
    for input, assignment in zip(inputs, assignments):
        clusters[assignment].append(input)

    # if a cluster is empty, just use a random point
    return [vector_mean(cluster) if cluster else random.choice(inputs)
            for cluster in clusters]
```

And now we're ready to code up our clusterer. As usual, we'll use `tqdm` to track our progress, but here we don't know how many iterations it will take, so we then use `itertools.count`, which creates an infinite iterable, and we'll `return` out of it when we're done:

```
import itertools
import random
import tqdm
from scratch.linear_algebra import squared_distance

class KMeans:
    def __init__(self, k: int) -> None:
        self.k = k                      # number of clusters
        self.means = None

    def classify(self, input: Vector) -> int:
        """return the index of the cluster closest to the input"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input, self.means[i]))

    def train(self, inputs: List[Vector]) -> None:
        # Start with random assignments
        assignments = [random.randrange(self.k) for _ in inputs]

        with tqdm.tqdm(itertools.count()) as t:
            for _ in t:
                # Compute means and find new assignments
                self.means = cluster_means(self.k, inputs, assignments)
                new_assignments = [self.classify(input) for input in inputs]

                # Check how many assignments changed and if we're done
                num_changed = num_differences(assignments, new_assignments)
                if num_changed == 0:
                    return
                # Otherwise keep the new assignments, and compute new means
                assignments = new_assignments
                self.means = cluster_means(self.k, inputs, assignments)
                t.set_description(f"changed: {num_changed} / {len(inputs)}")
```

Let's take a look at how this works.

Example: Meetups

To celebrate DataSciencester's growth, your VP of User Rewards wants to organize several in-person meetups for your hometown users, complete with beer, pizza, and DataSciencester t-shirts. You know the locations of all your local users (Figure 20-1), and she'd like you to choose meetup locations that make it convenient for everyone to attend.

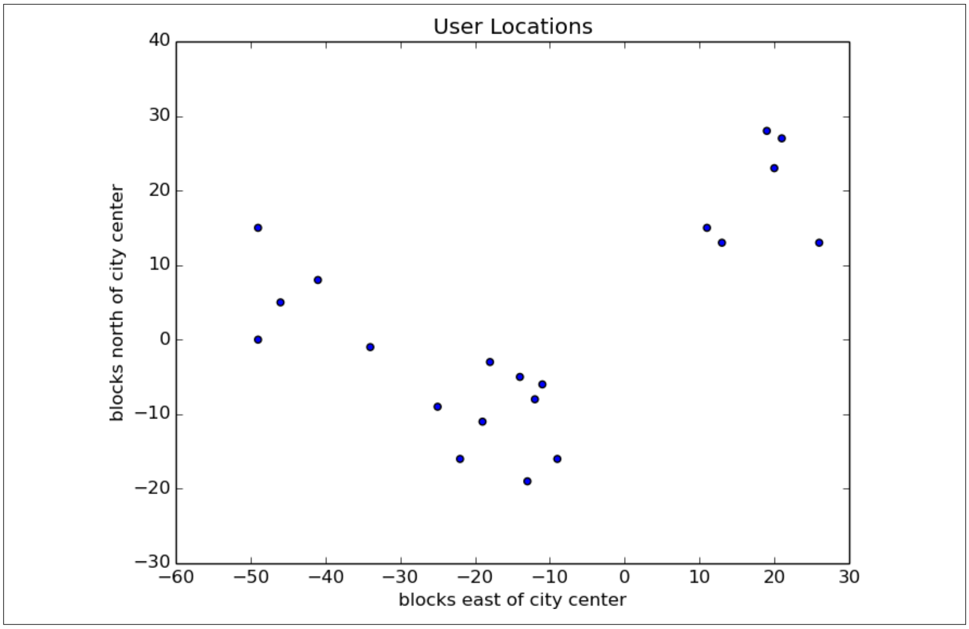


Figure 20-1. The locations of your hometown users

Depending on how you look at it, you probably see two or three clusters. (It's easy to do visually because the data is only two-dimensional. With more dimensions, it would be a lot harder to eyeball.)

Imagine first that she has enough budget for three meetups. You go to your computer and try this:

```
random.seed(12)                   # so you get the same results as me
clusterer = KMeans(k=3)
```

```
clusterer.train(inputs)
means = sorted(clusterer.means)   # sort for the unit test

assert len(means) == 3

# Check that the means are close to what we expect
assert squared_distance(means[0], [-44, 5]) < 1
assert squared_distance(means[1], [-16, -10]) < 1
assert squared_distance(means[2], [18, 20]) < 1
```

You find three clusters centered at [–44, 5], [–16, –10], and [18, 20], and you look for meetup venues near those locations (Figure 20-2).

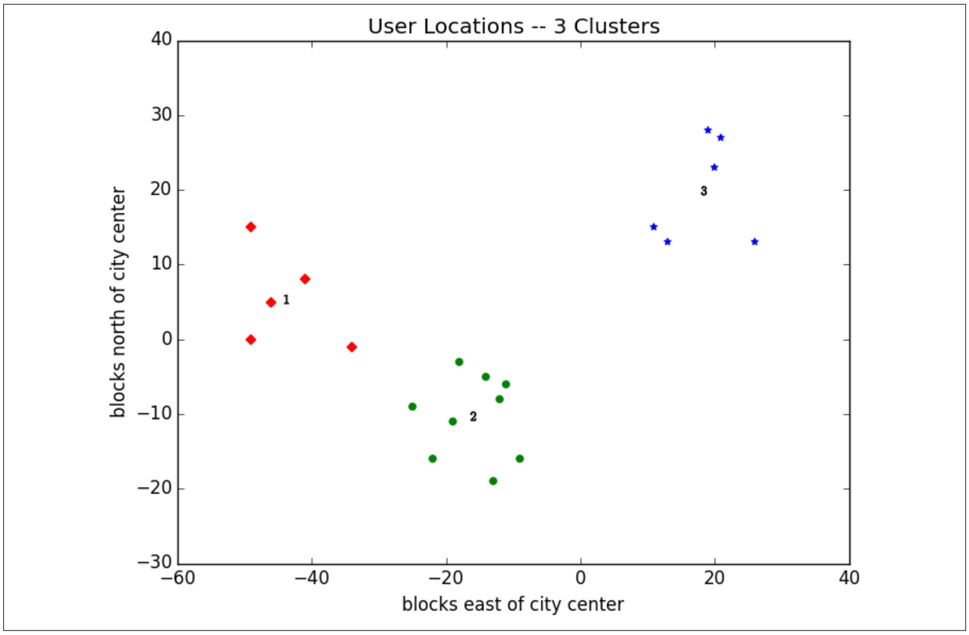


Figure 20-2. User locations grouped into three clusters

You show your results to the VP, who informs you that now she only has enough budgeted for *two* meetups.

"No problem," you say:

```
random.seed(0)
clusterer = KMeans(k=2)
clusterer.train(inputs)
means = sorted(clusterer.means)

assert len(means) == 2
assert squared_distance(means[0], [-26, -5]) < 1
assert squared_distance(means[1], [18, 20]) < 1
```

As shown in Figure 20-3, one meetup should still be near [18, 20], but now the other should be near [–26, –5].

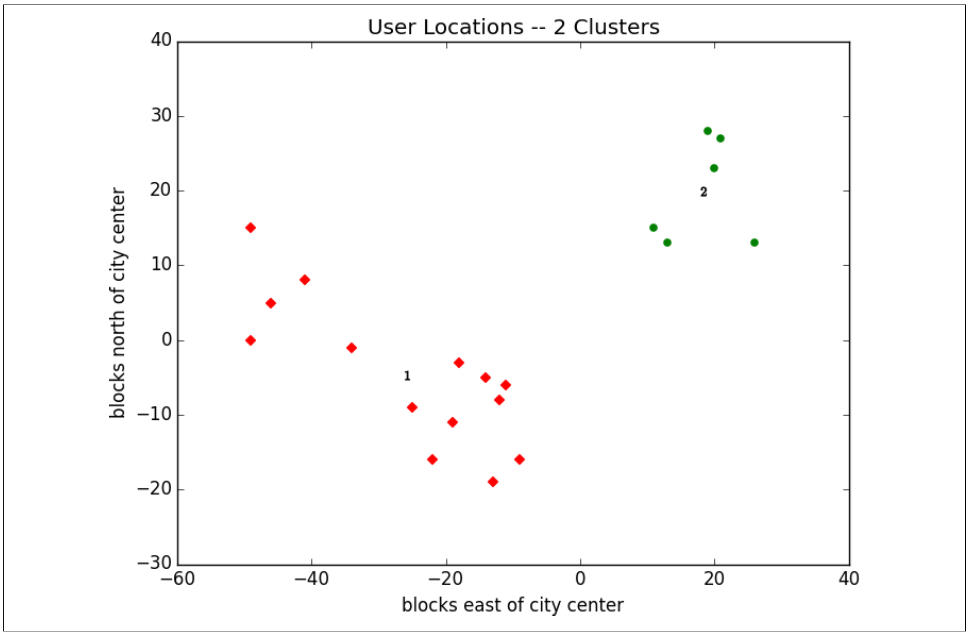


Figure 20-3. User locations grouped into two clusters

Choosing k

In the previous example, the choice of *k* was driven by factors outside of our control. In general, this won't be the case. There are various ways to choose a *k*. One that's reasonably easy to understand involves plotting the sum of squared errors (between each point and the mean of its cluster) as a function of *k* and looking at where the graph "bends":

```
from matplotlib import pyplot as plt

def squared_clustering_errors(inputs: List[Vector], k: int) -> float:
    """finds the total squared error from k-means clustering the inputs"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = [clusterer.classify(input) for input in inputs]

    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs, assignments))
```

which we can apply to our previous example:

```
# now plot from 1 up to len(inputs) clusters

ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]

plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("total squared error")
plt.title("Total Error vs. # of Clusters")
plt.show()
```

Looking at Figure 20-4, this method agrees with our original eyeballing that three is the "right" number of clusters.

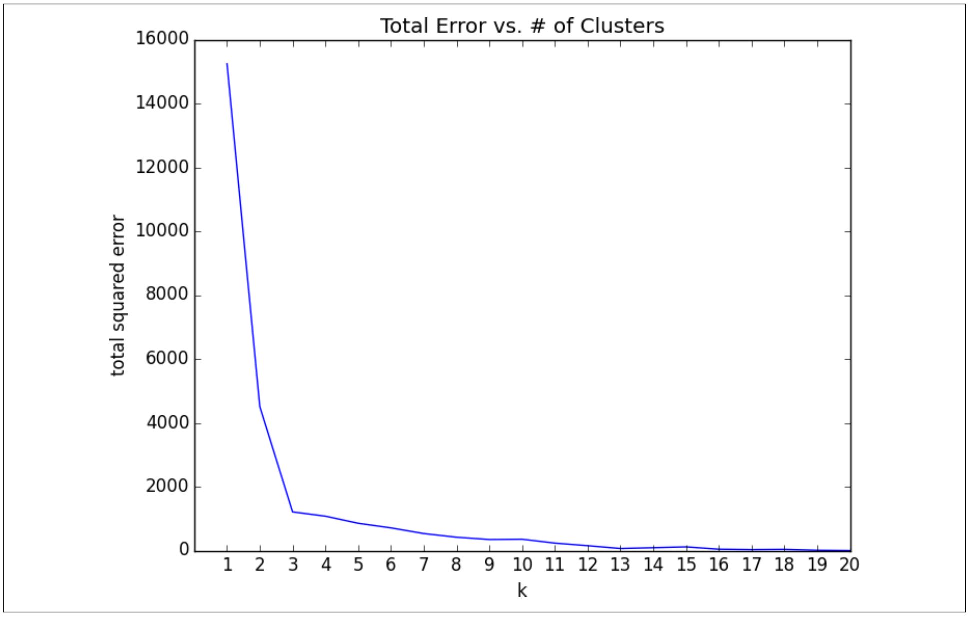


Figure 20-4. Choosing a k

Example: Clustering Colors

The VP of Swag has designed attractive DataSciencester stickers that he'd like you to hand out at meetups. Unfortunately, your sticker printer can print at most five colors per sticker. And since the VP of Art is on sabbatical, the VP of Swag asks if there's some way you can take his design and modify it so that it contains only five colors.

Computer images can be represented as two-dimensional arrays of pixels, where each pixel is itself a three-dimensional vector `(red, green, blue)` indicating its color.

Creating a five-color version of the image, then, entails:

1. Choosing five colors.
2. Assigning one of those colors to each pixel.

It turns out this is a great task for *k*-means clustering, which can partition the pixels into five clusters in red-green-blue space. If we then recolor the pixels in each cluster to the mean color, we're done.

To start with, we'll need a way to load an image into Python. We can do this with matplotlib, if we first install the pillow library:

```
python -m pip install pillow
```

Then we can just use `matplotlib.image.imread`:

```
image_path = r"girl_with_book.jpg"    # wherever your image is
import matplotlib.image as mpimg
img = mpimg.imread(image_path) / 256  # rescale to between 0 and 1
```

Behind the scenes `img` is a NumPy array, but for our purposes, we can treat it as a list of lists of lists.

`img[i][j]` is the pixel in the *i*th row and *j*th column, and each pixel is a list `[red, green, blue]` of numbers between 0 and 1 indicating the color of that pixel:

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

In particular, we can get a flattened list of all the pixels as:

```
# .tolist() converts a NumPy array to a Python list
pixels = [pixel.tolist() for row in img for pixel in row]
```

and then feed them to our clusterer:

```
clusterer = KMeans(5)
clusterer.train(pixels)   # this might take a while
```

Once it finishes, we just construct a new image with the same format:

```
def recolor(pixel: Vector) -> Vector:
    cluster = clusterer.classify(pixel)        # index of the closest cluster
    return clusterer.means[cluster]            # mean of the closest cluster

new_img = [[recolor(pixel) for pixel in row]   # recolor this row of pixels
           for row in img]                     # for each row in the image
```

and display it, using `plt.imshow`:

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

It is difficult to show color results in a black-and-white book, but Figure 20-5 shows grayscale versions of a full-color picture and the output of using this process to reduce it to five colors.

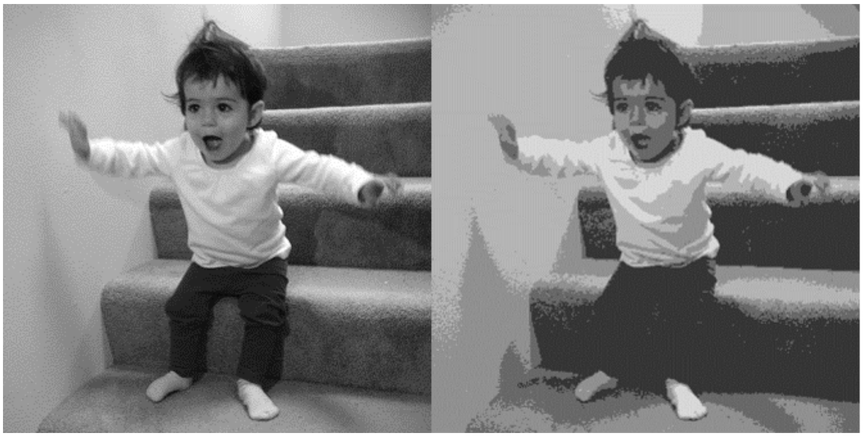


Figure 20-5. Original picture and its 5-means decoloring

Bottom-Up Hierarchical Clustering

An alternative approach to clustering is to "grow" clusters from the bottom up. We can do this in the following way:

1. Make each input its own cluster of one.
2. As long as there are multiple clusters remaining, find the two closest clusters and merge them.

At the end, we'll have one giant cluster containing all the inputs. If we keep track of the merge order, we can re-create any number of clusters by unmerging. For example, if we want three clusters, we can just undo the last two merges.

We'll use a really simple representation of clusters. Our values will live in *leaf* clusters, which we will represent as `NamedTuple`s:

```
from typing import NamedTuple, Union

class Leaf(NamedTuple):
    value: Vector

leaf1 = Leaf([10,  20])
leaf2 = Leaf([30, -15])
```

We'll use these to grow *merged* clusters, which we will also represent as `NamedTuple`s:

```
class Merged(NamedTuple):
    children: tuple
    order: int

merged = Merged((leaf1, leaf2), order=1)

Cluster = Union[Leaf, Merged]
```

This is another case where Python's type annotations have let us down. You'd like to type hint `Merged.children` as `Tuple[Cluster, Cluster]` but `mypy` doesn't allow recursive types like that.

We'll talk about merge order in a bit, but first let's create a helper function that recursively returns all the values contained in a (possibly merged) cluster:

```
def get_values(cluster: Cluster) -> List[Vector]:
    if isinstance(cluster, Leaf):
        return [cluster.value]
    else:
        return [value
                for child in cluster.children
                for value in get_values(child)]

assert get_values(merged) == [[10, 20], [30, -15]]
```

In order to merge the closest clusters, we need some notion of the distance between clusters. We'll use the *minimum* distance between elements of the two clusters, which merges the two clusters that are closest to touching (but will sometimes produce large chain-like clusters that aren't very tight). If we wanted tight spherical clusters, we might use the *maximum* distance instead, as it merges the two clusters that fit in the smallest ball. Both are common choices, as is the *average* distance:

```
from typing import Callable
from scratch.linear_algebra import distance

def cluster_distance(cluster1: Cluster,
                     cluster2: Cluster,
                     distance_agg: Callable = min) -> float:
    """
    compute all the pairwise distances between cluster1 and cluster2
    and apply the aggregation function _distance_agg_ to the resulting list
    """
    return distance_agg([distance(v1, v2)
                         for v1 in get_values(cluster1)
                         for v2 in get_values(cluster2)])
```

We'll use the merge order slot to track the order in which we did the merging. Smaller numbers will represent *later* merges. This means when we want to unmerge clusters, we do so from lowest merge order to highest. Since `Leaf` clusters were never merged, we'll assign them infinity, the highest possible value. And since they don't have an `.order` property, we'll create a helper function:

```
def get_merge_order(cluster: Cluster) -> float:
    if isinstance(cluster, Leaf):
        return float('inf')  # was never merged
    else:
        return cluster.order
```

Similarly, since `Leaf` clusters don't have children, we'll create and add a helper function for that:

```
from typing import Tuple

def get_children(cluster: Cluster):
    if isinstance(cluster, Leaf):
        raise TypeError("Leaf has no children")
    else:
        return cluster.children
```

Now we're ready to create the clustering algorithm:

```
def bottom_up_cluster(inputs: List[Vector],
                      distance_agg: Callable = min) -> Cluster:
    # Start with all leaves
    clusters: List[Cluster] = [Leaf(input) for input in inputs]

    def pair_distance(pair: Tuple[Cluster, Cluster]) -> float:
        return cluster_distance(pair[0], pair[1], distance_agg)

    # as long as we have more than one cluster left...
    while len(clusters) > 1:
        # find the two closest clusters
        c1, c2 = min(((cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]),
                      key=pair_distance)

        # remove them from the list of clusters
        clusters = [c for c in clusters if c != c1 and c != c2]

        # merge them, using merge_order = # of clusters left
        merged_cluster = Merged((c1, c2), order=len(clusters))

        # and add their merge
        clusters.append(merged_cluster)

    # when there's only one cluster left, return it
    return clusters[0]
```

Its use is very simple:

```
    base_cluster = bottom_up_cluster(inputs)
```

This produces a clustering that looks as follows:

```
  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
──┬──┬─────┬────────────────────────────────┬───────────┬─ [19, 28]
  │  │     │                                │           └─ [21, 27]
  │  │     │                                └─ [20, 23]
  │  │     └─ [26, 13]
  │  └────────────────────────────────────────────┬─ [11, 15]
  │                                               └─ [13, 13]
  └─────┬─────┬──┬───────────┬─────┬─ [-49, 0]
        │     │  │           │     └─ [-46, 5]
        │     │  │           └─ [-41, 8]
        │     │  └─ [-49, 15]
        │     └─ [-34, 1]
        └───────────┬──┬──┬─────┬─ [-22, -16]
                    │  │  │     └─ [-19, -11]
                    │  │  └─ [-25, -9]
                    │  └─────────────────┬─────┬─────┬─ [-11, -6]
                    │                    │     │     └─ [-12, -8]
                    │                    │     └─ [-14, 5]
                    │                    └─ [-18, -3]
                    └─────────────────┬─ [-13, -19]
                                      └─ [-9, -16]
```

The numbers at the top indicate "merge order." Since we had 20 inputs, it took 19 merges to get to this one cluster. The first merge created cluster 18 by combining the leaves [19, 28] and [21, 27]. And the last merge created cluster 0.

If you wanted only two clusters, you'd split at the first fork ("0"), creating one cluster with six points and a second with the rest. For three clusters, you'd continue to the second fork ("1"), which indicates to split that first cluster into the cluster with ([19, 28], [21, 27], [20, 23], [26, 13]) and the cluster with ([11, 15], [13, 13]). And so on.

Generally, though, we don't want to be squinting at nasty text representations like this. Instead, let's write a function that generates any number of clusters by performing the appropriate number of unmerges:

```
    def generate_clusters(base_cluster: Cluster,
                          num_clusters: int) -> List[Cluster]:
        # start with a list with just the base cluster
        clusters = [base_cluster]

        # as long as we don't have enough clusters yet...
        while len(clusters) < num_clusters:
            # choose the last-merged of our clusters
            next_cluster = min(clusters, key=get_merge_order)
            # remove it from the list
            clusters = [c for c in clusters if c != next_cluster]

            # and add its children to the list (i.e., unmerge it)
            clusters.extend(get_children(next_cluster))

        # once we have enough clusters...
        return clusters
```

So, for example, if we want to generate three clusters, we can just do:

```
    three_clusters = [get_values(cluster)
                      for cluster in generate_clusters(base_cluster, 3)]
```

which we can easily plot:

```
    for i, cluster, marker, color in zip([1, 2, 3],
                                         three_clusters,
                                         ['D','o','*'],
                                         ['r','g','b']):
        xs, ys = zip(*cluster)  # magic unzipping trick
        plt.scatter(xs, ys, color=color, marker=marker)

        # put a number at the mean of the cluster
        x, y = vector_mean(cluster)
        plt.plot(x, y, marker='$' + str(i) + '$', color='black')

    plt.title("User Locations -- 3 Bottom-Up Clusters, Min")
    plt.xlabel("blocks east of city center")
    plt.ylabel("blocks north of city center")
    plt.show()
```

This gives very different results than *k*-means did, as shown in Figure 20-6.

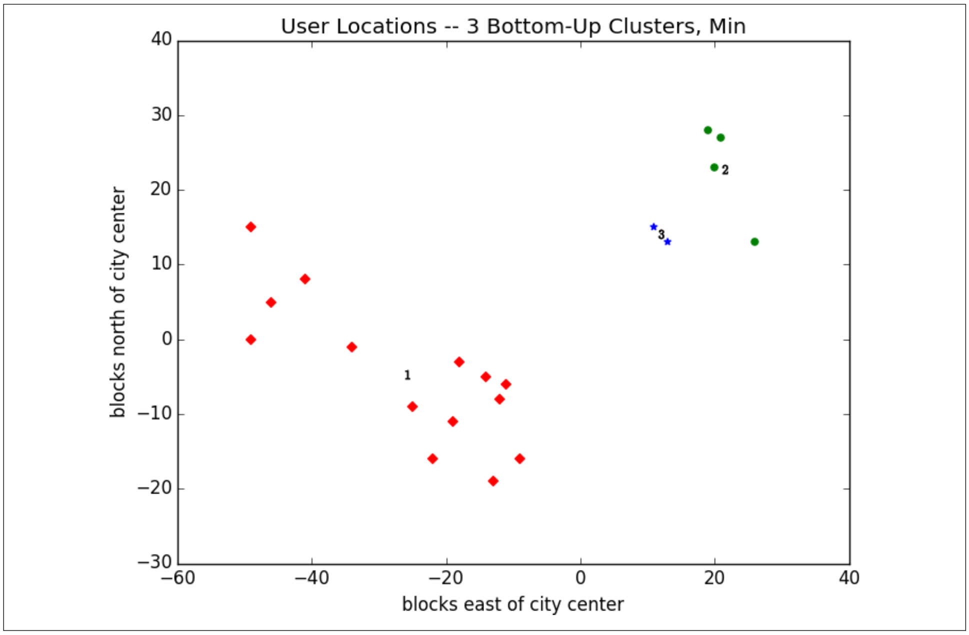


Figure 20-6. Three bottom-up clusters using min distance

As mentioned previously, this is because using `min` in `cluster_distance` tends to give chain-like clusters. If we instead use `max` (which gives tight clusters), it looks the same as the 3-means result (Figure 20-7).

> The previous `bottom_up_clustering` implementation is relatively simple, but also shockingly inefficient. In particular, it recomputes the distance between each pair of inputs at every step. A more efficient implementation might instead precompute the distances between each pair of inputs and then perform a lookup inside `cluster_distance`. A *really* efficient implementation would likely also remember the `cluster_distances` from the previous step.

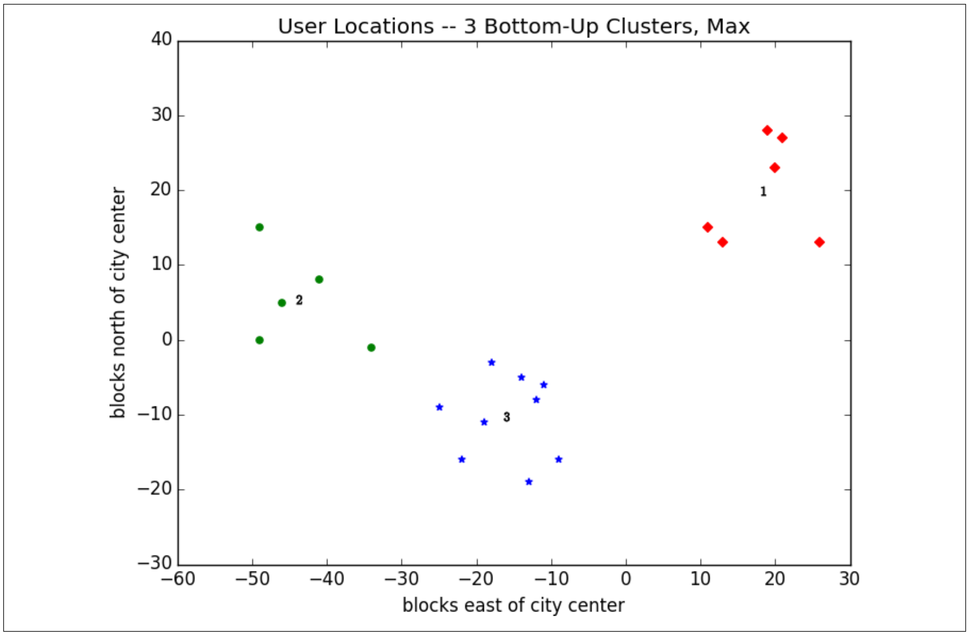


Figure 20-7. Three bottom-up clusters using max distance

For Further Exploration

- scikit-learn has an entire module, `sklearn.cluster`, that contains several clustering algorithms including `KMeans` and the `Ward` hierarchical clustering algorithm (which uses a different criterion for merging clusters than ours did).
- SciPy has two clustering models: `scipy.cluster.vq`, which does *k*-means, and `scipy.cluster.hierarchy`, which has a variety of hierarchical clustering algorithms.

CHAPTER 47

In Depth: k-Means Clustering

In the previous chapters we explored unsupervised machine learning models for dimensionality reduction. Now we will move on to another class of unsupervised machine learning models: clustering algorithms. Clustering algorithms seek to learn, from the properties of the data, an optimal division or discrete labeling of groups of points.

Many clustering algorithms are available in Scikit-Learn and elsewhere, but perhaps the simplest to understand is an algorithm known as *k-means clustering*, which is implemented in `sklearn.cluster.KMeans`.

We begin with the standard imports:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Introducing k-Means

The *k*-means algorithm searches for a predetermined number of clusters within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like:

- The *cluster center* is the arithmetic mean of all the points belonging to the cluster.
- Each point is closer to its own cluster center than to other cluster centers.

Those two assumptions are the basis of the *k*-means model. We will soon dive into exactly *how* the algorithm reaches this solution, but for now let's take a look at a simple dataset and see the *k*-means result.

First, let's generate a two-dimensional dataset containing four distinct blobs. To emphasize that this is an unsupervised algorithm, we will leave the labels out of the visualization (see Figure 47-1).

```
In [2]: from sklearn.datasets import make_blobs
        X, y_true = make_blobs(n_samples=300, centers=4,
                               cluster_std=0.60, random_state=0)
        plt.scatter(X[:, 0], X[:, 1], s=50);
```

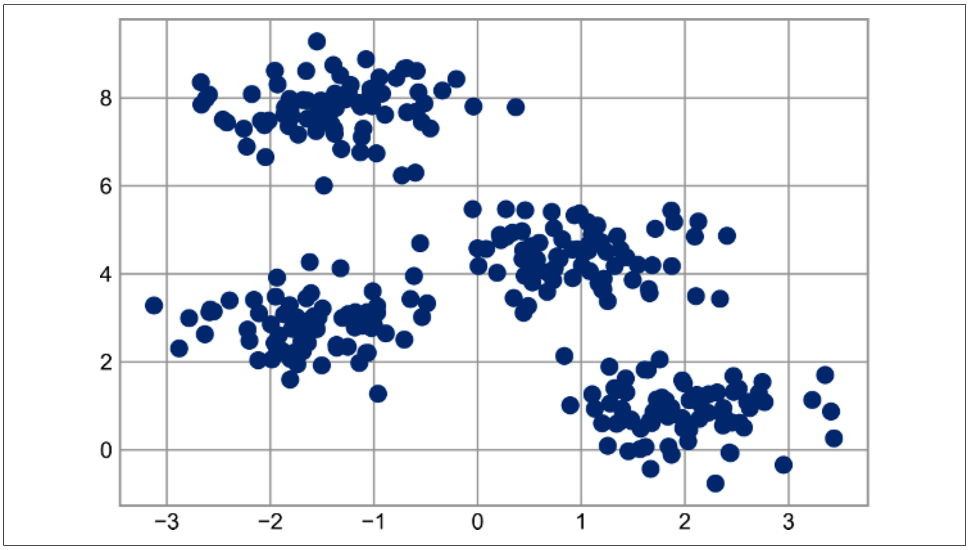


Figure 47-1. Data for demonstration of clustering

By eye, it is relatively easy to pick out the four clusters. The *k*-means algorithm does this automatically, and in Scikit-Learn uses the typical estimator API:

```
In [3]: from sklearn.cluster import KMeans
        kmeans = KMeans(n_clusters=4)
        kmeans.fit(X)
        y_kmeans = kmeans.predict(X)
```

Let's visualize the results by plotting the data colored by these labels (Figure 47-2). We will also plot the cluster centers as determined by the *k*-means estimator:

```
In [4]: plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

        centers = kmeans.cluster_centers_
        plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200);
```

The good news is that the *k*-means algorithm (at least in this simple case) assigns the points to clusters very similarly to how we might assign them by eye. But you might wonder how this algorithm finds these clusters so quickly: after all, the number of possible combinations of cluster assignments is exponential in the number of data points—an exhaustive search would be very, very costly. Fortunately for us, such an exhaustive search is not necessary: instead, the typical approach to *k*-means involves an intuitive iterative approach known as *expectation–maximization*.

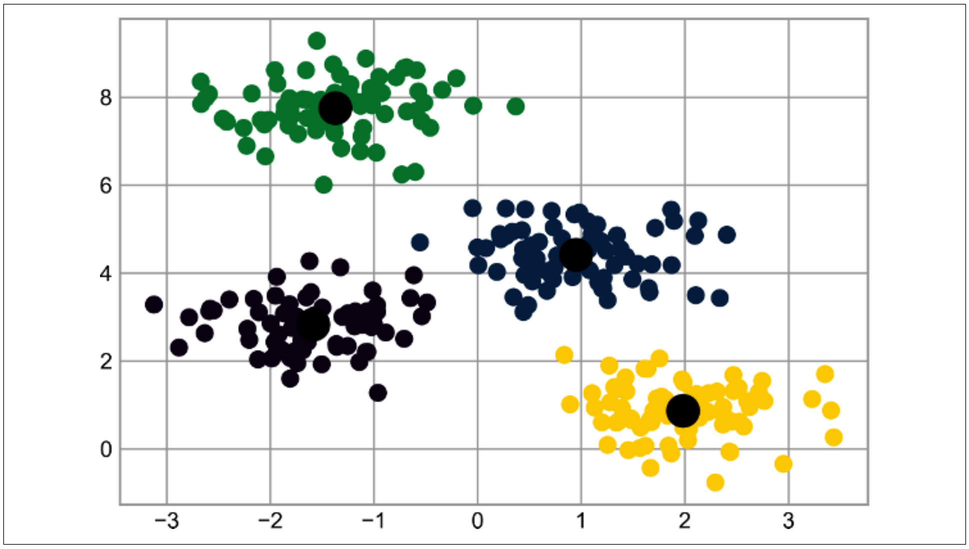


Figure 47-2. k-*means cluster centers with clusters indicated by color*

Expectation–Maximization

Expectation–maximization (E–M) is a powerful algorithm that comes up in a variety of contexts within data science. *k*-means is a particularly simple and easy-to-understand application of the algorithm; we'll walk through it briefly here. In short, the expectation–maximization approach here consists of the following procedure:

1. Guess some cluster centers.
2. Repeat until converged:
    a. *E-step*: Assign points to the nearest cluster center.
    b. *M-step*: Set the cluster centers to the mean of their assigned points.

Here the *E-step* or *expectation step* is so named because it involves updating our expectation of which cluster each point belongs to. The *M-step* or *maximization step* is so named because it involves maximizing some fitness function that defines the locations of the cluster centers—in this case, that maximization is accomplished by taking a simple mean of the data in each cluster.

The literature about this algorithm is vast, but can be summarized as follows: under typical circumstances, each repetition of the E-step and M-step will always result in a better estimate of the cluster characteristics.

We can visualize the algorithm as shown in Figure 47-3. For the particular initialization shown here, the clusters converge in just three iterations. (For an interactive version of this figure, refer to the code in the online appendix.)

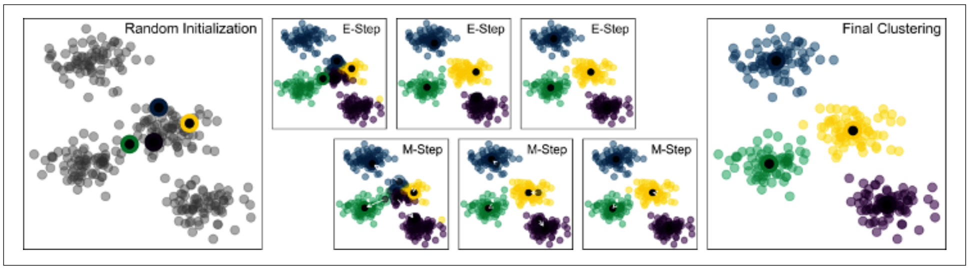


Figure 47-3. Visualization of the E–M algorithm for k-*means*[1]

The *k*-means algorithm is simple enough that we can write it in a few lines of code. The following is a very basic implementation (see Figure 47-4).

```
In [5]: from sklearn.metrics import pairwise_distances_argmin

        def find_clusters(X, n_clusters, rseed=2):
            # 1. Randomly choose clusters
            rng = np.random.RandomState(rseed)
            i = rng.permutation(X.shape[0])[:n_clusters]
            centers = X[i]

            while True:
                # 2a. Assign labels based on closest center
                labels = pairwise_distances_argmin(X, centers)

                # 2b. Find new centers from means of points
                new_centers = np.array([X[labels == i].mean(0)
                                        for i in range(n_clusters)])

                # 2c. Check for convergence
                if np.all(centers == new_centers):
                    break
                centers = new_centers

            return centers, labels

        centers, labels = find_clusters(X, 4)
```

1 Code to produce this figure can be found in the online appendix.

```
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```

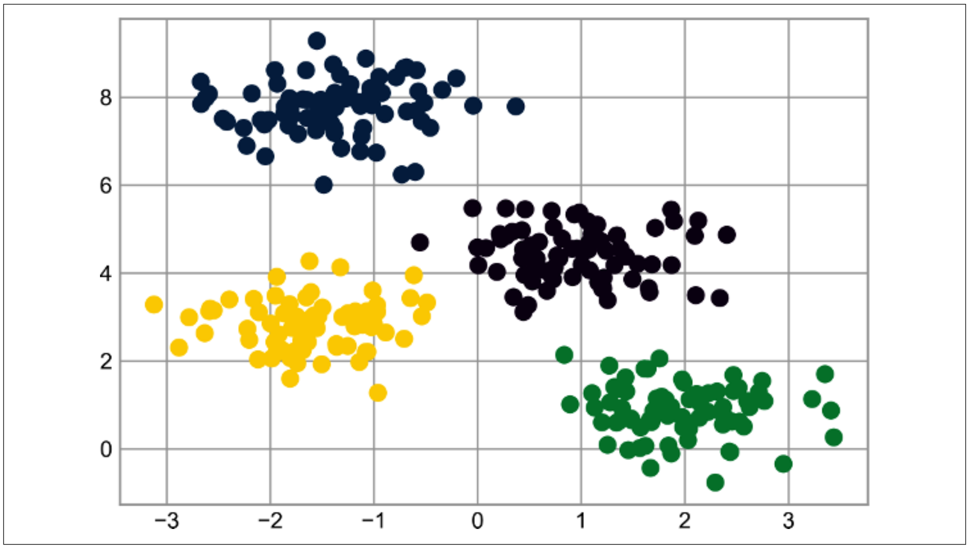


Figure 47-4. Data labeled with k*-means*

Most well-tested implementations will do a bit more than this under the hood, but the preceding function gives the gist of the expectation–maximization approach. There are a few caveats to be aware of when using the expectation–maximization algorithm:

The globally optimal result may not be achieved

First, although the E–M procedure is guaranteed to improve the result in each step, there is no assurance that it will lead to the *global* best solution. For example, if we use a different random seed in our simple procedure, the particular starting guesses lead to poor results (see Figure 47-5).

```
In [6]: centers, labels = find_clusters(X, 4, rseed=0)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

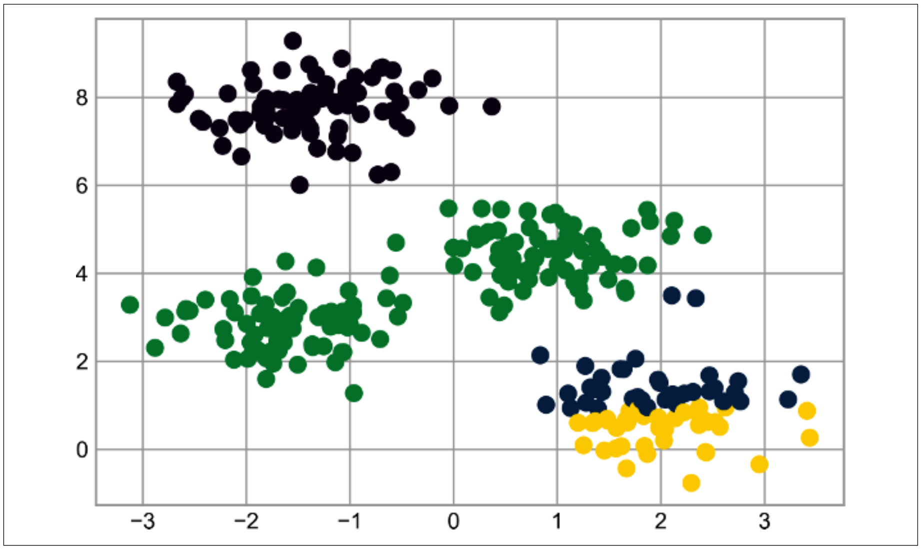


Figure 47-5. An example of poor convergence in k*-means*

Here the E–M approach has converged, but has not converged to a globally optimal configuration. For this reason, it is common for the algorithm to be run for multiple starting guesses, as indeed Scikit-Learn does by default (the number is set by the `n_init` parameter, which defaults to 10).

The number of clusters must be selected beforehand

Another common challenge with *k*-means is that you must tell it how many clusters you expect: it cannot learn the number of clusters from the data. For example, if we ask the algorithm to identify six clusters, it will happily proceed and find the best six clusters, as shown in Figure 40-1:

```
In [7]: labels = KMeans(6, random_state=0).fit_predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

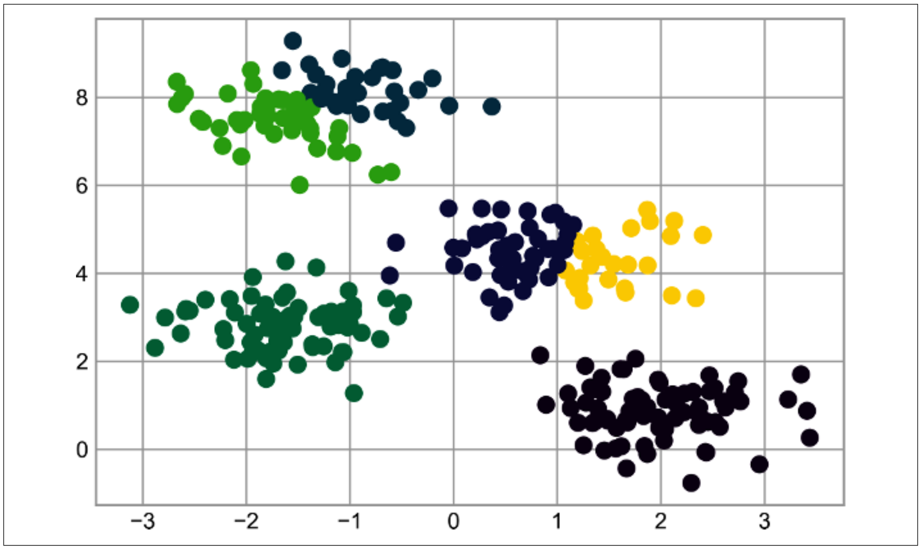


Figure 47-6. An example where the number of clusters is chosen poorly

Whether the result is meaningful is a question that is difficult to answer definitively; one approach that is rather intuitive, but that we won't discuss further here, is called silhouette analysis.

Alternatively, you might use a more complicated clustering algorithm that has a better quantitative measure of the fitness per number of clusters (e.g., Gaussian mixture models; see Chapter 48) or which *can* choose a suitable number of clusters (e.g., DBSCAN, mean-shift, or affinity propagation, all available in the `sklearn.cluster` submodule).

k-*means is limited to linear cluster boundaries*

The fundamental model assumptions of *k*-means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters have complicated geometries.

In particular, the boundaries between *k*-means clusters will always be linear, which means that it will fail for more complicated boundaries. Consider the following data, along with the cluster labels found by the typical *k*-means approach (see Figure 47-7).

```
In [8]: from sklearn.datasets import make_moons
        X, y = make_moons(200, noise=.05, random_state=0)
```

```
In [9]: labels = KMeans(2, random_state=0).fit_predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

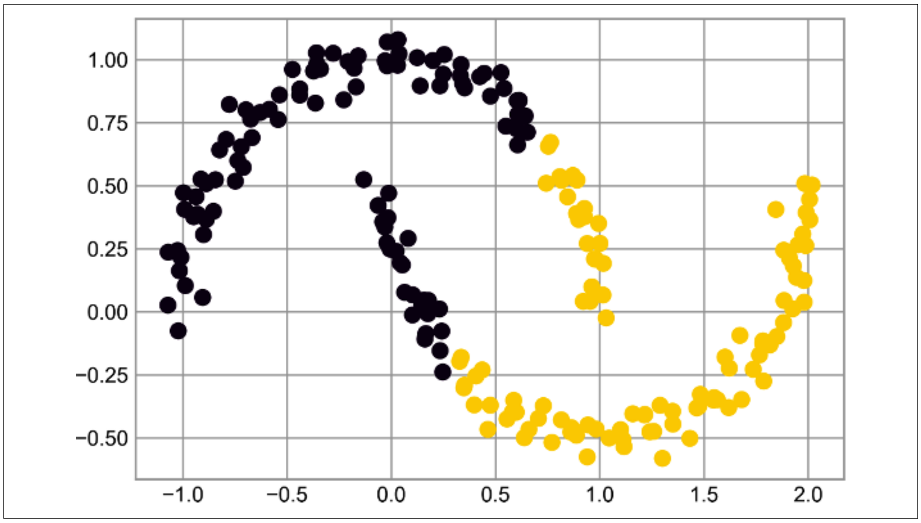


Figure 47-7. Failure of k-*means with nonlinear boundaries*

This situation is reminiscent of the discussion in Chapter 43, where we used a kernel transformation to project the data into a higher dimension where a linear separation is possible. We might imagine using the same trick to allow *k*-means to discover non-linear boundaries.

One version of this kernelized *k*-means is implemented in Scikit-Learn within the `SpectralClustering` estimator. It uses the graph of nearest neighbors to compute a higher-dimensional representation of the data, and then assigns labels using a *k*-means algorithm (see Figure 47-8).

```
In [10]: from sklearn.cluster import SpectralClustering
         model = SpectralClustering(n_clusters=2,
                                    affinity='nearest_neighbors',
                                    assign_labels='kmeans')
         labels = model.fit_predict(X)
         plt.scatter(X[:, 0], X[:, 1], c=labels,
                     s=50, cmap='viridis');
```

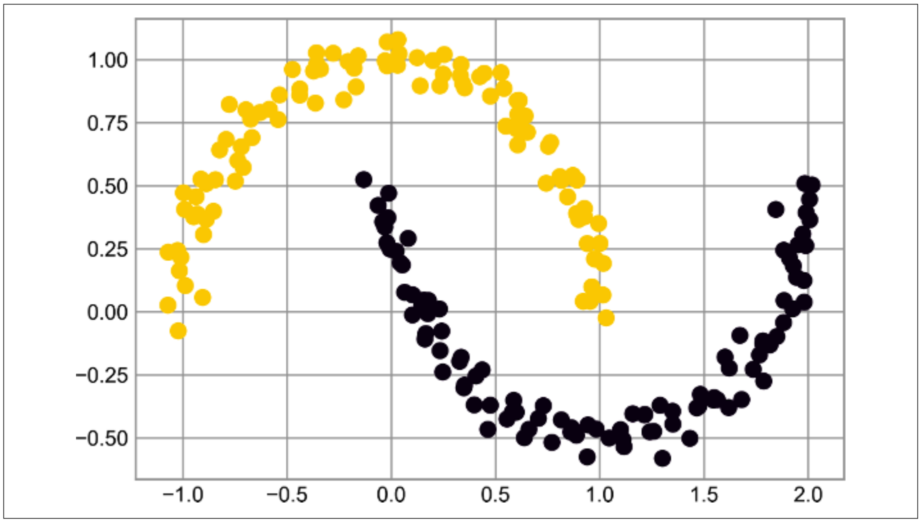


Figure 47-8. Nonlinear boundaries learned by SpectralClustering

We see that with this kernel transform approach, the kernelized *k*-means is able to find the more complicated nonlinear boundaries between clusters.

k-*means can be slow for large numbers of samples*

Because each iteration of *k*-means must access every point in the dataset, the algorithm can be relatively slow as the number of samples grows. You might wonder if this requirement to use all data at each iteration can be relaxed; for example, you might just use a subset of the data to update the cluster centers at each step. This is the idea behind batch-based *k*-means algorithms, one form of which is implemented in `sklearn.cluster.MiniBatchKMeans`. The interface for this is the same as for standard `KMeans`; we will see an example of its use as we continue our discussion.

Examples

Being careful about these limitations of the algorithm, we can use *k*-means to our advantage in a variety of situations. We'll now take a look at a couple of examples.

Example 1: k-Means on Digits

To start, let's take a look at applying *k*-means on the same simple digits data that we saw in Chapters 44 and 45. Here we will attempt to use *k*-means to try to identify similar digits *without using the original label information*; this might be similar to a first step in extracting meaning from a new dataset about which you don't have any *a priori* label information.

We will start by loading the dataset, then find the clusters. Recall that the digits dataset consists of 1,797 samples with 64 features, where each of the 64 features is the brightness of one pixel in an 8 × 8 image:

```
In [11]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.data.shape
Out[11]: (1797, 64)
```

The clustering can be performed as we did before:

```
In [12]: kmeans = KMeans(n_clusters=10, random_state=0)
         clusters = kmeans.fit_predict(digits.data)
         kmeans.cluster_centers_.shape
Out[12]: (10, 64)
```

The result is 10 clusters in 64 dimensions. Notice that the cluster centers themselves are 64-dimensional points, and can be interpreted as representing the "typical" digit within the cluster. Let's see what these cluster centers look like (see Figure 47-9).

```
In [13]: fig, ax = plt.subplots(2, 5, figsize=(8, 3))
         centers = kmeans.cluster_centers_.reshape(10, 8, 8)
         for axi, center in zip(ax.flat, centers):
             axi.set(xticks=[], yticks=[])
             axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```

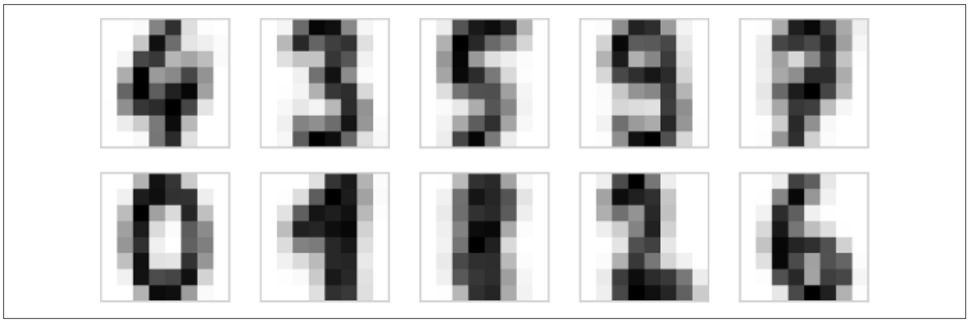


Figure 47-9. Cluster centers learned by k*-means*

We see that *even without the labels*, `KMeans` is able to find clusters whose centers are recognizable digits, with perhaps the exception of 1 and 8.

Because *k*-means knows nothing about the identities of the clusters, the 0–9 labels may be permuted. We can fix this by matching each learned cluster label with the true labels found in the clusters:

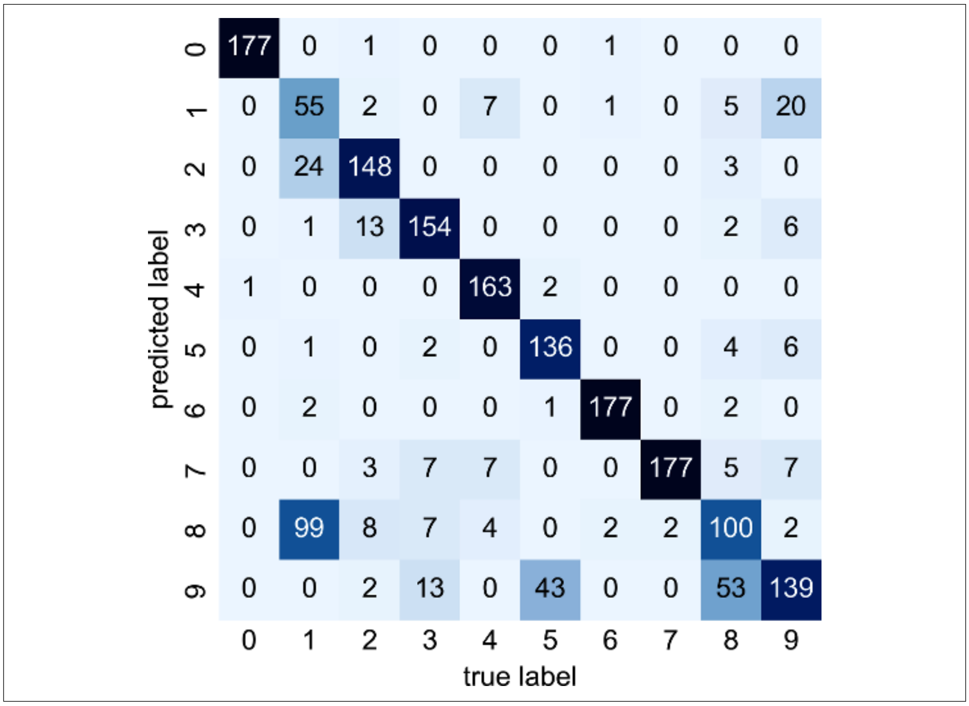
```
In [14]: from scipy.stats import mode

         labels = np.zeros_like(clusters)
         for i in range(10):
             mask = (clusters == i)
             labels[mask] = mode(digits.target[mask])[0]
```

Now we can check how accurate our unsupervised clustering was in finding similar digits within the data:

```
In [15]: from sklearn.metrics import accuracy_score
         accuracy_score(digits.target, labels)
Out[15]: 0.7935447968836951
```

With just a simple *k*-means algorithm, we discovered the correct grouping for 80% of the input digits! Let's check the confusion matrix for this, visualized in Figure 47-10.

```
In [16]: from sklearn.metrics import confusion_matrix
         import seaborn as sns
         mat = confusion_matrix(digits.target, labels)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d',
                     cbar=False, cmap='Blues',
                     xticklabels=digits.target_names,
                     yticklabels=digits.target_names)
         plt.xlabel('true label')
         plt.ylabel('predicted label');
```



predicted label \ true label	0	1	2	3	4	5	6	7	8	9
0	177	0	1	0	0	0	1	0	0	0
1	0	55	2	0	7	0	1	0	5	20
2	0	24	148	0	0	0	0	0	3	0
3	0	1	13	154	0	0	0	0	2	6
4	1	0	0	0	163	2	0	0	0	0
5	0	1	0	2	0	136	0	0	4	6
6	0	2	0	0	0	1	177	0	2	0
7	0	0	3	7	7	0	0	177	5	7
8	0	99	8	7	4	0	2	2	100	2
9	0	0	2	13	0	43	0	0	53	139

Figure 47-10. Confusion matrix for the k*-means classifier*

As we might expect from the cluster centers we visualized before, the main point of confusion is between the eights and ones. But this still shows that using *k*-means, we can essentially build a digit classifier *without reference to any known labels*!

Just for fun, let's try to push this even farther. We can use the t-distributed stochastic neighbor embedding algorithm (mentioned in Chapter 46) to preprocess the data before performing *k*-means. t-SNE is a nonlinear embedding algorithm that is particularly adept at preserving points within clusters. Let's see how it does:

```
In [17]: from sklearn.manifold import TSNE

         # Project the data: this step will take several seconds
         tsne = TSNE(n_components=2, init='random',
                     learning_rate='auto',random_state=0)
         digits_proj = tsne.fit_transform(digits.data)

         # Compute the clusters
         kmeans = KMeans(n_clusters=10, random_state=0)
         clusters = kmeans.fit_predict(digits_proj)

         # Permute the labels
         labels = np.zeros_like(clusters)
         for i in range(10):
             mask = (clusters == i)
             labels[mask] = mode(digits.target[mask])[0]

         # Compute the accuracy
         accuracy_score(digits.target, labels)
Out[17]: 0.9415692821368948
```

That's a 94% classification accuracy *without using the labels*. This is the power of unsupervised learning when used carefully: it can extract information from the dataset that it might be difficult to extract by hand or by eye.

Example 2: k-Means for Color Compression

One interesting application of clustering is in color compression within images (this example is adapted from Scikit-Learn's "Color Quantization Using K-Means"). For example, imagine you have an image with millions of colors. In most images, a large number of the colors will be unused, and many of the pixels in the image will have similar or even identical colors.

For example, consider the image shown in Figure 47-11, which is from the Scikit-Learn `datasets` module (for this to work, you'll have to have the `PIL` Python package installed):[2]

2 For a color version of this and following images, see the online version of this book.

```
In [18]: # Note: this requires the PIL package to be installed
         from sklearn.datasets import load_sample_image
         china = load_sample_image("china.jpg")
         ax = plt.axes(xticks=[], yticks=[])
         ax.imshow(china);
```

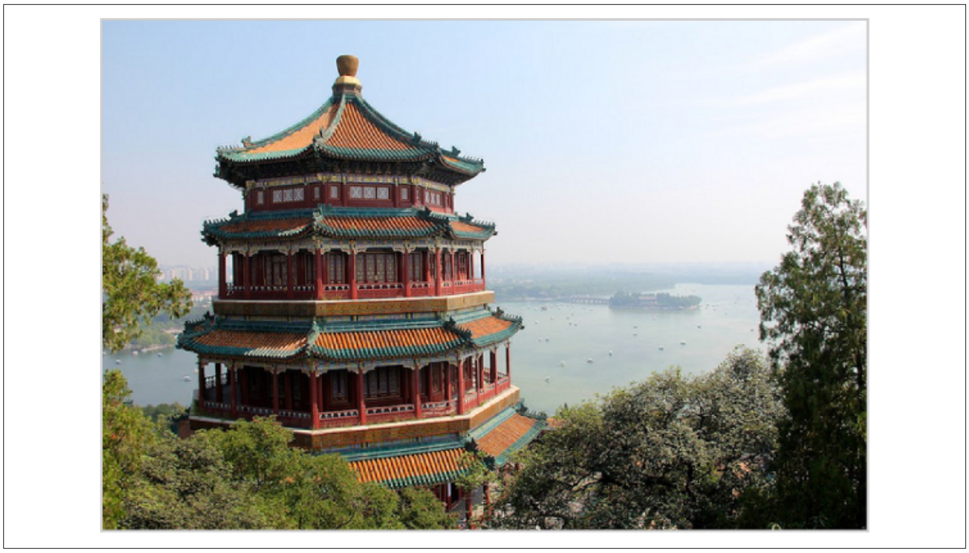


Figure 47-11. The input image

The image itself is stored in a three-dimensional array of size `(height, width, RGB)`, containing red/blue/green contributions as integers from 0 to 255:

```
In [19]: china.shape
Out[19]: (427, 640, 3)
```

One way we can view this set of pixels is as a cloud of points in a three-dimensional color space. We will reshape the data to `[n_samples, n_features]` and rescale the colors so that they lie between 0 and 1:

```
In [20]: data = china / 255.0  # use 0...1 scale
         data = data.reshape(-1, 3)
         data.shape
Out[20]: (273280, 3)
```

We can visualize these pixels in this color space, using a subset of 10,000 pixels for efficiency (see Figure 47-12).

```
In [21]: def plot_pixels(data, title, colors=None, N=10000):
             if colors is None:
                 colors = data

             # choose a random subset
             rng = np.random.default_rng(0)
```

```
        i = rng.permutation(data.shape[0])[:N]
        colors = colors[i]
        R, G, B = data[i].T

        fig, ax = plt.subplots(1, 2, figsize=(16, 6))
        ax[0].scatter(R, G, color=colors, marker='.')
        ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

        ax[1].scatter(R, B, color=colors, marker='.')
        ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

        fig.suptitle(title, size=20);
In [22]: plot_pixels(data, title='Input color space: 16 million possible colors')
```

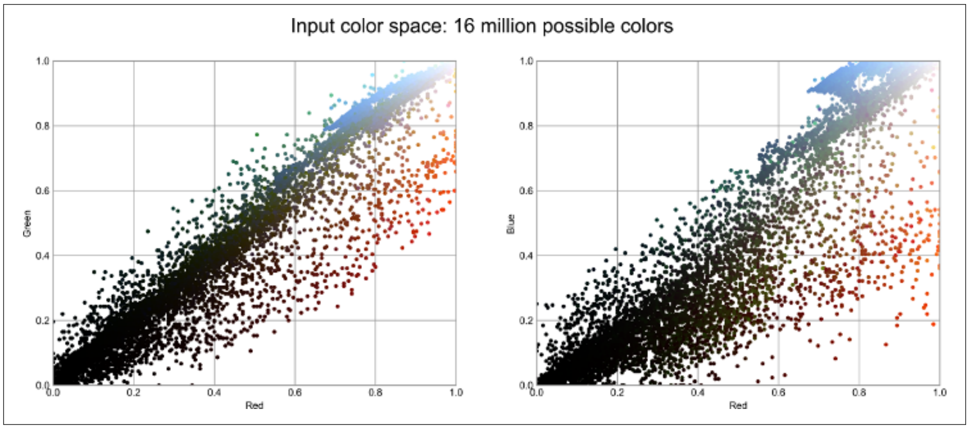


Figure 47-12. The distribution of the pixels in RGB color space[3]

Now let's reduce these 16 million colors to just 16 colors, using a *k*-means clustering across the pixel space. Because we are dealing with a very large dataset, we will use the mini-batch *k*-means, which operates on subsets of the data to compute the result (shown in Figure 47-13) much more quickly than the standard *k*-means algorithm:

```
In [23]: from sklearn.cluster import MiniBatchKMeans
         kmeans = MiniBatchKMeans(16)
         kmeans.fit(data)
         new_colors = kmeans.cluster_centers_[kmeans.predict(data)]

         plot_pixels(data, colors=new_colors,
                     title="Reduced color space: 16 colors")
```

3 A full-size version of this figure can be found on GitHub.

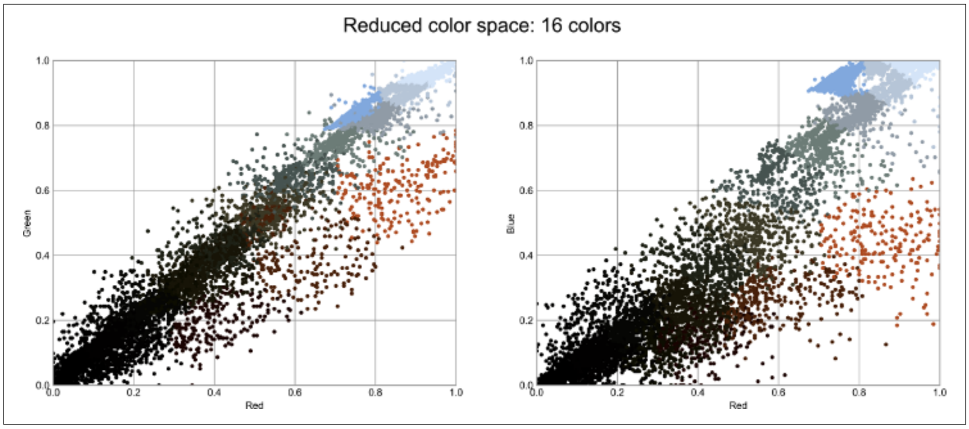


Figure 47-13. 16 clusters in RGB color space[4]

The result is a recoloring of the original pixels, where each pixel is assigned the color of its closest cluster center. Plotting these new colors in the image space rather than the pixel space shows us the effect of this (see Figure 47-14).

```
In [24]: china_recolored = new_colors.reshape(china.shape)

         fig, ax = plt.subplots(1, 2, figsize=(16, 6),
                                subplot_kw=dict(xticks=[], yticks=[]))
         fig.subplots_adjust(wspace=0.05)
         ax[0].imshow(china)
         ax[0].set_title('Original Image', size=16)
         ax[1].imshow(china_recolored)
         ax[1].set_title('16-color Image', size=16);
```

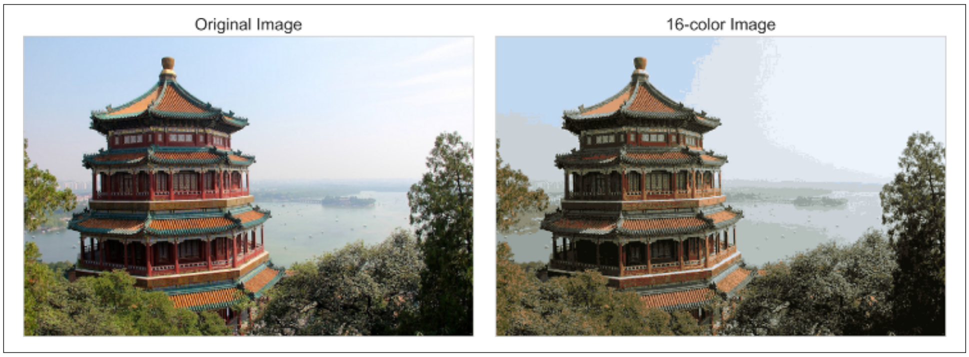


Figure 47-14. A comparison of the full-color image (left) and the 16-color image (right)

4 A full-size version of this figure can be found on GitHub.

Some detail is certainly lost in the rightmost panel, but the overall image is still easily recognizable. In terms of the bytes required to store the raw data, the image on the right achieves a compression factor of around 1 million! Now, this kind of approach is not going to match the fidelity of purpose-built image compression schemes like JPEG, but the example shows the power of thinking outside of the box with unsupervised methods like *k*-means.

CHAPTER 48

In Depth: Gaussian Mixture Models

The *k*-means clustering model explored in the previous chapter is simple and relatively easy to understand, but its simplicity leads to practical challenges in its application. In particular, the nonprobabilistic nature of *k*-means and its use of simple distance from cluster center to assign cluster membership leads to poor performance for many real-world situations. In this chapter we will take a look at Gaussian mixture models, which can be viewed as an extension of the ideas behind *k*-means, but can also be a powerful tool for estimation beyond simple clustering.

We begin with the standard imports:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Motivating Gaussian Mixtures: Weaknesses of k-Means

Let's take a look at some of the weaknesses of *k*-means and think about how we might improve the cluster model. As we saw in the previous chapter, given simple, well-separated data, *k*-means finds suitable clustering results.

For example, if we have simple blobs of data, the *k*-means algorithm can quickly label those clusters in a way that closely matches what we might do by eye (see Figure 48-1).

```
In [2]: # Generate some data
        from sklearn.datasets import make_blobs
        X, y_true = make_blobs(n_samples=400, centers=4,
                               cluster_std=0.60, random_state=0)
        X = X[:, ::-1] # flip axes for better plotting
```

```
In [3]: # Plot the data with k-means labels
        from sklearn.cluster import KMeans
        kmeans = KMeans(4, random_state=0)
        labels = kmeans.fit(X).predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

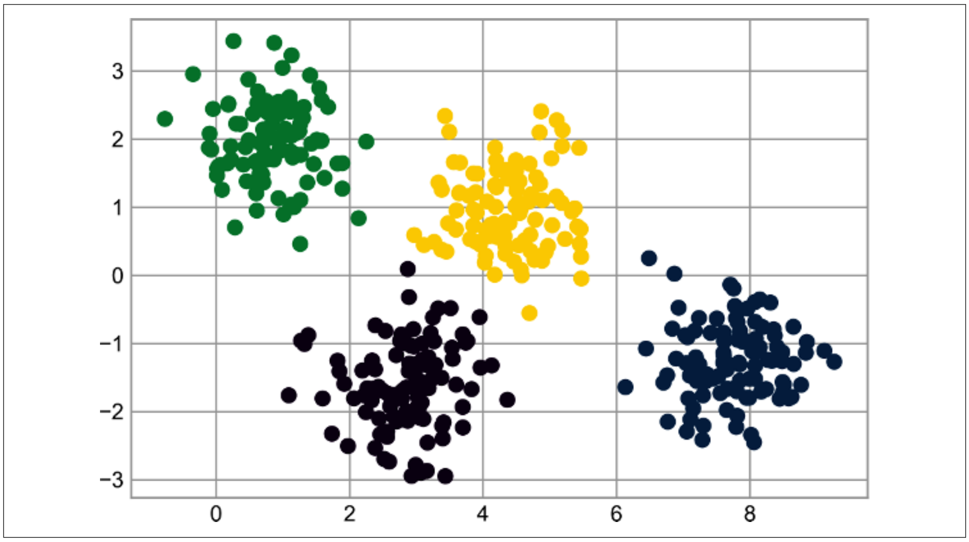


Figure 48-1. k-means labels for simple data

From an intuitive standpoint, we might expect that the clustering assignment for some points is more certain than others: for example, there appears to be a very slight overlap between the two middle clusters, such that we might not have complete confidence in the cluster assignment of points between them. Unfortunately, the *k*-means model has no intrinsic measure of probability or uncertainty of cluster assignments (although it may be possible to use a bootstrap approach to estimate this uncertainty). For this, we must think about generalizing the model.

One way to think about the *k*-means model is that it places a circle (or, in higher dimensions, a hypersphere) at the center of each cluster, with a radius defined by the most distant point in the cluster. This radius acts as a hard cutoff for cluster assignment within the training set: any point outside this circle is not considered a member of the cluster. We can visualize this cluster model with the following function (see Figure 48-2).

```
In [4]: from sklearn.cluster import KMeans
        from scipy.spatial.distance import cdist

        def plot_kmeans(kmeans, X, n_clusters=4, rseed=0, ax=None):
            labels = kmeans.fit_predict(X)

            # plot the input data
            ax = ax or plt.gca()
            ax.axis('equal')
            ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)

            # plot the representation of the KMeans model
            centers = kmeans.cluster_centers_
            radii = [cdist(X[labels == i], [center]).max()
                     for i, center in enumerate(centers)]
            for c, r in zip(centers, radii):
                ax.add_patch(plt.Circle(c, r, ec='black', fc='lightgray',
                                        lw=3, alpha=0.5, zorder=1))

In [5]: kmeans = KMeans(n_clusters=4, random_state=0)
        plot_kmeans(kmeans, X)
```

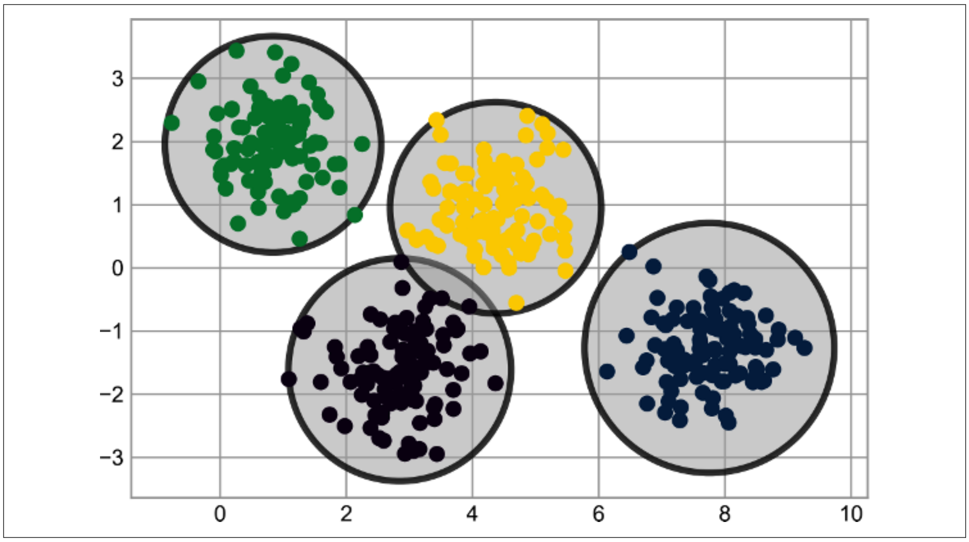


Figure 48-2. The circular clusters implied by the k*-means model*

An important observation for *k*-means is that these cluster models *must be circular*: *k*-means has no built-in way of accounting for oblong or elliptical clusters. So, for example, if we take the same data and transform it, the cluster assignments end up becoming muddled, as you can see in Figure 48-3.

```
In [6]: rng = np.random.RandomState(13)
        X_stretched = np.dot(X, rng.randn(2, 2))

        kmeans = KMeans(n_clusters=4, random_state=0)
        plot_kmeans(kmeans, X_stretched)
```

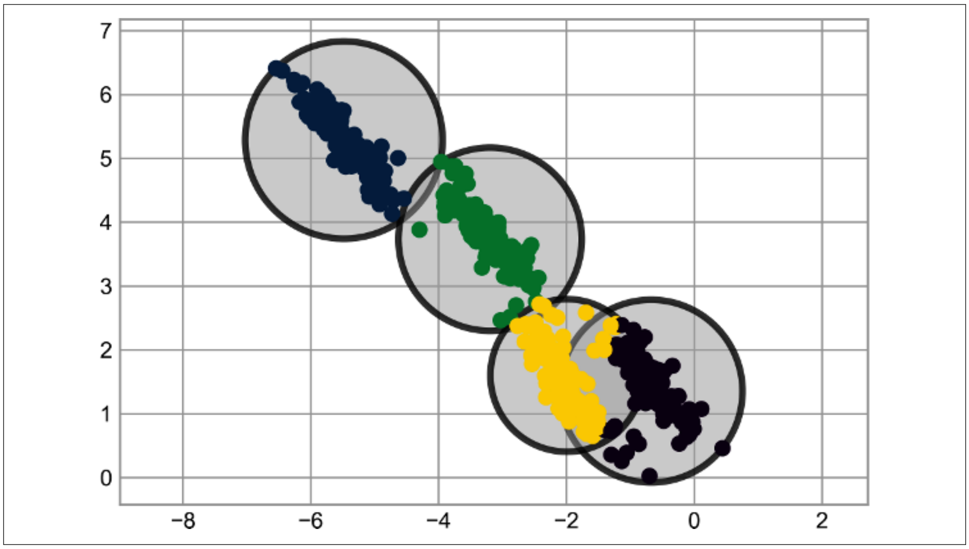


Figure 48-3. Poor performance of k-means *for noncircular clusters*

By eye, we recognize that these transformed clusters are noncircular, and thus circular clusters would be a poor fit. Nevertheless, *k*-means is not flexible enough to account for this, and tries to force-fit the data into four circular clusters. This results in a mixing of cluster assignments where the resulting circles overlap: see especially the bottom-right of this plot. One might imagine addressing this particular situation by preprocessing the data with PCA (see Chapter 45), but in practice there is no guarantee that such a global operation will circularize the individual groups.

These two disadvantages of *k*-means—its lack of flexibility in cluster shape and lack of probabilistic cluster assignment—mean that for many datasets (especially low-dimensional datasets) it may not perform as well as you might hope.

You might imagine addressing these weaknesses by generalizing the *k*-means model: for example, you could measure uncertainty in cluster assignment by comparing the distances of each point to *all* cluster centers, rather than focusing on just the closest. You might also imagine allowing the cluster boundaries to be ellipses rather than circles, so as to account for noncircular clusters. It turns out these are two essential components of a different type of clustering model, Gaussian mixture models.

Generalizing E–M: Gaussian Mixture Models

A Gaussian mixture model (GMM) attempts to find a mixture of multidimensional Gaussian probability distributions that best model any input dataset. In the simplest case, GMMs can be used for finding clusters in the same manner as *k*-means (see Figure 48-4).

```
In [7]: from sklearn.mixture import GaussianMixture
        gmm = GaussianMixture(n_components=4).fit(X)
        labels = gmm.predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

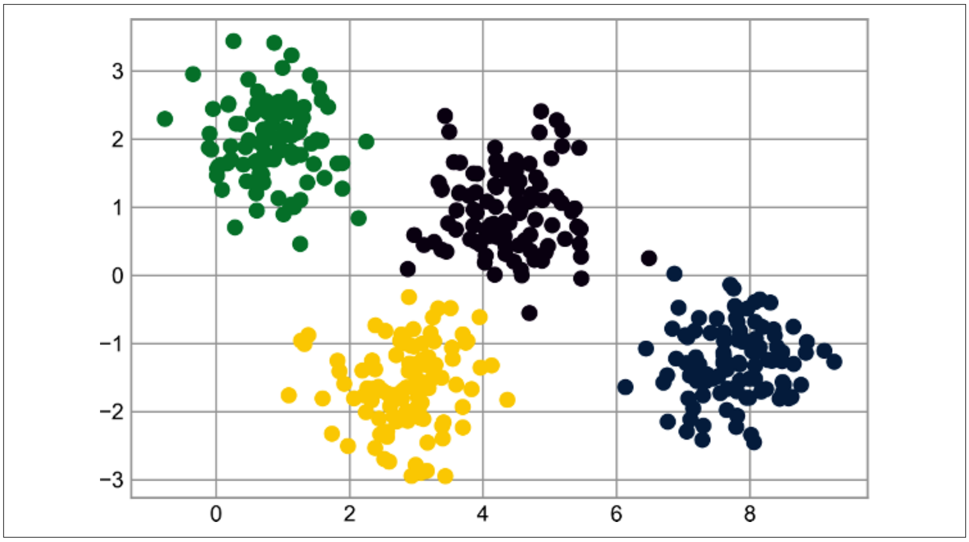


Figure 48-4. Gaussian mixture model labels for the data

But because a GMM contains a probabilistic model under the hood, it is also possible to find probabilistic cluster assignments—in Scikit-Learn this is done using the `predict_proba` method. This returns a matrix of size `[n_samples, n_clusters]` which measures the probability that any point belongs to the given cluster:

```
In [8]: probs = gmm.predict_proba(X)
        print(probs[:5].round(3))
Out[8]: [[0.    0.531 0.469 0.    ]
         [0.    0.    0.    1.    ]
         [0.    0.    0.    1.    ]
         [0.    1.    0.    0.    ]
         [0.    0.    0.    1.    ]]
```

We can visualize this uncertainty by, for example, making the size of each point proportional to the certainty of its prediction; looking at Figure 48-5, we can see that it is precisely the points at the boundaries between clusters that reflect this uncertainty of cluster assignment:

```
In [9]: size = 50 * probs.max(1) ** 2  # square emphasizes differences
        plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size);
```

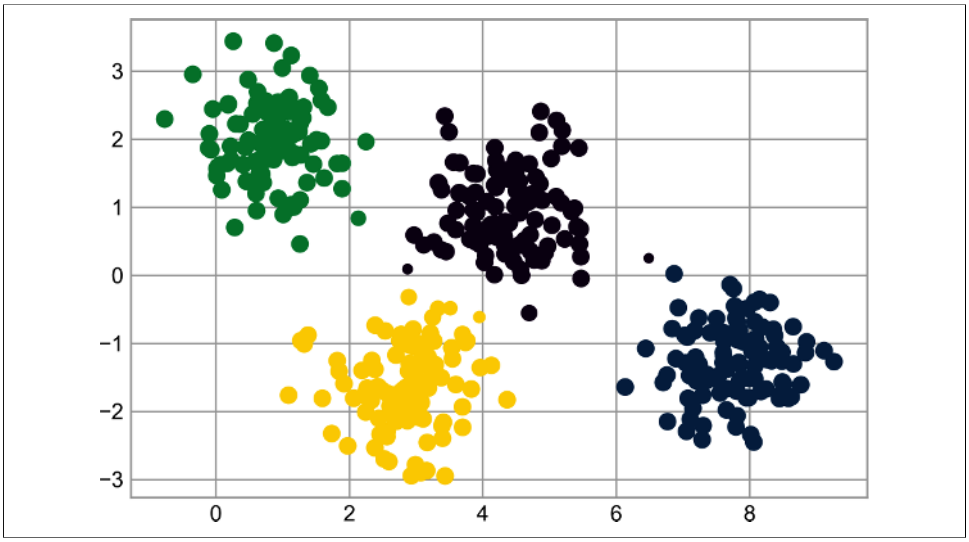


Figure 48-5. GMM probabilistic labels: probabilities are shown by the size of points

Under the hood, a Gaussian mixture model is very similar to *k*-means: it uses an expectation–maximization approach, which qualitatively does the following:

1. Choose starting guesses for the location and shape.
2. Repeat until converged:
   a. *E-step*: For each point, find weights encoding the probability of membership in each cluster.
   b. *M-step*: For each cluster, update its location, normalization, and shape based on *all* data points, making use of the weights.

The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the *k*-means expectation–maximization approach, this algorithm can sometimes miss the globally optimal solution, and thus in practice multiple random initializations are used.

Let's create a function that will help us visualize the locations and shapes of the GMM clusters by drawing ellipses based on the GMM output:

```
In [10]: from matplotlib.patches import Ellipse

         def draw_ellipse(position, covariance, ax=None, **kwargs):
             """Draw an ellipse with a given position and covariance"""
             ax = ax or plt.gca()

             # Convert covariance to principal axes
             if covariance.shape == (2, 2):
                 U, s, Vt = np.linalg.svd(covariance)
                 angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
                 width, height = 2 * np.sqrt(s)
             else:
                 angle = 0
                 width, height = 2 * np.sqrt(covariance)

             # Draw the ellipse
             for nsig in range(1, 4):
                 ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                                      angle, **kwargs))

         def plot_gmm(gmm, X, label=True, ax=None):
             ax = ax or plt.gca()
             labels = gmm.fit(X).predict(X)
             if label:
                 ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis',
                            zorder=2)
             else:
                 ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
             ax.axis('equal')

             w_factor = 0.2 / gmm.weights_.max()
             for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
                 draw_ellipse(pos, covar, alpha=w * w_factor)
```

With this in place, we can take a look at what the four-component GMM gives us for our initial data (see Figure 48-6).

```
In [11]: gmm = GaussianMixture(n_components=4, random_state=42)
         plot_gmm(gmm, X)
```

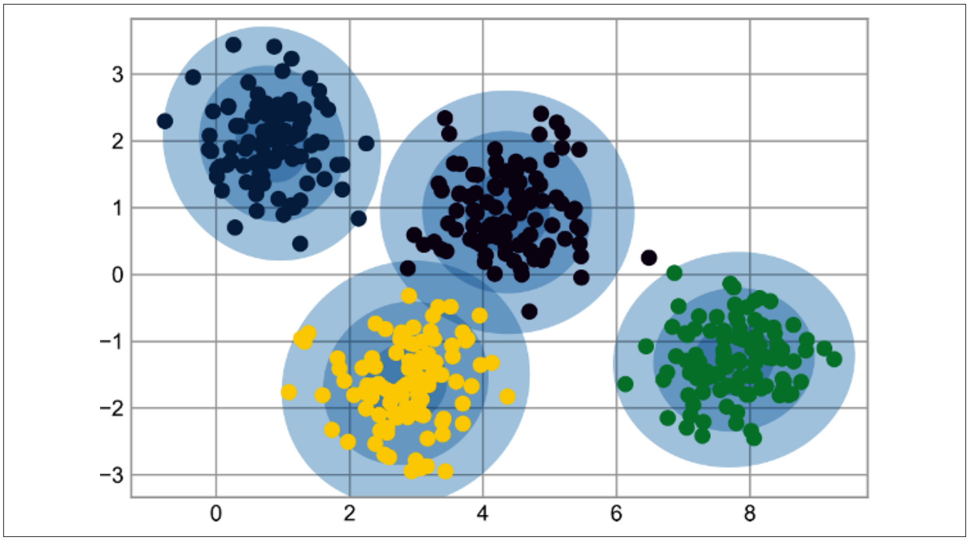


Figure 48-6. The four-component GMM in the presence of circular clusters

Similarly, we can use the GMM approach to fit our stretched dataset; allowing for a full covariance the model will fit even very oblong, stretched-out clusters, as we can see in Figure 48-7.

```
In [12]: gmm = GaussianMixture(n_components=4, covariance_type='full',
                               random_state=42)
         plot_gmm(gmm, X_stretched)
```

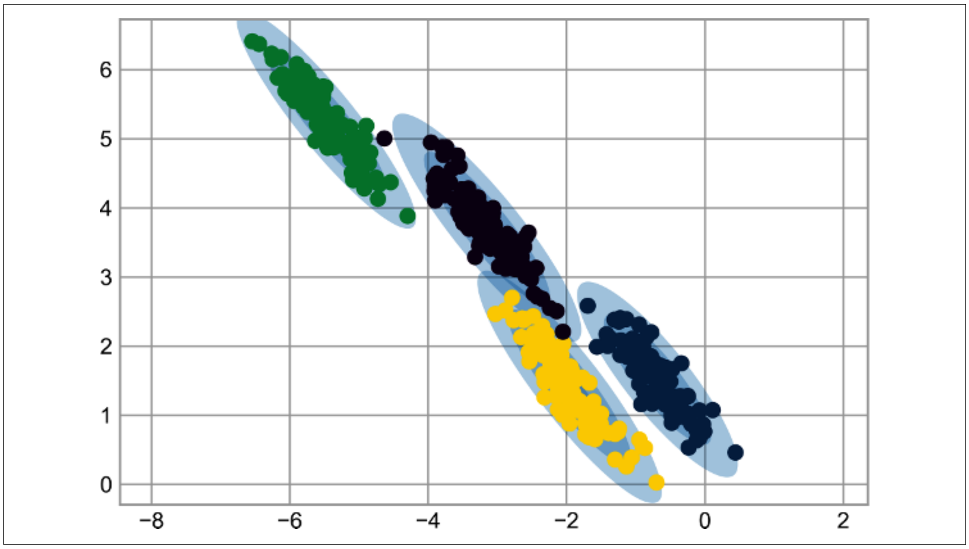


Figure 48-7. The four-component GMM in the presence of noncircular clusters

This makes clear that GMMs address the two main practical issues with *k*-means encountered before.

Choosing the Covariance Type

If you look at the details of the preceding fits, you will see that the `covariance_type` option was set differently within each. This hyperparameter controls the degrees of freedom in the shape of each cluster; it's essential to set this carefully for any given problem. The default is `covariance_type="diag"`, which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes. `covariance_type="spherical"` is a slightly simpler and faster model, which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of *k*-means, though it's not entirely equivalent. A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use `covariance_type="full"`, which allows each cluster to be modeled as an ellipse with arbitrary orientation. Figure 48-8 represents these three choices for a single cluster.

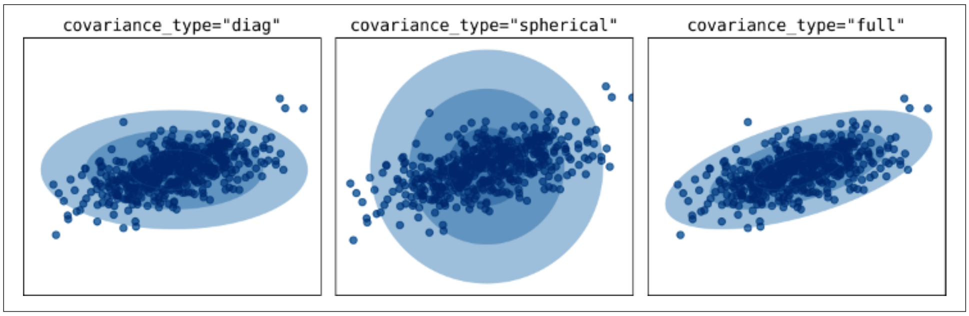


Figure 48-8. Visualization of GMM covariance types[1]

Gaussian Mixture Models as Density Estimation

Though the GMM is often categorized as a clustering algorithm, fundamentally it is an algorithm for *density estimation*. That is to say, the result of a GMM fit to some data is technically not a clustering model, but a generative probabilistic model describing the distribution of the data.

As an example, consider some data generated from Scikit-Learn's `make_moons` function, introduced in Chapter 47 (see Figure 48-9).

1 Code to produce this figure can be found in the online appendix.

```
In [13]: from sklearn.datasets import make_moons
         Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
         plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```

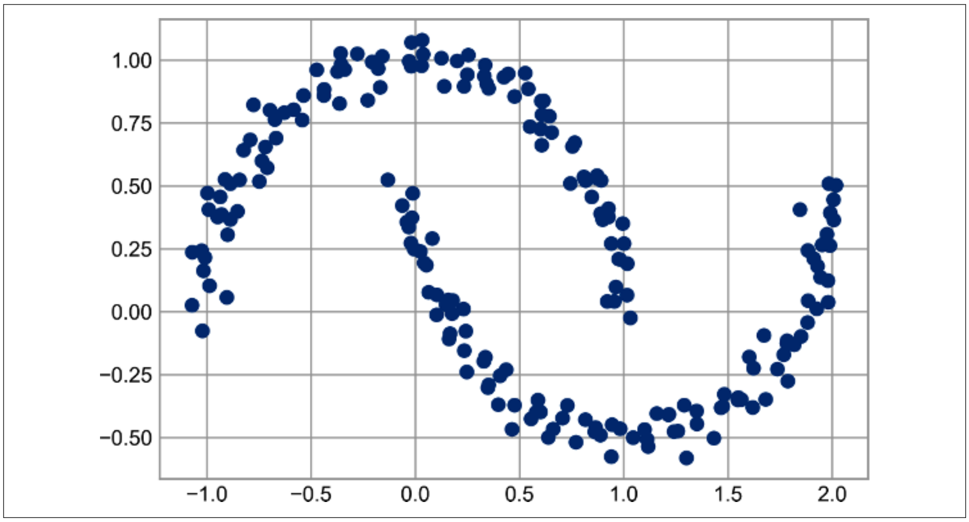


Figure 48-9. GMM applied to clusters with nonlinear boundaries

If we try to fit this with a two-component GMM viewed as a clustering model, the results are not particularly useful (see Figure 48-10).

```
In [14]: gmm2 = GaussianMixture(n_components=2, covariance_type='full',
                                random_state=0)
         plot_gmm(gmm2, Xmoon)
```

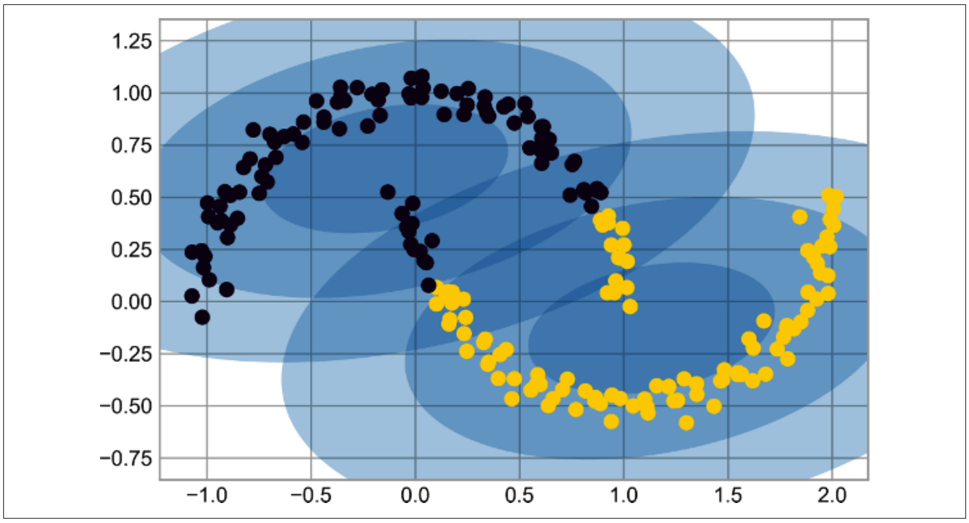


Figure 48-10. Two-component GMM fit to nonlinear clusters

But if we instead use many more components and ignore the cluster labels, we find a fit that is much closer to the input data (see Figure 48-11).

```
In [15]: gmm16 = GaussianMixture(n_components=16, covariance_type='full',
                                 random_state=0)
         plot_gmm(gmm16, Xmoon, label=False)
```

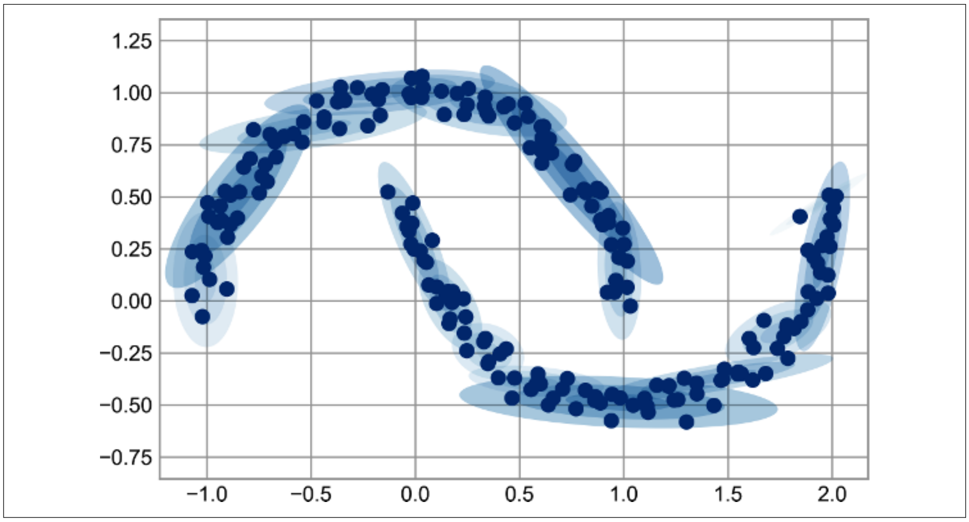


Figure 48-11. Using many GMM clusters to model the distribution of points

Here the mixture of 16 Gaussian components serves not to find separated clusters of data, but rather to model the overall *distribution* of the input data. This is a generative model of the distribution, meaning that the GMM gives us the recipe to generate new random data distributed similarly to our input. For example, here are 400 new points drawn from this 16-component GMM fit to our original data (see Figure 48-12).

```
In [16]: Xnew, ynew = gmm16.sample(400)
         plt.scatter(Xnew[:, 0], Xnew[:, 1]);
```

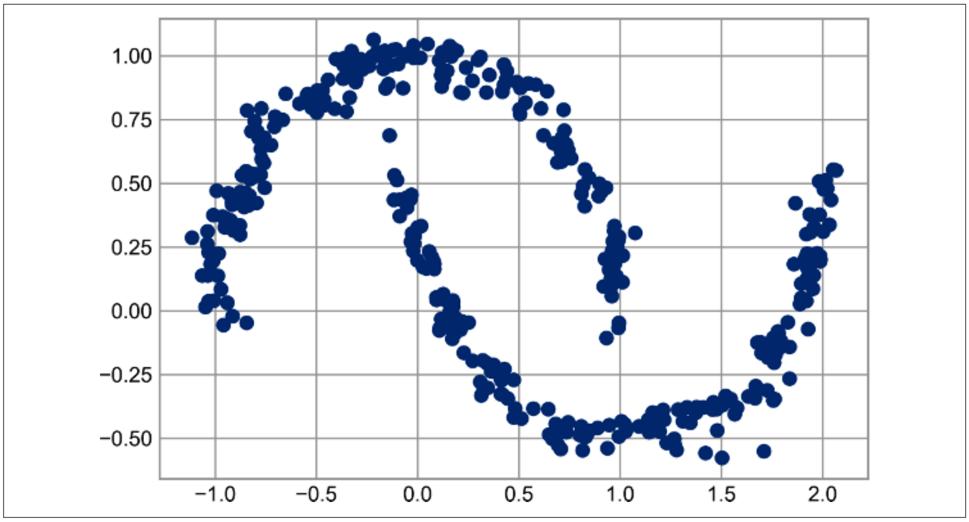


Figure 48-12. New data drawn from the 16-component GMM

A GMM is convenient as a flexible means of modeling an arbitrary multidimensional distribution of data.

The fact that a GMM is a generative model gives us a natural means of determining the optimal number of components for a given dataset. A generative model is inherently a probability distribution for the dataset, and so we can simply evaluate the *likelihood* of the data under the model, using cross-validation to avoid overfitting. Another means of correcting for overfitting is to adjust the model likelihoods using some analytic criterion such as the Akaike information criterion (AIC) or the Bayesian information criterion (BIC). Scikit-Learn's `GaussianMixture` estimator actually includes built-in methods that compute both of these, so it is very easy to operate using this approach.

Let's look at the AIC and BIC versus the number of GMM components for our moons dataset (see Figure 48-13).

```
In [17]: n_components = np.arange(1, 21)
         models = [GaussianMixture(n, covariance_type='full',
                                   random_state=0).fit(Xmoon)
                   for n in n_components]

         plt.plot(n_components, [m.bic(Xmoon) for m in models], label='BIC')
         plt.plot(n_components, [m.aic(Xmoon) for m in models], label='AIC')
         plt.legend(loc='best')
         plt.xlabel('n_components');
```

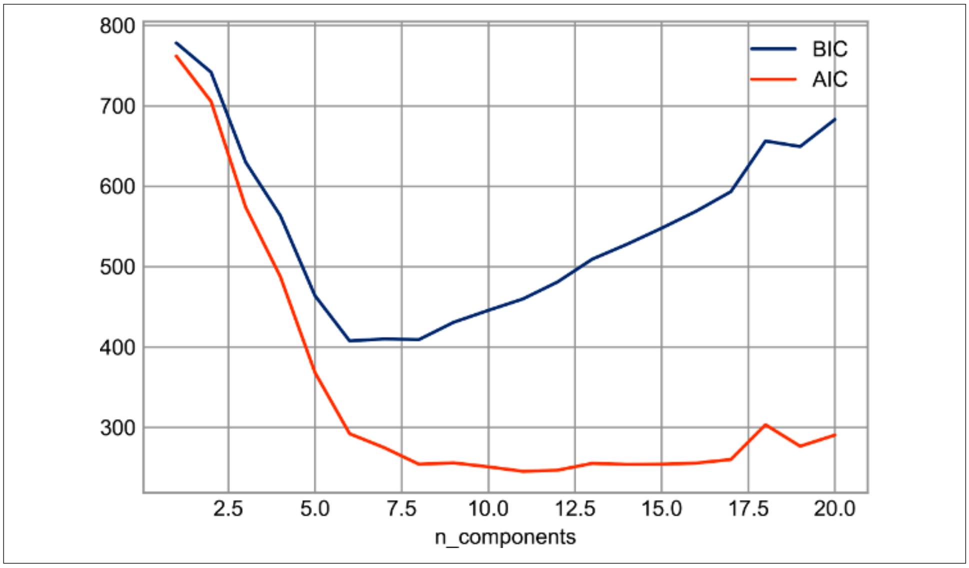


Figure 48-13. Visualization of AIC and BIC for choosing the number of GMM components

The optimal number of clusters is the value that minimizes the AIC or BIC, depending on which approximation we wish to use. The AIC tells us that our choice of 16 components earlier was probably too many: around 8–12 components would have been a better choice. As is typical with this sort of problem, the BIC recommends a simpler model.

Notice the important point: this choice of number of components measures how well a GMM works *as a density estimator*, not how well it works *as a clustering algorithm*. I'd encourage you to think of the GMM primarily as a density estimator, and use it for clustering only when warranted within simple datasets.

Example: GMMs for Generating New Data

We just saw a simple example of using a GMM as a generative model in order to create new samples from the distribution defined by the input data. Here we will run with this idea and generate *new handwritten digits* from the standard digits corpus that we have used before.

To start with, let's load the digits data using Scikit-Learn's data tools:

```
In [18]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.data.shape
Out[18]: (1797, 64)
```

Next, let's plot the first 50 of these to recall exactly what we're looking at (see Figure 48-14).

```
In [19]: def plot_digits(data):
             fig, ax = plt.subplots(5, 10, figsize=(8, 4),
                                    subplot_kw=dict(xticks=[], yticks=[]))
             fig.subplots_adjust(hspace=0.05, wspace=0.05)
             for i, axi in enumerate(ax.flat):
                 im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
                 im.set_clim(0, 16)
         plot_digits(digits.data)
```

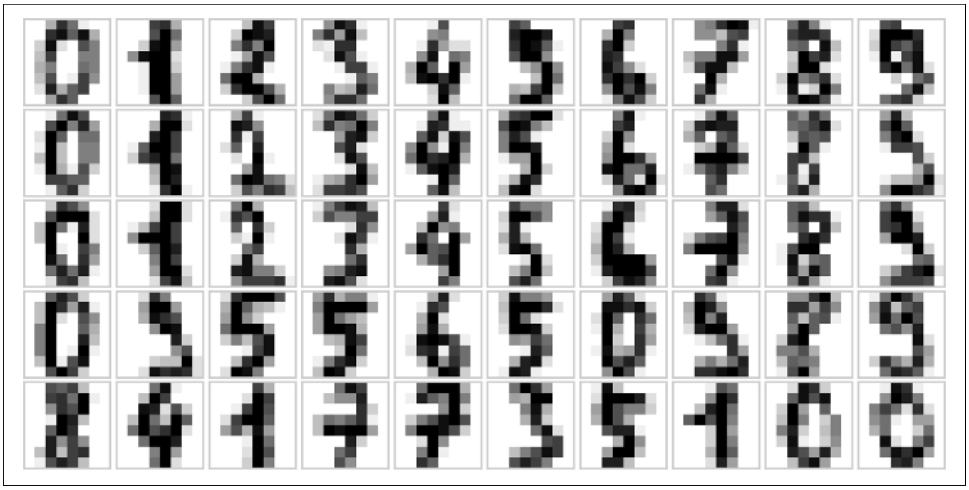


Figure 48-14. Handwritten digits input

We have nearly 1,800 digits in 64 dimensions, and we can build a GMM on top of these to generate more. GMMs can have difficulty converging in such a high-dimensional space, so we will start with an invertible dimensionality reduction algorithm on the data. Here we will use a straightforward PCA, asking it to preserve 99% of the variance in the projected data:

```
In [20]: from sklearn.decomposition import PCA
         pca = PCA(0.99, whiten=True)
         data = pca.fit_transform(digits.data)
         data.shape
Out[20]: (1797, 41)
```

The result is 41 dimensions, a reduction of nearly 1/3 with almost no information loss. Given this projected data, let's use the AIC to get a gauge for the number of GMM components we should use (see Figure 48-15).

```
In [21]: n_components = np.arange(50, 210, 10)
         models = [GaussianMixture(n, covariance_type='full', random_state=0)
                   for n in n_components]
         aics = [model.fit(data).aic(data) for model in models]
         plt.plot(n_components, aics);
```

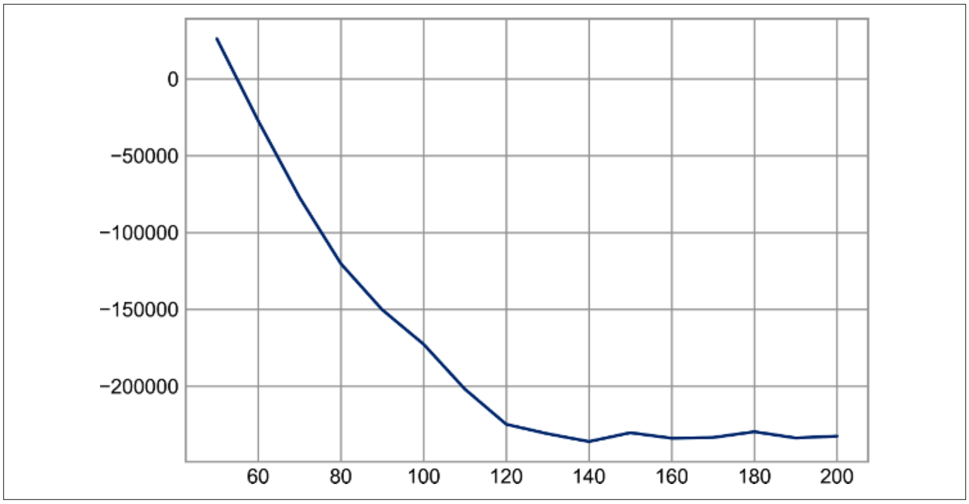


Figure 48-15. AIC curve for choosing the appropriate number of GMM components

It appears that around 140 components minimizes the AIC; we will use this model. Let's quickly fit this to the data and confirm that it has converged:

```
In [22]: gmm = GaussianMixture(140, covariance_type='full', random_state=0)
         gmm.fit(data)
         print(gmm.converged_)
Out[22]: True
```

Now we can draw samples of 100 new points within this 41-dimensional projected space, using the GMM as a generative model:

```
In [23]: data_new, label_new = gmm.sample(100)
         data_new.shape
Out[23]: (100, 41)
```

Finally, we can use the inverse transform of the PCA object to construct the new digits (see Figure 48-16).

```
In [24]: digits_new = pca.inverse_transform(data_new)
         plot_digits(digits_new)
```

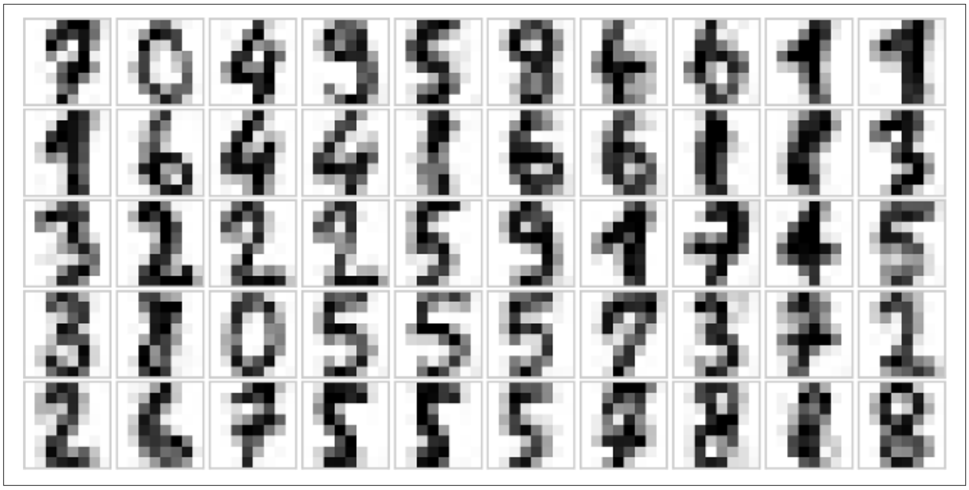


Figure 48-16. "New" digits randomly drawn from the underlying model of the GMM estimator

The results for the most part look like plausible digits from the dataset!

Consider what we've done here: given a sampling of handwritten digits, we have modeled the distribution of that data in such a way that we can generate brand new samples of digits from the data: these are "handwritten digits," which do not individually appear in the original dataset, but rather capture the general features of the input data as modeled by the mixture model. Such a generative model of digits can prove very useful as a component of a Bayesian generative classifier, as we shall see in the next chapter.

	Jerry Maguire	Oceans	Road to Perdition	A Fortunate Man	Catch Me If You Can	Driving Miss Daisy	The Two Popes	The Laundromat	Code 8	The Social Network	...
Customer 1	●	●	●	●	4	●	●	●	●	●	...
Customer 2	●	●	3	●	●	●	3	●	●	3	...
Customer 3	●	2	●	4	●	●	●	●	2	●	...
Customer 4	3	●	●	●	●	●	●	●	●	●	...
Customer 5	5	1	●	●	4	●	●	●	●	●	...
Customer 6	●	●	●	●	●	2	4	●	●	●	...
Customer 7	●	●	5	●	●	●	●	3	●	●	...
Customer 8	●	●	●	●	●	●	●	●	●	●	...
Customer 9	3	●	●	●	5	●	●	1	●	●	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

TABLE 12.2. *Excerpt of the Netflix movie rating data. The movies are rated from 1 (worst) to 5 (best). The symbol ● represents a missing value: a movie that was not rated by the corresponding customer.*

typically massive, algorithms have been developed that can exploit the high level of missingness in order to perform efficient computations.

12.4 Clustering Methods

Clustering refers to a very broad set of techniques for finding *subgroups*, or *clusters*, in a data set. When we cluster the observations of a data set, we seek to partition them into distinct groups so that the observations within each group are quite similar to each other, while observations in different groups are quite different from each other. Of course, to make this concrete, we must define what it means for two or more observations to be *similar* or *different*. Indeed, this is often a domain-specific consideration that must be made based on knowledge of the data being studied.

clustering

For instance, suppose that we have a set of n observations, each with p features. The n observations could correspond to tissue samples for patients with breast cancer, and the p features could correspond to measurements collected for each tissue sample; these could be clinical measurements, such as tumor stage or grade, or they could be gene expression measurements. We may have a reason to believe that there is some heterogeneity among the n tissue samples; for instance, perhaps there are a few different *unknown* subtypes of breast cancer. Clustering could be used to find these subgroups. This is an unsupervised problem because we are trying to discover structure—in this case, distinct clusters—on the basis of a data set. The goal in supervised problems, on the other hand, is to try to predict some outcome vector such as survival time or response to drug treatment.

Both clustering and PCA seek to simplify the data via a small number of summaries, but their mechanisms are different:

- PCA looks to find a low-dimensional representation of the observations that explain a good fraction of the variance;
- Clustering looks to find homogeneous subgroups among the observations.

Another application of clustering arises in marketing. We may have access to a large number of measurements (e.g. median household income, occupation, distance from nearest urban area, and so forth) for a large number of people. Our goal is to perform *market segmentation* by identifying subgroups of people who might be more receptive to a particular form of advertising, or more likely to purchase a particular product. The task of performing market segmentation amounts to clustering the people in the data set.

Since clustering is popular in many fields, there exist a great number of clustering methods. In this section we focus on perhaps the two best-known clustering approaches: *K-means clustering* and *hierarchical clustering*. In K-means clustering, we seek to partition the observations into a pre-specified number of clusters. On the other hand, in hierarchical clustering, we do not know in advance how many clusters we want; in fact, we end up with a tree-like visual representation of the observations, called a *dendrogram*, that allows us to view at once the clusterings obtained for each possible number of clusters, from 1 to n. There are advantages and disadvantages to each of these clustering approaches, which we highlight in this chapter.

K-means clustering

hierarchical clustering

dendrogram

In general, we can cluster observations on the basis of the features in order to identify subgroups among the observations, or we can cluster features on the basis of the observations in order to discover subgroups among the features. In what follows, for simplicity we will discuss clustering observations on the basis of the features, though the converse can be performed by simply transposing the data matrix.

12.4.1 K-Means Clustering

K-means clustering is a simple and elegant approach for partitioning a data set into K distinct, non-overlapping clusters. To perform K-means clustering, we must first specify the desired number of clusters K; then the K-means algorithm will assign each observation to exactly one of the K clusters. Figure 12.7 shows the results obtained from performing K-means clustering on a simulated example consisting of 150 observations in two dimensions, using three different values of K.

The K-means clustering procedure results from a simple and intuitive mathematical problem. We begin by defining some notation. Let $C_1, \ldots, C_K$ denote sets containing the indices of the observations in each cluster. These sets satisfy two properties:

1. $C_1 \cup C_2 \cup \cdots \cup C_K = \{1, \ldots, n\}$. In other words, each observation belongs to at least one of the K clusters.

2. $C_k \cap C_k = \emptyset$ for all $k = k$. In other words, the clusters are non-overlapping: no observation belongs to more than one cluster.

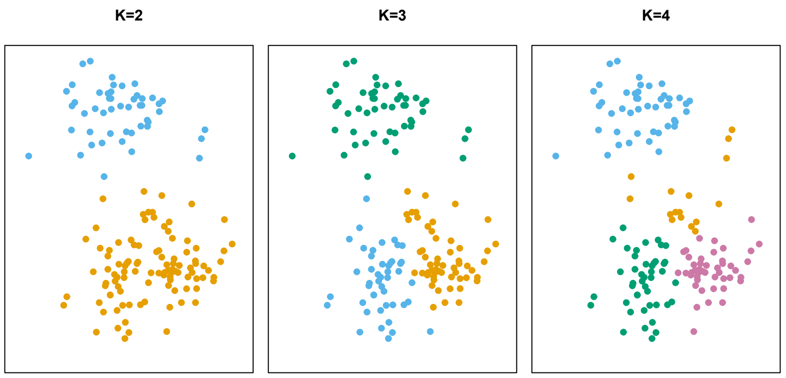


FIGURE 12.7. *A simulated data set with 150 observations in two-dimensional space. Panels show the results of applying K-means clustering with different values of K, the number of clusters. The color of each observation indicates the cluster to which it was assigned using the K-means clustering algorithm. Note that there is no ordering of the clusters, so the cluster coloring is arbitrary. These cluster labels were not used in clustering; instead, they are the outputs of the clustering procedure.*

For instance, if the ith observation is in the kth cluster, then $i \in C_k$. The idea behind K-means clustering is that a *good* clustering is one for which the *within-cluster variation* is as small as possible. The within-cluster variation for cluster C_k is a measure $W(C_k)$ of the amount by which the observations within a cluster differ from each other. Hence we want to solve the problem

$$\underset{C_1,\ldots,C_K}{\text{minimize}} \quad \mathop{}\limits_{k=1}^{K} W(C_k) \quad . \tag{12.15}$$

In words, this formula says that we want to partition the observations into K clusters such that the total within-cluster variation, summed over all K clusters, is as small as possible.

Solving (12.15) seems like a reasonable idea, but in order to make it actionable we need to define the within-cluster variation. There are many possible ways to define this concept, but by far the most common choice involves *squared Euclidean distance.* That is, we define

$$W(C_k) = \frac{1}{|C_k|} \mathop{}\limits_{i,i \in C_k} \mathop{}\limits_{j=1}^{p} (x_{ij} - x_{i\ j})^2, \tag{12.16}$$

where $|C_k|$ denotes the number of observations in the kth cluster. In other words, the within-cluster variation for the kth cluster is the sum of all of the pairwise squared Euclidean distances between the observations in the kth cluster, divided by the total number of observations in the kth cluster. Combining (12.15) and (12.16) gives the optimization problem that defines

K-means clustering,

$$\underset{C_1,\ldots,C_K}{\text{minimize}} \left\{ \overset{K}{\underset{k=1}{}} \frac{1}{|C_k|} \underset{i,i\ \in C_k}{} \overset{p}{\underset{j=1}{}} (x_{ij} - x_{i\ j})^2 \right\}. \tag{12.17}$$

Now, we would like to find an algorithm to solve (12.17)—that is, a method to partition the observations into K clusters such that the objective of (12.17) is minimized. This is in fact a very difficult problem to solve precisely, since there are almost K^n ways to partition n observations into K clusters. This is a huge number unless K and n are tiny! Fortunately, a very simple algorithm can be shown to provide a local optimum—a *pretty good solution*—to the K-means optimization problem (12.17). This approach is laid out in Algorithm 12.2.

Algorithm 12.2 *K-Means Clustering*

1. Randomly assign a number, from 1 to K, to each of the observations. These serve as initial cluster assignments for the observations.
2. Iterate until the cluster assignments stop changing:
   (a) For each of the K clusters, compute the cluster *centroid*. The kth cluster centroid is the vector of the p feature means for the observations in the kth cluster.
   (b) Assign each observation to the cluster whose centroid is closest (where *closest* is defined using Euclidean distance).

Algorithm 12.2 is guaranteed to decrease the value of the objective (12.17) at each step. To understand why, the following identity is illuminating:

$$\frac{1}{|C_k|} \underset{i,i\ \in C_k}{} \overset{p}{\underset{j=1}{}} (x_{ij} - x_{i\ j})^2 = 2 \underset{i\in C_k}{} \overset{p}{\underset{j=1}{}} (x_{ij} - \bar{x}_{kj})^2, \tag{12.18}$$

where $\bar{x}_{kj} = \frac{1}{|C_k|} \underset{i\in C_k}{} x_{ij}$ is the mean for feature j in cluster C_k. In Step 2(a) the cluster means for each feature are the constants that minimize the sum-of-squared deviations, and in Step 2(b), reallocating the observations can only improve (12.18). This means that as the algorithm is run, the clustering obtained will continually improve until the result no longer changes; the objective of (12.17) will never increase. When the result no longer changes, a *local optimum* has been reached. Figure 12.8 shows the progression of the algorithm on the toy example from Figure 12.7. K-means clustering derives its name from the fact that in Step 2(a), the cluster centroids are computed as the mean of the observations assigned to each cluster.

Because the K-means algorithm finds a local rather than a global optimum, the results obtained will depend on the initial (random) cluster assignment of each observation in Step 1 of Algorithm 12.2. For this reason, it is important to run the algorithm multiple times from different random

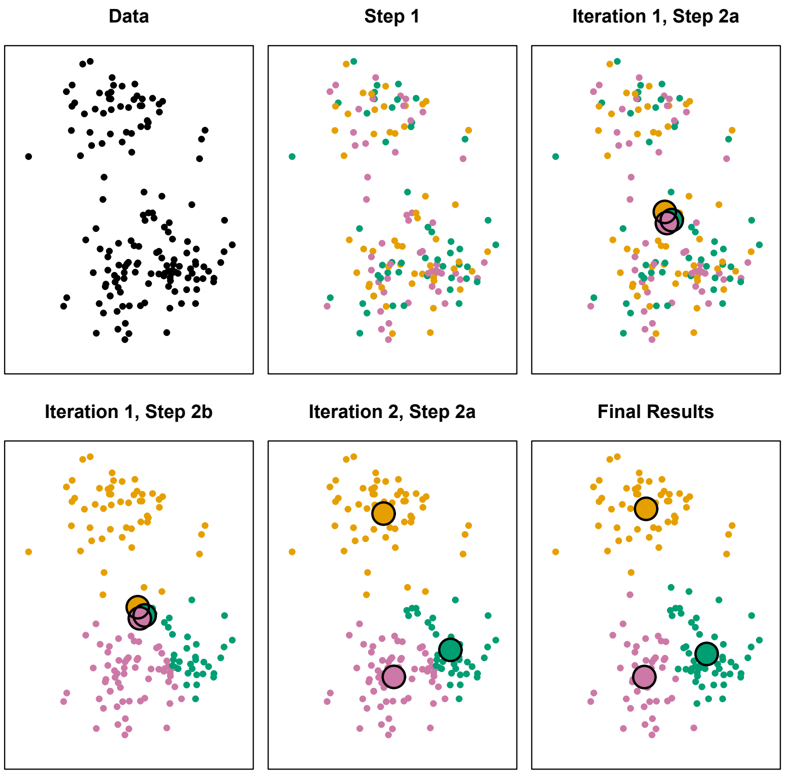


FIGURE 12.8. *The progress of the K-means algorithm on the example of Figure 12.7 with K=3.* Top left: *the observations are shown.* Top center: *in Step 1 of the algorithm, each observation is randomly assigned to a cluster.* Top right: *in Step 2(a), the cluster centroids are computed. These are shown as large colored disks. Initially the centroids are almost completely overlapping because the initial cluster assignments were chosen at random.* Bottom left: *in Step 2(b), each observation is assigned to the nearest centroid.* Bottom center: *Step 2(a) is once again performed, leading to new cluster centroids.* Bottom right: *the results obtained after ten iterations.*

initial configurations. Then one selects the *best* solution, i.e. that for which the objective (12.17) is smallest. Figure 12.9 shows the local optima obtained by running K-means clustering six times using six different initial cluster assignments, using the toy data from Figure 12.7. In this case, the best clustering is the one with an objective value of 235.8.

As we have seen, to perform K-means clustering, we must decide how many clusters we expect in the data. The problem of selecting K is far from simple. This issue, along with other practical considerations that arise in performing K-means clustering, is addressed in Section 12.4.3.

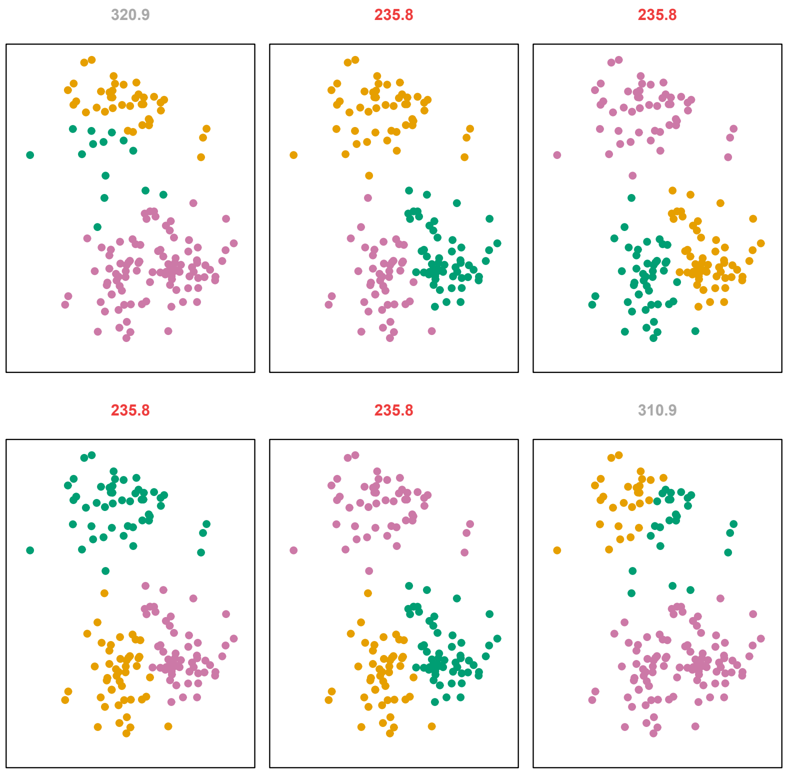


FIGURE 12.9. *K-means clustering performed six times on the data from Figure 12.7 with $K = 3$, each time with a different random assignment of the observations in Step 1 of the K-means algorithm. Above each plot is the value of the objective (12.17). Three different local optima were obtained, one of which resulted in a smaller value of the objective and provides better separation between the clusters. Those labeled in red all achieved the same best solution, with an objective value of 235.8.*

12.4.2 Hierarchical Clustering

One potential disadvantage of K-means clustering is that it requires us to pre-specify the number of clusters K. *Hierarchical clustering* is an alternative approach which does not require that we commit to a particular choice of K. Hierarchical clustering has an added advantage over K-means clustering in that it results in an attractive tree-based representation of the observations, called a *dendrogram*.

In this section, we describe *bottom-up* or *agglomerative* clustering. This is the most common type of hierarchical clustering, and refers to the fact that a dendrogram (generally depicted as an upside-down tree; see Figure 12.11) is built starting from the leaves and combining clusters up to the trunk. We will begin with a discussion of how to interpret a dendrogram

bottom-up
agglomerative

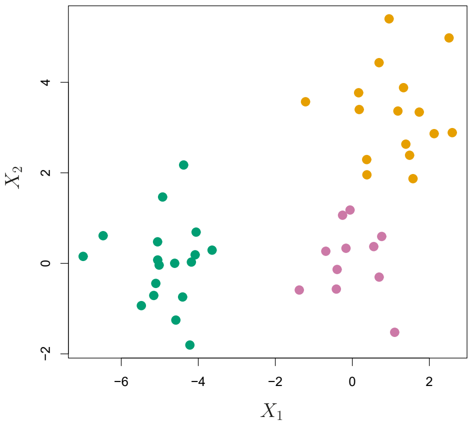


FIGURE 12.10. *Forty-five observations generated in two-dimensional space. In reality there are three distinct classes, shown in separate colors. However, we will treat these class labels as unknown and will seek to cluster the observations in order to discover the classes from the data.*

and then discuss how hierarchical clustering is actually performed—that is, how the dendrogram is built.

Interpreting a Dendrogram

We begin with the simulated data set shown in Figure 12.10, consisting of 45 observations in two-dimensional space. The data were generated from a three-class model; the true class labels for each observation are shown in distinct colors. However, suppose that the data were observed without the class labels, and that we wanted to perform hierarchical clustering of the data. Hierarchical clustering (with complete linkage, to be discussed later) yields the result shown in the left-hand panel of Figure 12.11. How can we interpret this dendrogram?

In the left-hand panel of Figure 12.11, each *leaf* of the dendrogram represents one of the 45 observations in Figure 12.10. However, as we move up the tree, some leaves begin to *fuse* into branches. These correspond to observations that are similar to each other. As we move higher up the tree, branches themselves fuse, either with leaves or other branches. The earlier (lower in the tree) fusions occur, the more similar the groups of observations are to each other. On the other hand, observations that fuse later (near the top of the tree) can be quite different. In fact, this statement can be made precise: for any two observations, we can look for the point in the tree where branches containing those two observations are first fused. The height of this fusion, as measured on the vertical axis, indicates how different the two observations are. Thus, observations that fuse at the very bottom of the tree are quite similar to each other, whereas observations that fuse close to the top of the tree will tend to be quite different.

This highlights a very important point in interpreting dendrograms that is often misunderstood. Consider the left-hand panel of Figure 12.12, which shows a simple dendrogram obtained from hierarchically clustering nine

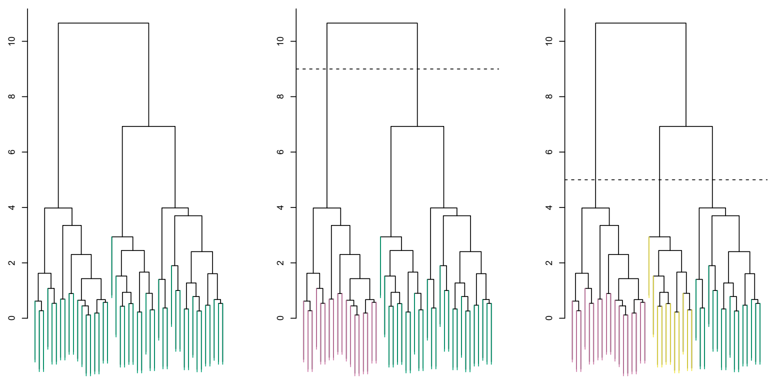


FIGURE 12.11. Left: *dendrogram obtained from hierarchically clustering the data from Figure 12.10 with complete linkage and Euclidean distance.* Center: *the dendrogram from the left-hand panel, cut at a height of nine (indicated by the dashed line). This cut results in two distinct clusters, shown in different colors.* Right: *the dendrogram from the left-hand panel, now cut at a height of five. This cut results in three distinct clusters, shown in different colors. Note that the colors were not used in clustering, but are simply used for display purposes in this figure.*

observations. One can see that observations 5 and 7 are quite similar to each other, since they fuse at the lowest point on the dendrogram. Observations 1 and 6 are also quite similar to each other. However, it is tempting but incorrect to conclude from the figure that observations 9 and 2 are quite similar to each other on the basis that they are located near each other on the dendrogram. In fact, based on the information contained in the dendrogram, observation 9 is no more similar to observation 2 than it is to observations $8, 5$, and 7. (This can be seen from the right-hand panel of Figure 12.12, in which the raw data are displayed.) To put it mathematically, there are 2^{n-1} possible reorderings of the dendrogram, where n is the number of leaves. This is because at each of the $n-1$ points where fusions occur, the positions of the two fused branches could be swapped without affecting the meaning of the dendrogram. Therefore, we cannot draw conclusions about the similarity of two observations based on their proximity along the *horizontal axis*. Rather, we draw conclusions about the similarity of two observations based on the location on the *vertical axis* where branches containing those two observations first are fused.

Now that we understand how to interpret the left-hand panel of Figure 12.11, we can move on to the issue of identifying clusters on the basis of a dendrogram. In order to do this, we make a horizontal cut across the dendrogram, as shown in the center and right-hand panels of Figure 12.11. The distinct sets of observations beneath the cut can be interpreted as clusters. In the center panel of Figure 12.11, cutting the dendrogram at a height of nine results in two clusters, shown in distinct colors. In the right-hand panel, cutting the dendrogram at a height of five results in three clusters. Further cuts can be made as one descends the dendrogram in order to obtain any number of clusters, between 1 (corresponding to no cut) and n

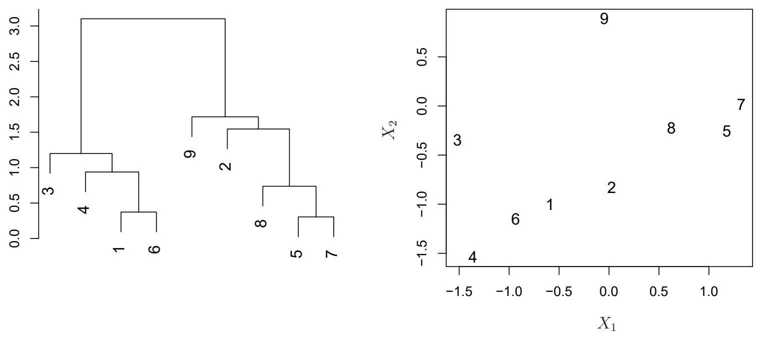


FIGURE 12.12. *An illustration of how to properly interpret a dendrogram with nine observations in two-dimensional space.* Left: *a dendrogram generated using Euclidean distance and complete linkage. Observations* 5 *and* 7 *are quite similar to each other, as are observations* 1 *and* 6*. However, observation* 9 *is* no more similar to *observation* 2 *than it is to observations* $8, 5,$ *and* 7*, even though observations* 9 *and* 2 *are close together in terms of horizontal distance. This is because observations* $2, 8, 5,$ *and* 7 *all fuse with observation* 9 *at the same height, approximately* 1.8*.* Right: *the raw data used to generate the dendrogram can be used to confirm that indeed, observation* 9 *is no more similar to observation* 2 *than it is to observations* $8, 5,$ *and* 7*.*

(corresponding to a cut at height 0, so that each observation is in its own cluster). In other words, the height of the cut to the dendrogram serves the same role as the K in K-means clustering: it controls the number of clusters obtained.

Figure 12.11 therefore highlights a very attractive aspect of hierarchical clustering: one single dendrogram can be used to obtain any number of clusters. In practice, people often look at the dendrogram and select by eye a sensible number of clusters, based on the heights of the fusion and the number of clusters desired. In the case of Figure 12.11, one might choose to select either two or three clusters. However, often the choice of where to cut the dendrogram is not so clear.

The term *hierarchical* refers to the fact that clusters obtained by cutting the dendrogram at a given height are necessarily nested within the clusters obtained by cutting the dendrogram at any greater height. However, on an arbitrary data set, this assumption of hierarchical structure might be unrealistic. For instance, suppose that our observations correspond to a group of men and women, evenly split among Americans, Japanese, and French. We can imagine a scenario in which the best division into two groups might split these people by gender, and the best division into three groups might split them by nationality. In this case, the true clusters are not nested, in the sense that the best division into three groups does not result from taking the best division into two groups and splitting up one of those groups. Consequently, this situation could not be well-represented by hierarchical clustering. Due to situations such as this one, hierarchical clustering can sometimes yield *worse* (i.e. less accurate) results than K-means clustering for a given number of clusters.

Algorithm 12.3 *Hierarchical Clustering*

```
1. Begin with $n$ observations and a measure (such as Euclidean distance) of all the $\binom{n}{2} = n(n-1)/2$ pairwise dissimilarities. Treat each observation as its own cluster.
2. For $i = n, n-1, \ldots, 2$:
    (a) Examine all pairwise inter-cluster dissimilarities among the $i$ clusters and identify the pair of clusters that are least dissimilar (that is, most similar). Fuse these two clusters. The dissimilarity between these two clusters indicates the height in the dendrogram at which the fusion should be placed.
    (b) Compute the new pairwise inter-cluster dissimilarities among the $i-1$ remaining clusters.
```

The Hierarchical Clustering Algorithm

The hierarchical clustering dendrogram is obtained via an extremely simple algorithm. We begin by defining some sort of *dissimilarity* measure between each pair of observations. Most often, Euclidean distance is used; we will discuss the choice of dissimilarity measure later in this chapter. The algorithm proceeds iteratively. Starting out at the bottom of the dendrogram, each of the n observations is treated as its own cluster. The two clusters that are most similar to each other are then *fused* so that there now are $n-1$ clusters. Next the two clusters that are most similar to each other are fused again, so that there now are $n-2$ clusters. The algorithm proceeds in this fashion until all of the observations belong to one single cluster, and the dendrogram is complete. Figure 12.13 depicts the first few steps of the algorithm, for the data from Figure 12.12. To summarize, the hierarchical clustering algorithm is given in Algorithm 12.3.

This algorithm seems simple enough, but one issue has not been addressed. Consider the bottom right panel in Figure 12.13. How did we determine that the cluster $\{5, 7\}$ should be fused with the cluster $\{8\}$? We have a concept of the dissimilarity between pairs of observations, but how do we define the dissimilarity between two clusters if one or both of the clusters contains multiple observations? The concept of dissimilarity between a pair of observations needs to be extended to a pair of *groups of observations*. This extension is achieved by developing the notion of *linkage*, which defines the dissimilarity between two groups of observations. The four most common types of linkage—*complete*, *average*, *single*, and *centroid*—are briefly described in Table 12.3. Average, complete, and single linkage are most popular among statisticians. Average and complete linkage are generally preferred over single linkage, as they tend to yield more balanced dendrograms. Centroid linkage is often used in genomics, but suffers from a major drawback in that an *inversion* can occur, whereby two clusters are fused at a height *below* either of the individual clusters in the dendrogram. This can lead to difficulties in visualization as well as in interpretation of the dendrogram. The dissimilarities computed in Step 2(b)

linkage

inversion

Linkage	*Description*
Complete	Maximal intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B, and record the *largest* of these dissimilarities.
Single	Minimal intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B, and record the *smallest* of these dissimilarities. Single linkage can result in extended, trailing clusters in which single observations are fused one-at-a-time.
Average	Mean intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B, and record the *average* of these dissimilarities.
Centroid	Dissimilarity between the centroid for cluster A (a mean vector of length p) and the centroid for cluster B. Centroid linkage can result in undesirable *inversions*.

TABLE 12.3. *A summary of the four most commonly-used types of linkage in hierarchical clustering.*

of the hierarchical clustering algorithm will depend on the type of linkage used, as well as on the choice of dissimilarity measure. Hence, the resulting dendrogram typically depends quite strongly on the type of linkage used, as is shown in Figure 12.14.

Choice of Dissimilarity Measure

Thus far, the examples in this chapter have used Euclidean distance as the dissimilarity measure. But sometimes other dissimilarity measures might be preferred. For example, *correlation-based distance* considers two observations to be similar if their features are highly correlated, even though the observed values may be far apart in terms of Euclidean distance. This is an unusual use of correlation, which is normally computed between variables; here it is computed between the observation profiles for each pair of observations. Figure 12.15 illustrates the difference between Euclidean and correlation-based distance. Correlation-based distance focuses on the shapes of observation profiles rather than their magnitudes.

The choice of dissimilarity measure is very important, as it has a strong effect on the resulting dendrogram. In general, careful attention should be paid to the type of data being clustered and the scientific question at hand. These considerations should determine what type of dissimilarity measure is used for hierarchical clustering.

For instance, consider an online retailer interested in clustering shoppers based on their past shopping histories. The goal is to identify subgroups of *similar* shoppers, so that shoppers within each subgroup can be shown items and advertisements that are particularly likely to interest them. Suppose the data takes the form of a matrix where the rows are the shoppers and the columns are the items available for purchase; the elements of the data matrix indicate the number of times a given shopper has purchased a

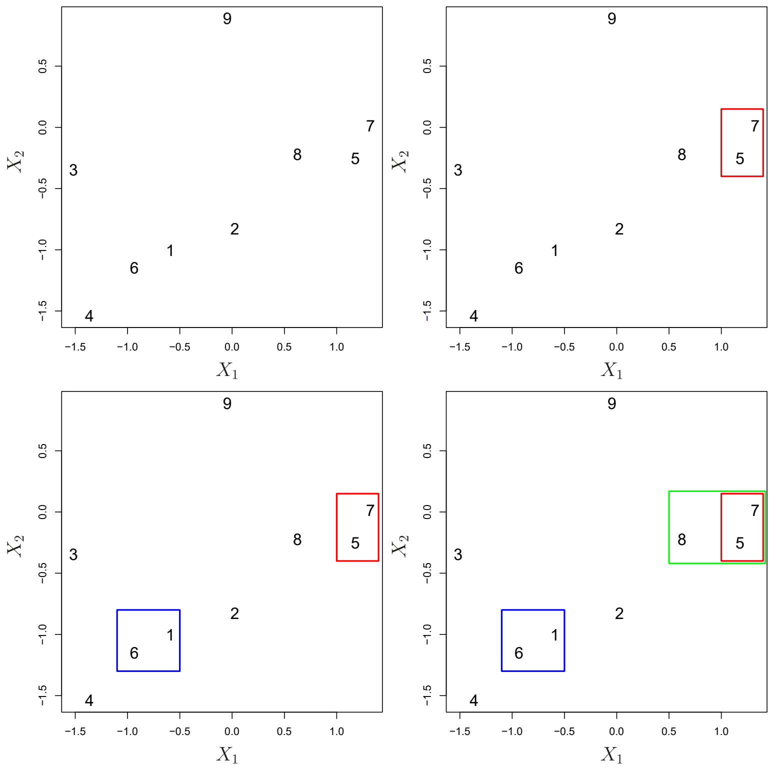


FIGURE 12.13. *An illustration of the first few steps of the hierarchical clustering algorithm, using the data from Figure 12.12, with complete linkage and Euclidean distance.* Top Left: *initially, there are nine distinct clusters,* $\{1\}, \{2\}, \ldots, \{9\}$. Top Right: *the two clusters that are closest together,* $\{5\}$ *and* $\{7\}$, *are fused into a single cluster.* Bottom Left: *the two clusters that are closest together,* $\{6\}$ *and* $\{1\}$, *are fused into a single cluster.* Bottom Right: *the two clusters that are closest together using* complete linkage, $\{8\}$ *and the cluster* $\{5, 7\}$, *are fused into a single cluster.*

given item (i.e. a 0 if the shopper has never purchased this item, a 1 if the shopper has purchased it once, etc.) What type of dissimilarity measure should be used to cluster the shoppers? If Euclidean distance is used, then shoppers who have bought very few items overall (i.e. infrequent users of the online shopping site) will be clustered together. This may not be desirable. On the other hand, if correlation-based distance is used, then shoppers with similar preferences (e.g. shoppers who have bought items A and B but never items C or D) will be clustered together, even if some shoppers with these preferences are higher-volume shoppers than others. Therefore, for this application, correlation-based distance may be a better choice.

In addition to carefully selecting the dissimilarity measure used, one must also consider whether or not the variables should be scaled to have standard deviation one before the dissimilarity between the observations is computed. To illustrate this point, we continue with the online shopping ex-

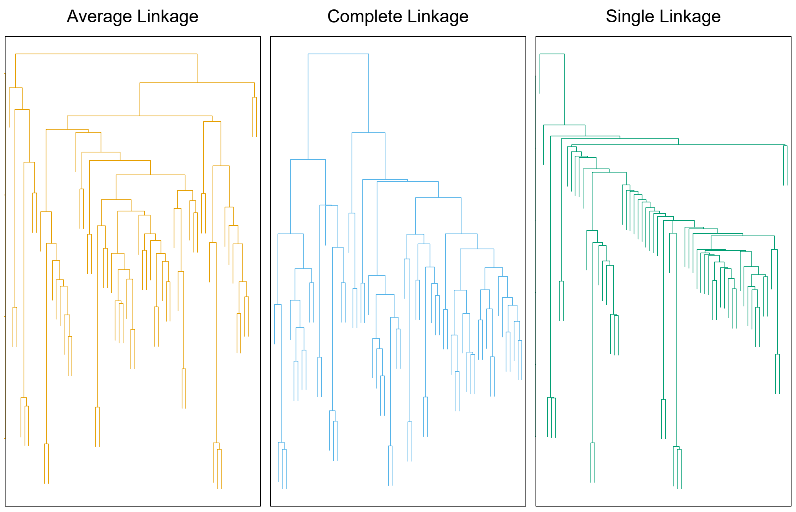


FIGURE 12.14. *Average, complete, and single linkage applied to an example data set. Average and complete linkage tend to yield more balanced clusters.*

ample just described. Some items may be purchased more frequently than others; for instance, a shopper might buy ten pairs of socks a year, but a computer very rarely. High-frequency purchases like socks therefore tend to have a much larger effect on the inter-shopper dissimilarities, and hence on the clustering ultimately obtained, than rare purchases like computers. This may not be desirable. If the variables are scaled to have standard deviation one before the inter-observation dissimilarities are computed, then each variable will in effect be given equal importance in the hierarchical clustering performed. We might also want to scale the variables to have standard deviation one if they are measured on different scales; otherwise, the choice of units (e.g. centimeters versus kilometers) for a particular variable will greatly affect the dissimilarity measure obtained. It should come as no surprise that whether or not it is a good decision to scale the variables before computing the dissimilarity measure depends on the application at hand. An example is shown in Figure 12.16. We note that the issue of whether or not to scale the variables before performing clustering applies to K-means clustering as well.

12.4.3 Practical Issues in Clustering

Clustering can be a very useful tool for data analysis in the unsupervised setting. However, there are a number of issues that arise in performing clustering. We describe some of these issues here.

Small Decisions with Big Consequences

In order to perform clustering, some decisions must be made.

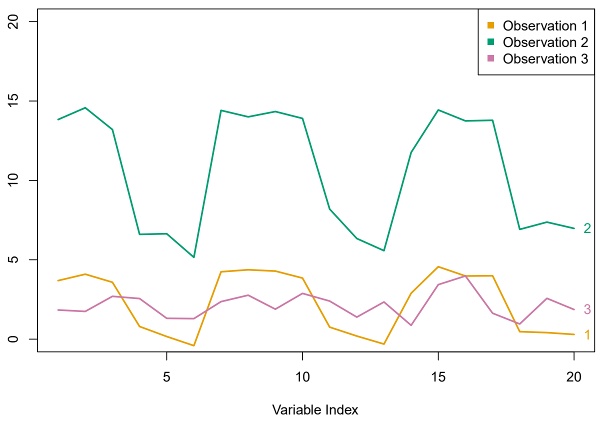


FIGURE 12.15. *Three observations with measurements on 20 variables are shown. Observations 1 and 3 have similar values for each variable and so there is a small Euclidean distance between them. But they are very weakly correlated, so they have a large correlation-based distance. On the other hand, observations 1 and 2 have quite different values for each variable, and so there is a large Euclidean distance between them. But they are highly correlated, so there is a small correlation-based distance between them.*

- Should the observations or features first be standardized in some way? For instance, maybe the variables should be scaled to have standard deviation one.
- In the case of hierarchical clustering,
  - What dissimilarity measure should be used?
  - What type of linkage should be used?
  - Where should we cut the dendrogram in order to obtain clusters?
- In the case of K-means clustering, how many clusters should we look for in the data?

Each of these decisions can have a strong impact on the results obtained. In practice, we try several different choices, and look for the one with the most useful or interpretable solution. With these methods, there is no single right answer—any solution that exposes some interesting aspects of the data should be considered.

Validating the Clusters Obtained

Any time clustering is performed on a data set we will find clusters. But we really want to know whether the clusters that have been found represent true subgroups in the data, or whether they are simply a result of *clustering the noise*. For instance, if we were to obtain an independent set of observations, then would those observations also display the same set of clusters? This is a hard question to answer. There exist a number of techniques for assigning a p-value to a cluster in order to assess whether there is more

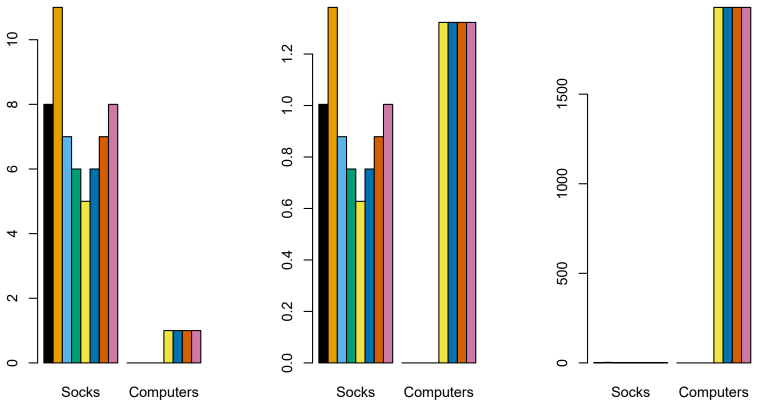


FIGURE 12.16. *An eclectic online retailer sells two items: socks and computers.* Left: *the number of pairs of socks, and computers, purchased by eight online shoppers is displayed. Each shopper is shown in a different color. If inter-observation dissimilarities are computed using Euclidean distance on the raw variables, then the number of socks purchased by an individual will drive the dissimilarities obtained, and the number of computers purchased will have little effect. This might be undesirable, since (1) computers are more expensive than socks and so the online retailer may be more interested in encouraging shoppers to buy computers than socks, and (2) a large difference in the number of socks purchased by two shoppers may be less informative about the shoppers' overall shopping preferences than a small difference in the number of computers purchased.* Center: *the same data are shown, after scaling each variable by its standard deviation. Now the two products will have a comparable effect on the inter-observation dissimilarities obtained.* Right: *the same data are displayed, but now the y-axis represents the number of dollars spent by each online shopper on socks and on computers. Since computers are much more expensive than socks, now computer purchase history will drive the inter-observation dissimilarities obtained.*

evidence for the cluster than one would expect due to chance. However, there has been no consensus on a single best approach. More details can be found in ESL.[8]

Other Considerations in Clustering

Both K-means and hierarchical clustering will assign each observation to a cluster. However, sometimes this might not be appropriate. For instance, suppose that most of the observations truly belong to a small number of (unknown) subgroups, and a small subset of the observations are quite different from each other and from all other observations. Then since K-means and hierarchical clustering force *every* observation into a cluster, the clusters found may be heavily distorted due to the presence of outliers that do not belong to any cluster. Mixture models are an attractive approach for accommodating the presence of such outliers. These amount to a *soft* version of K-means clustering, and are described in ESL.

[8]ESL: *The Elements of Statistical Learning* by Hastie, Tibshirani and Friedman.

In addition, clustering methods generally are not very robust to perturbations to the data. For instance, suppose that we cluster n observations, and then cluster the observations again after removing a subset of the n observations at random. One would hope that the two sets of clusters obtained would be quite similar, but often this is not the case!

A Tempered Approach to Interpreting the Results of Clustering

We have described some of the issues associated with clustering. However, clustering can be a very useful and valid statistical tool if used properly. We mentioned that small decisions in how clustering is performed, such as how the data are standardized and what type of linkage is used, can have a large effect on the results. Therefore, we recommend performing clustering with different choices of these parameters, and looking at the full set of results in order to see what patterns consistently emerge. Since clustering can be non-robust, we recommend clustering subsets of the data in order to get a sense of the robustness of the clusters obtained. Most importantly, we must be careful about how the results of a clustering analysis are reported. These results should not be taken as the absolute truth about a data set. Rather, they should constitute a starting point for the development of a scientific hypothesis and further study, preferably on an independent data set.

12.5 Lab: Unsupervised Learning

In this lab we demonstrate PCA and clustering on several datasets. As in other labs, we import some of our libraries at this top level. This makes the code more readable, as scanning the first few lines of the notebook tell us what libraries are used in this notebook.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.datasets import get_rdataset
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from ISLP import load_data
```

We also collect the new imports needed for this lab.

In [2]:
```
from sklearn.cluster import \
     (KMeans,
      AgglomerativeClustering)
from scipy.cluster.hierarchy import \
     (dendrogram,
      cut_tree)
from ISLP.cluster import compute_linkage
```

12.5.1 Principal Components Analysis

In this lab, we perform PCA on `USArrests`, a data set in the `R` computing environment. We retrieve the data using `get_rdataset()`, which can fetch

`get_rdataset()`