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*.

k-Means Algorithm: 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, and we will walk through it briefly here. In short, the expectation–maximization approach 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

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 location 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 5-112.

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.

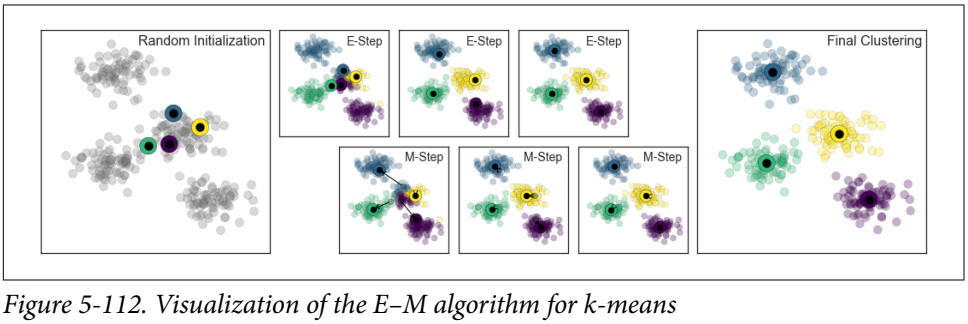

Figure 5-112. Visualization of the E–M algorithm for k-means

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 (Figure 5-113):

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

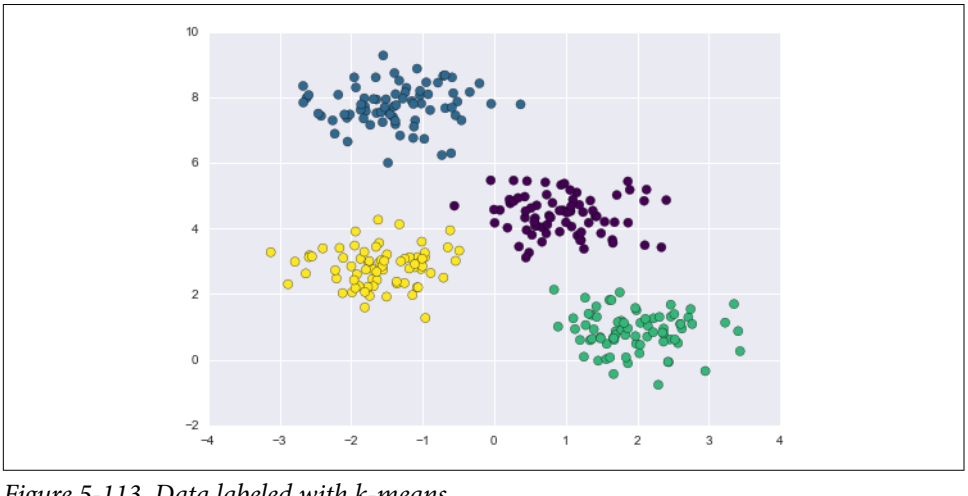

Figure 5-113. Data labeled with k-means

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.

Caveats of expectation–maximization

There are a few issues 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 (Figure 5-114):

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

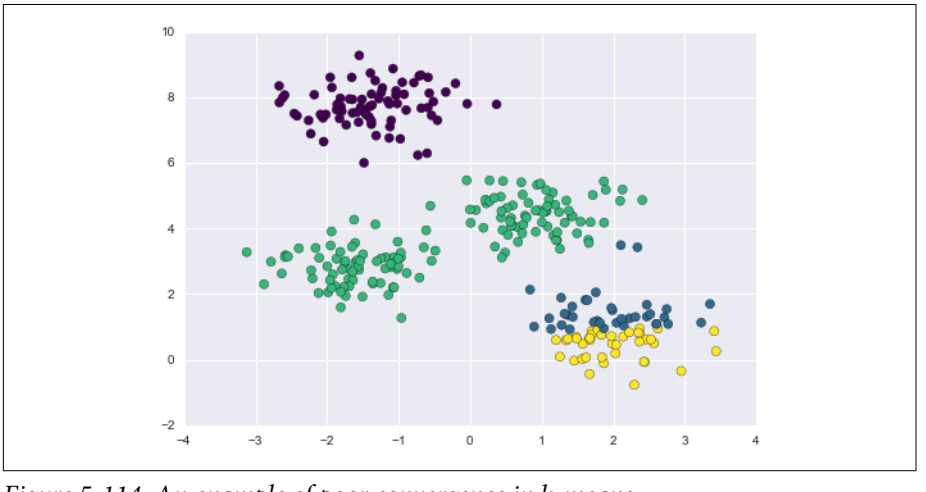

Figure 5-114. An example of poor convergence in k-means

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 (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 (Figure 5-115):

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

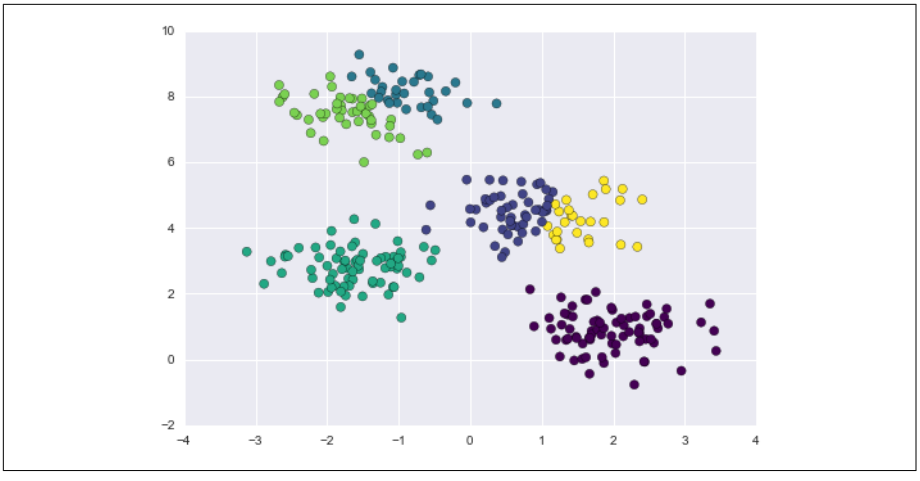

Figure 5-115. An example where the number of clusters is chosen poorly

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 which has a better quantitative measure of the fitness per number of clusters (e.g., Gaussian mixture models; see "In Depth: Gaussian Mixture Models" on page 476) 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 (Figure 5-116):

```
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');
```

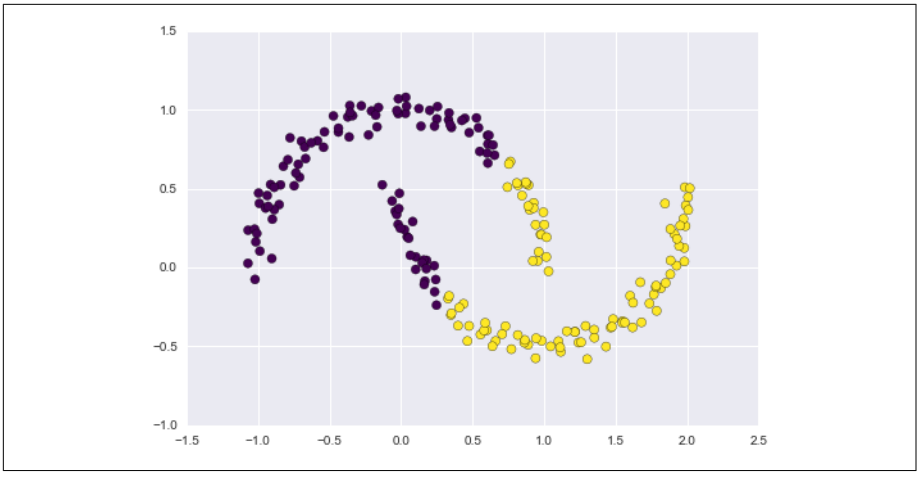

Figure 5-116. Failure of k-means with nonlinear boundaries

This situation is reminiscent of the discussion in "In-Depth: Support Vector Machines" on page 405, 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 nonlinear 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 (Figure 5-117):

```
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');
```

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

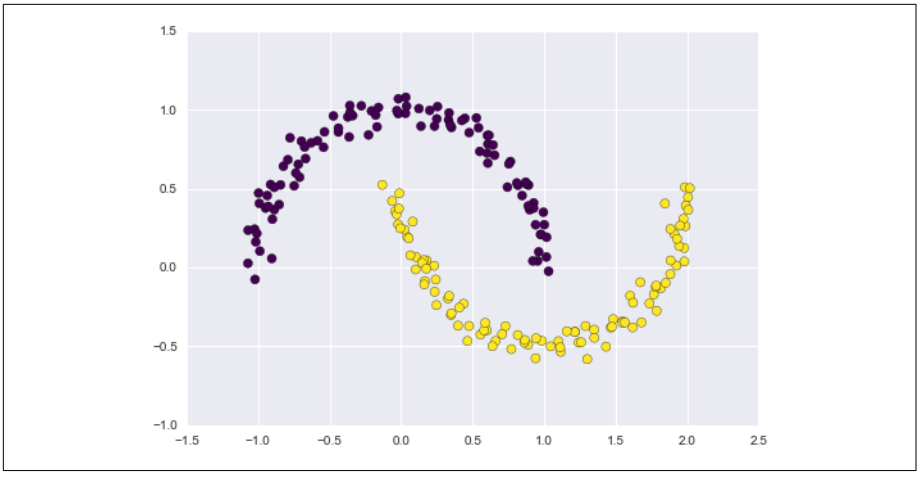

Figure 5-117. Nonlinear boundaries learned by SpectralClustering

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 wide variety of situations. We'll now take a look at a couple 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 "In-Depth: Decision Trees and Random Forests" on page 421 and "In Depth: Principal Component Analysis" on page 433. 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 digits and then finding the `KMeans` clusters. Recall that the digits consist of 1,797 samples with 64 features, where each of the 64 features is the brightness of one pixel in an 8×8 image: