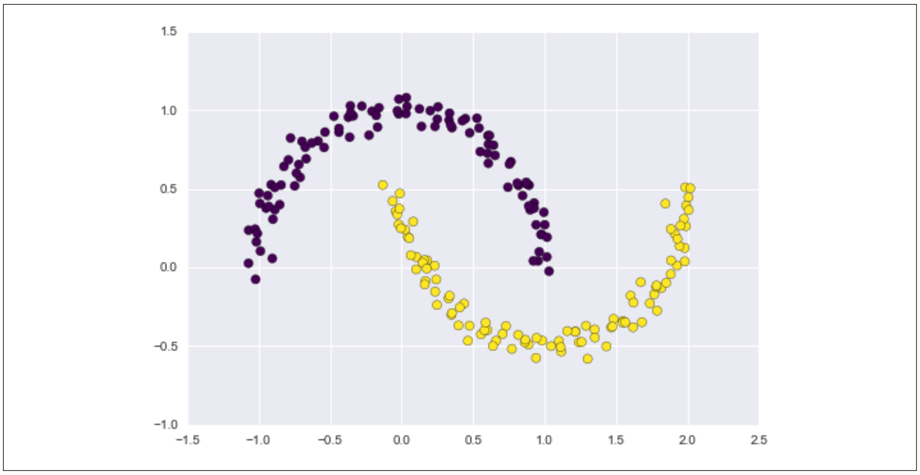


Figure 5-117. Nonlinear boundaries learned by SpectralClustering

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:

```
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 themselves be interpreted as the "typical" digit within the cluster. Let's see what these cluster centers look like (Figure 5-118):

```
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)
```

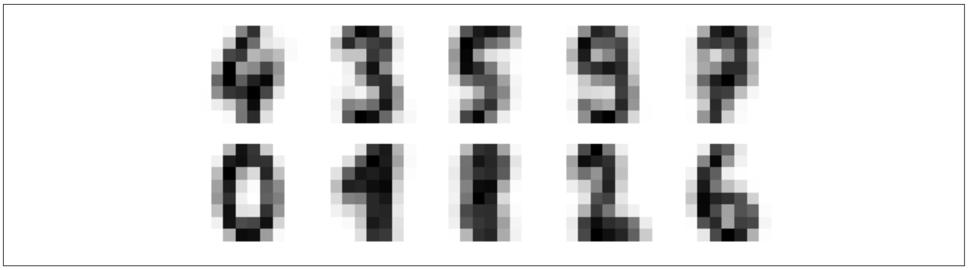


Figure 5-118. Cluster centers learned by k-means

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 identity of the cluster, the 0–9 labels may be permuted. We can fix this by matching each learned cluster label with the true labels found in them:

```
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]
```

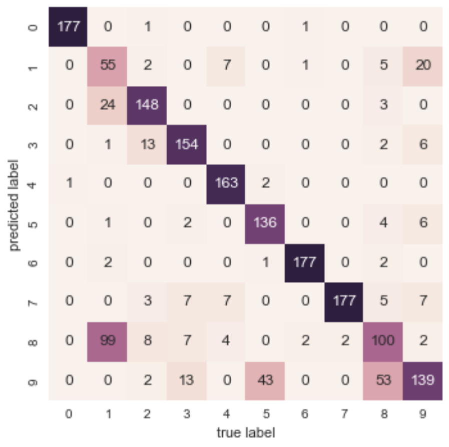
Now we can check how accurate our unsupervised clustering was in finding similar digits within the data:

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

Out[15]: 0.79354479688369506
```

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 (Figure 5-119):

```
In[16]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(digits.target, labels)
        sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                    xticklabels=digits.target_names,
                    yticklabels=digits.target_names)
        plt.xlabel('true label')
        plt.ylabel('predicted 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 5-119. 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 further. We can use the t-distributed stochastic neighbor embedding (t-SNE) algorithm (mentioned in "In-Depth: Manifold Learning" on page 445) 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='pca', 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]: 0.93356149137451305
```

That's nearly 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 do by hand or by eye.

Example 2: k-means for color compression

One interesting application of clustering is in color compression within images. 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 5-120, which is from Scikit-Learn's `datasets` module (for this to work, you'll have to have the `pillow` Python package installed):

```
In[18]: # Note: this requires the pillow 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);
```

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)
```

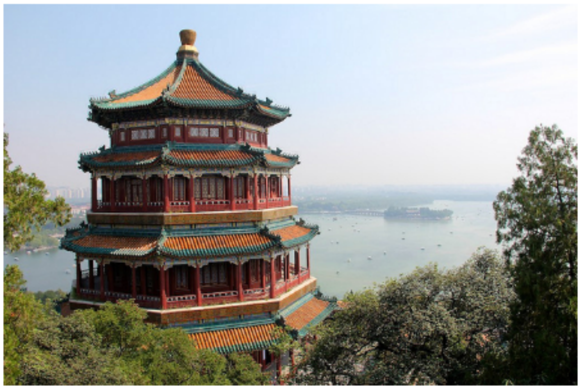


Figure 5-120. The input image

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 x 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(427 * 640, 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 (Figure 5-121):

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

            # choose a random subset
            rng = np.random.RandomState(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')
```

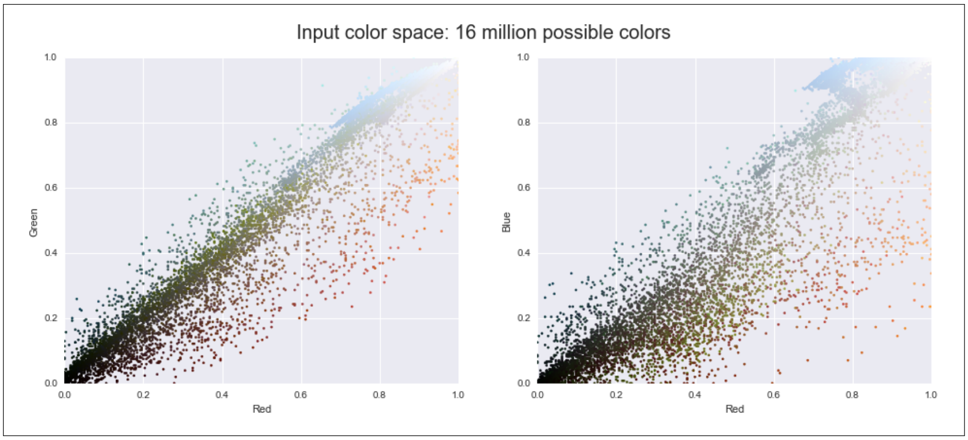


Figure 5-121. The distribution of the pixels in RGB color space

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 much more quickly than the standard *k*-means algorithm (Figure 5-122):

```
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")
```

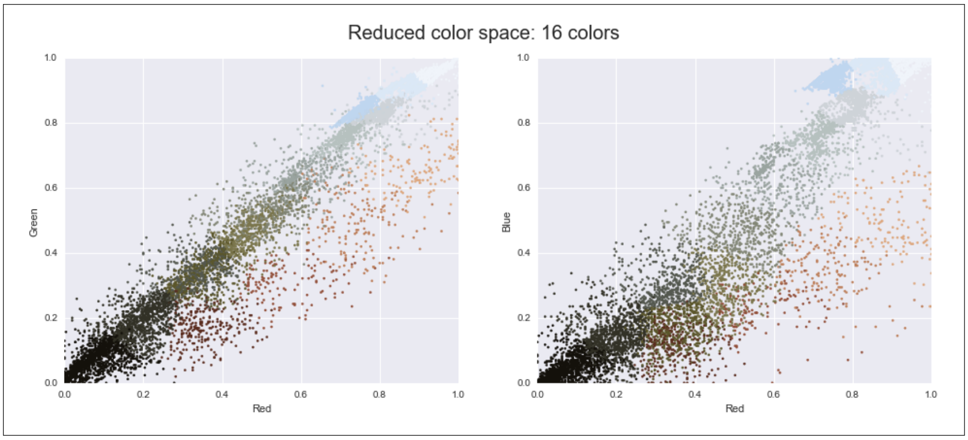


Figure 5-122. 16 clusters in RGB color space

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 (Figure 5-123):

```
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);
```

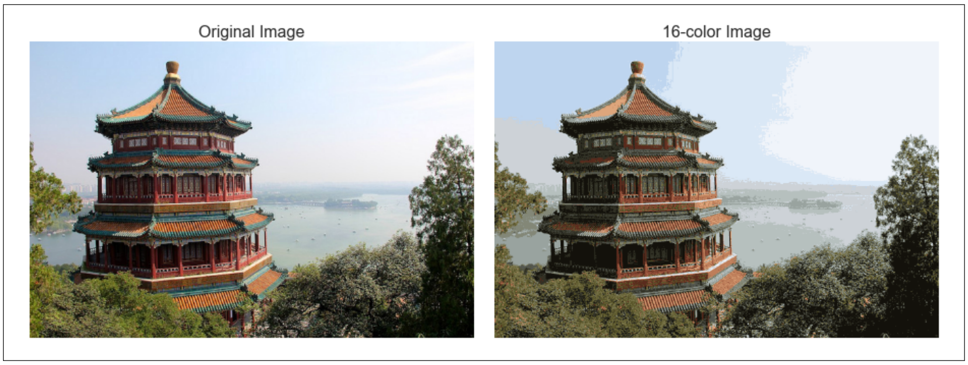


Figure 5-123. A comparison of the full-color image (left) and the 16-color image (right)

Some detail is certainly lost in the rightmost panel, but the overall image is still easily recognizable. This image on the right achieves a compression factor of around 1 million! While this is an interesting application of *k*-means, there are certainly better way to compress information in images. But the example shows the power of thinking outside of the box with unsupervised methods like *k*-means.

In Depth: Gaussian Mixture Models

The *k*-means clustering model explored in the previous section 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 section 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: