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);
```

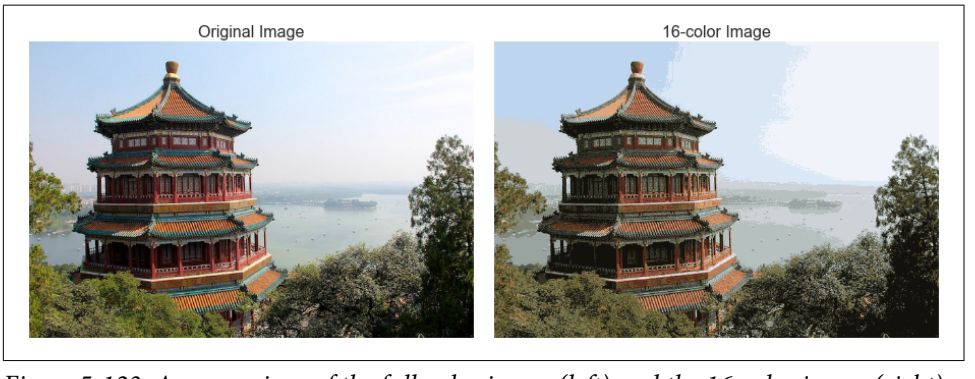

Figure 5-123. A comparison of the full-color image (left) and the 16-color image (right)

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:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Motivating GMM: 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 section, 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 (Figure 5-124):

```
In[2]: # Generate some data
       from sklearn.datasets.samples_generator 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');
```

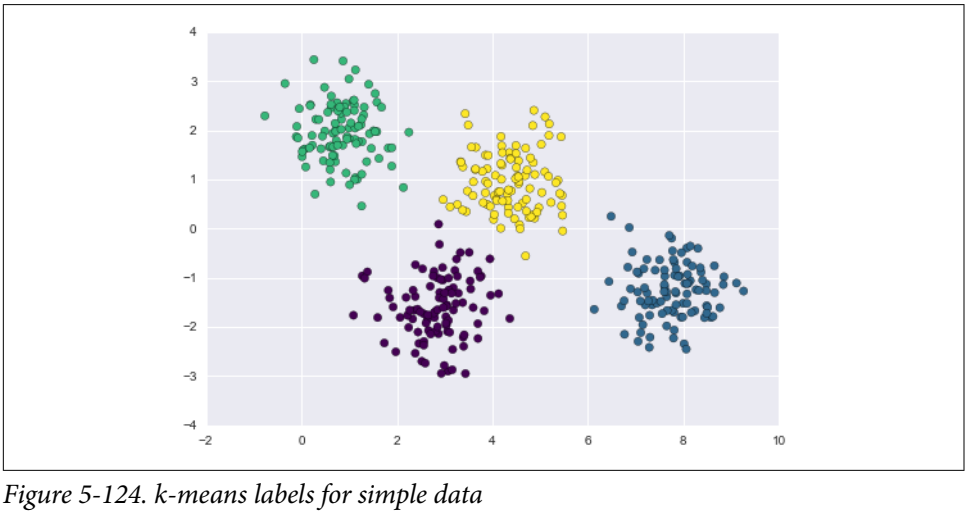

Figure 5-124. k-means labels for simple data

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