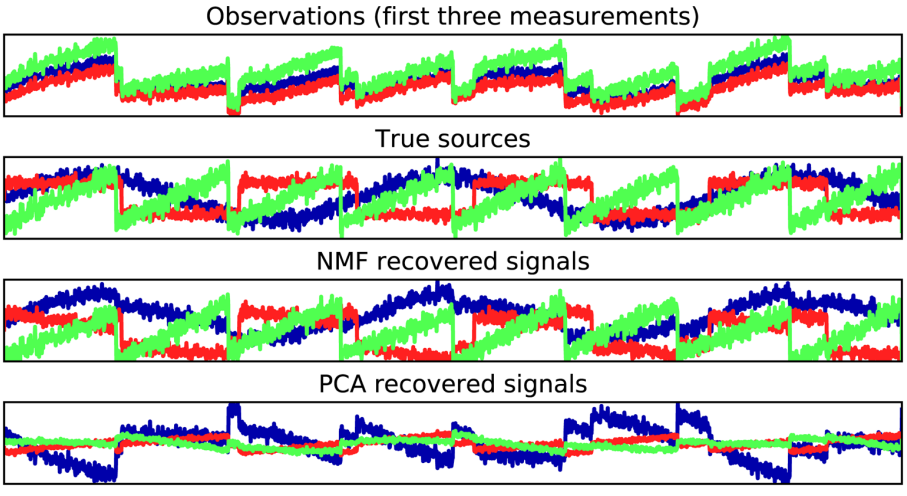

Figure 3-19. Recovering mixed sources using NMF and PCA

The figure includes 3 of the 100 measurements from `X` for reference. As you can see, NMF did a reasonable job of discovering the original sources, while PCA failed and used the first component to explain the majority of the variation in the data. Keep in mind that the components produced by NMF have no natural ordering. In this example, the ordering of the NMF components is the same as in the original signal (see the shading of the three curves), but this is purely accidental.

There are many other algorithms that can be used to decompose each data point into a weighted sum of a fixed set of components, as PCA and NMF do. Discussing all of them is beyond the scope of this book, and describing the constraints made on the components and coefficients often involves probability theory. If you are interested in this kind of pattern extraction, we recommend that you study the sections of the `scikit_learn` user guide on independent component analysis (ICA), factor analysis (FA), and sparse coding (dictionary learning), all of which you can find on the page about decomposition methods.

Manifold Learning with t-SNE

While PCA is often a good first approach for transforming your data so that you might be able to visualize it using a scatter plot, the nature of the method (applying a rotation and then dropping directions) limits its usefulness, as we saw with the scatter plot of the Labeled Faces in the Wild dataset. There is a class of algorithms for visualization called *manifold learning algorithms* that allow for much more complex mappings, and often provide better visualizations. A particularly useful one is the t-SNE algorithm.

Manifold learning algorithms are mainly aimed at visualization, and so are rarely used to generate more than two new features. Some of them, including t-SNE, compute a new representation of the training data, but don't allow transformations of new data. This means these algorithms cannot be applied to a test set: rather, they can only transform the data they were trained for. Manifold learning can be useful for exploratory data analysis, but is rarely used if the final goal is supervised learning. The idea behind t-SNE is to find a two-dimensional representation of the data that preserves the distances between points as best as possible. t-SNE starts with a random two-dimensional representation for each data point, and then tries to make points that are close in the original feature space closer, and points that are far apart in the original feature space farther apart. t-SNE puts more emphasis on points that are close by, rather than preserving distances between far-apart points. In other words, it tries to preserve the information indicating which points are neighbors to each other.

We will apply the t-SNE manifold learning algorithm on a dataset of handwritten digits that is included in `scikit-learn`.[2] Each data point in this dataset is an 8×8 gray-scale image of a handwritten digit between 0 and 1. Figure 3-20 shows an example image for each class:

In[43]:

```
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                         subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```

2 Not to be confused with the much larger MNIST dataset.

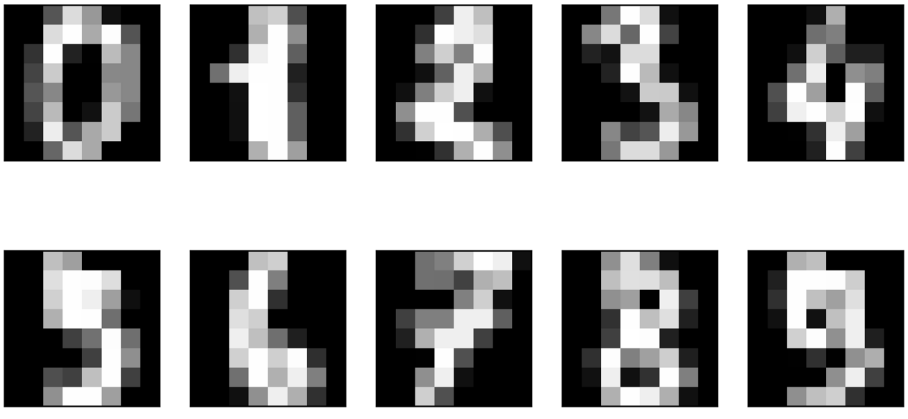

Figure 3-20. Example images from the digits dataset

Let's use PCA to visualize the data reduced to two dimensions. We plot the first two principal components, and color each dot by its class (see Figure 3-21):

In[44]:

```
# build a PCA model
pca = PCA(n_components=2)
pca.fit(digits.data)
# transform the digits data onto the first two principal components
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

Here, we actually used the true digit classes as glyphs, to show which class is where. The digits zero, six, and four are relatively well separated using the first two principal components, though they still overlap. Most of the other digits overlap significantly.

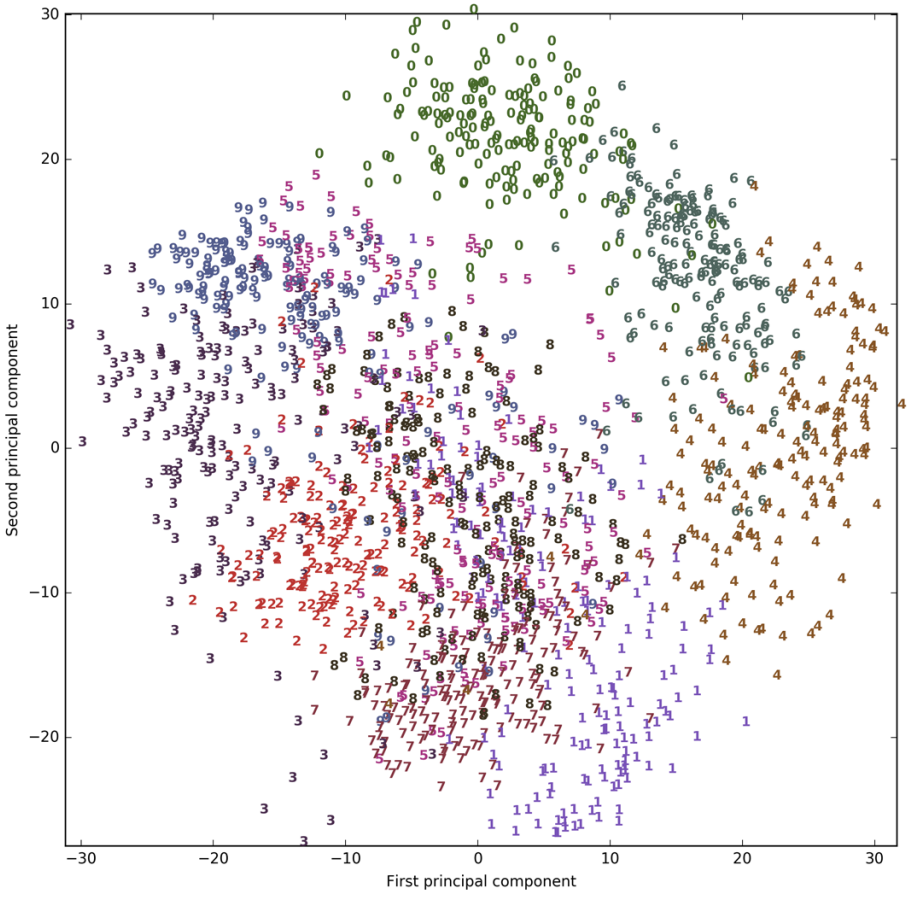

Figure 3-21. Scatter plot of the digits dataset using the first two principal components

Let's apply t-SNE to the same dataset, and compare the results. As t-SNE does not support transforming new data, the `TSNE` class has no `transform` method. Instead, we can call the `fit_transform` method, which will build the model and immediately return the transformed data (see Figure 3-22):

In[45]:

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# use fit_transform instead of fit, as TSNE has no transform method
digits_tsne = tsne.fit_transform(digits.data)
```

In[46]:

```
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE feature 0")
plt.xlabel("t-SNE feature 1")
```

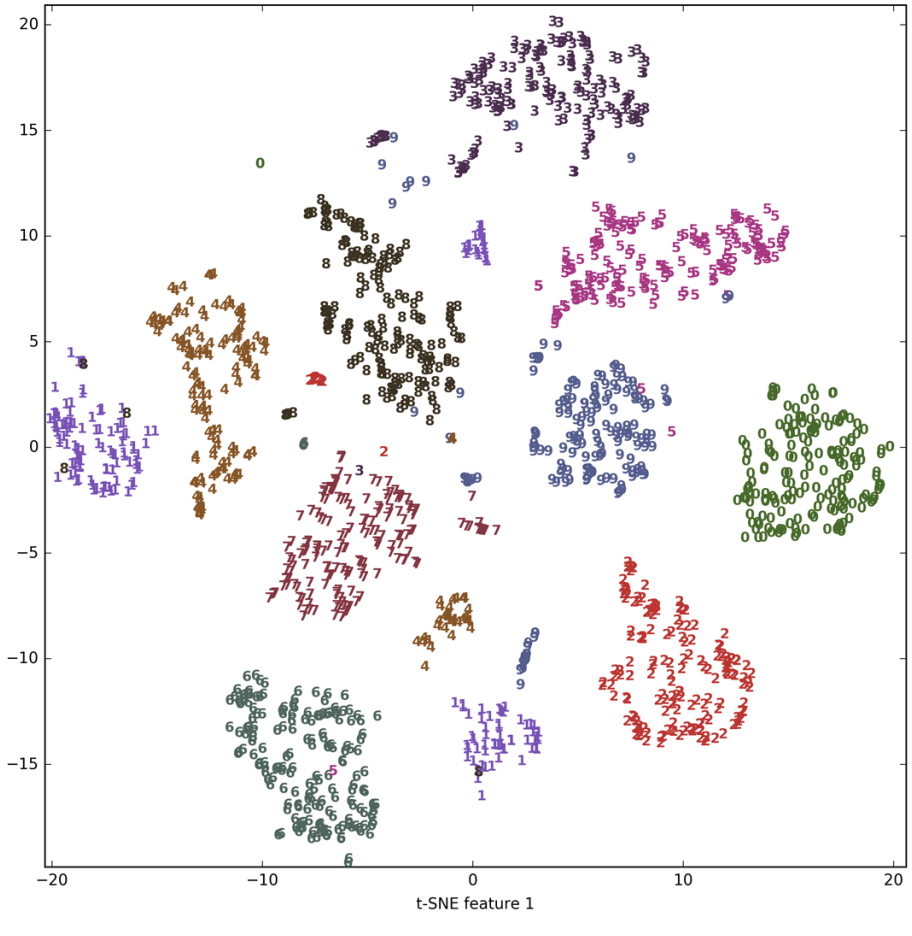

Figure 3-22. Scatter plot of the digits dataset using two components found by t-SNE

The result of t-SNE is quite remarkable. All the classes are quite clearly separated. The ones and nines are somewhat split up, but most of the classes form a single dense group. Keep in mind that this method has no knowledge of the class labels: it is completely unsupervised. Still, it can find a representation of the data in two dimensions that clearly separates the classes, based solely on how close points are in the original space.

The t-SNE algorithm has some tuning parameters, though it often works well with the default settings. You can try playing with `perplexity` and `early_exaggeration`, but the effects are usually minor.

Clustering

As we described earlier, *clustering* is the task of partitioning the dataset into groups, called clusters. The goal is to split up the data in such a way that points within a single cluster are very similar and points in different clusters are different. Similarly to classification algorithms, clustering algorithms assign (or predict) a number to each data point, indicating which cluster a particular point belongs to.

k-Means Clustering

k-means clustering is one of the simplest and most commonly used clustering algorithms. It tries to find *cluster centers* that are representative of certain regions of the data. The algorithm alternates between two steps: assigning each data point to the closest cluster center, and then setting each cluster center as the mean of the data points that are assigned to it. The algorithm is finished when the assignment of instances to clusters no longer changes. The following example (Figure 3-23) illustrates the algorithm on a synthetic dataset:

`In[47]:`

```
mglearn.plots.plot_kmeans_algorithm()
```