```
    X[:, 0] *= (data.shape[0] / data.shape[1])
    X = X[:N]
    return X[np.argsort(X[:, 0])]
```

Let's call the function and visualize the resulting data (Figure 5-94):

```
In[3]: X = make_hello(1000)
       colorize = dict(c=X[:, 0], cmap=plt.cm.get_cmap('rainbow', 5))
       plt.scatter(X[:, 0], X[:, 1], **colorize)
       plt.axis('equal');
```

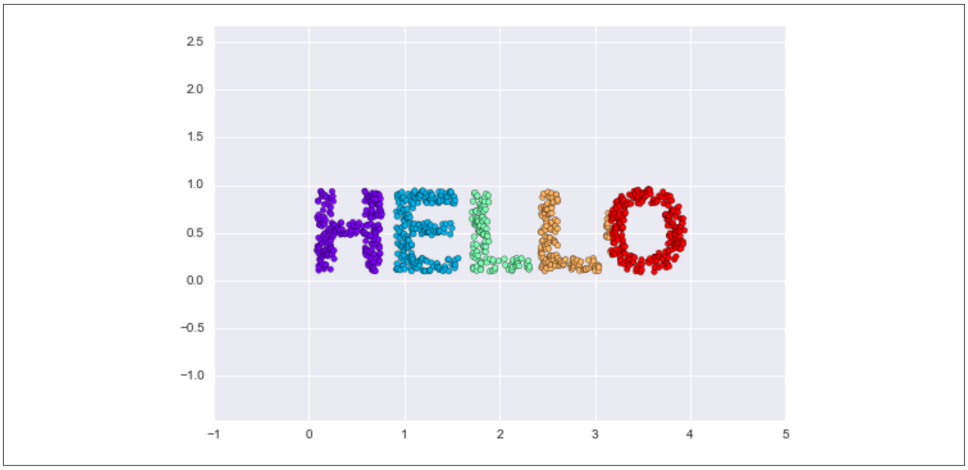

Figure 5-94. Data for use with manifold learning

The output is two dimensional, and consists of points drawn in the shape of the word "HELLO". This data form will help us to see visually what these algorithms are doing.

Multidimensional Scaling (MDS)

Looking at data like this, we can see that the particular choice of *x* and *y* values of the dataset are not the most fundamental description of the data: we can scale, shrink, or rotate the data, and the "HELLO" will still be apparent. For example, if we use a rotation matrix to rotate the data, the *x* and *y* values change, but the data is still fundamentally the same (Figure 5-95):

```
In[4]: def rotate(X, angle):
           theta = np.deg2rad(angle)
           R = [[np.cos(theta), np.sin(theta)],
                [-np.sin(theta), np.cos(theta)]]
           return np.dot(X, R)

       X2 = rotate(X, 20) + 5
       plt.scatter(X2[:, 0], X2[:, 1], **colorize)
       plt.axis('equal');
```

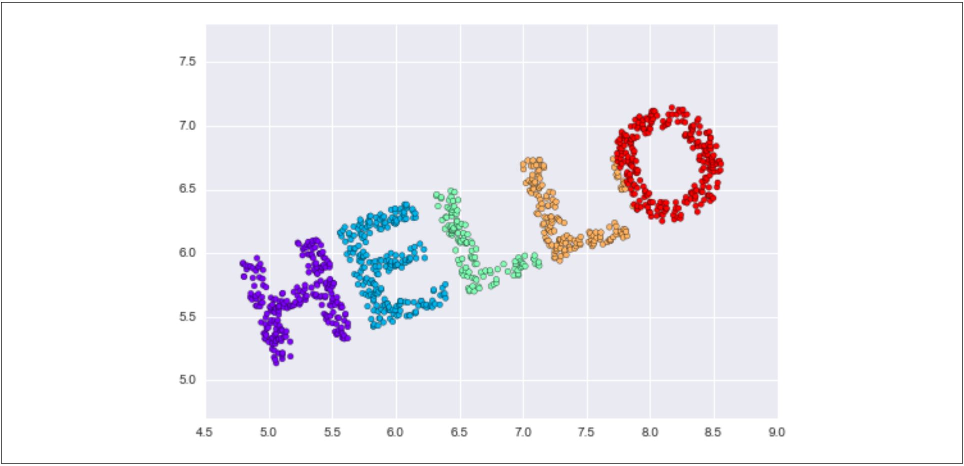

Figure 5-95. Rotated dataset

This tells us that the *x* and *y* values are not necessarily fundamental to the relationships in the data. What *is* fundamental, in this case, is the *distance* between each point and the other points in the dataset. A common way to represent this is to use a distance matrix: for *N* points, we construct an $N \times N$ array such that entry (i, j) contains the distance between point *i* and point *j*. Let's use Scikit-Learn's efficient `pairwise_distances` function to do this for our original data:

```
In[5]: from sklearn.metrics import pairwise_distances
       D = pairwise_distances(X)
       D.shape
```

```
Out[5]: (1000, 1000)
```

As promised, for our *N*=1,000 points, we obtain a 1,000×1,000 matrix, which can be visualized as shown in Figure 5-96:

```
In[6]: plt.imshow(D, zorder=2, cmap='Blues', interpolation='nearest')
       plt.colorbar();
```

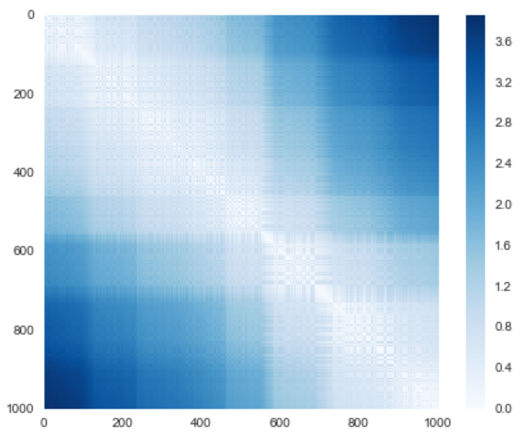

Figure 5-96. Visualization of the pairwise distances between points

If we similarly construct a distance matrix for our rotated and translated data, we see that it is the same:

```
In[7]: D2 = pairwise_distances(X2)
       np.allclose(D, D2)
```

```
Out[7]: True
```

This distance matrix gives us a representation of our data that is invariant to rotations and translations, but the visualization of the matrix is not entirely intuitive. In the representation presented in Figure 5-96, we have lost any visible sign of the interesting structure in the data: the "HELLO" that we saw before.

Further, while computing this distance matrix from the (x, y) coordinates is straightforward, transforming the distances back into *x* and *y* coordinates is rather difficult. This is exactly what the multidimensional scaling algorithm aims to do: given a distance matrix between points, it recovers a *D*-dimensional coordinate representation of the data. Let's see how it works for our distance matrix, using the `precomputed` dissimilarity to specify that we are passing a distance matrix (Figure 5-97):

```
In[8]: from sklearn.manifold import MDS
       model = MDS(n_components=2, dissimilarity='precomputed', random_state=1)
       out = model.fit_transform(D)
       plt.scatter(out[:, 0], out[:, 1], **colorize)
       plt.axis('equal');
```

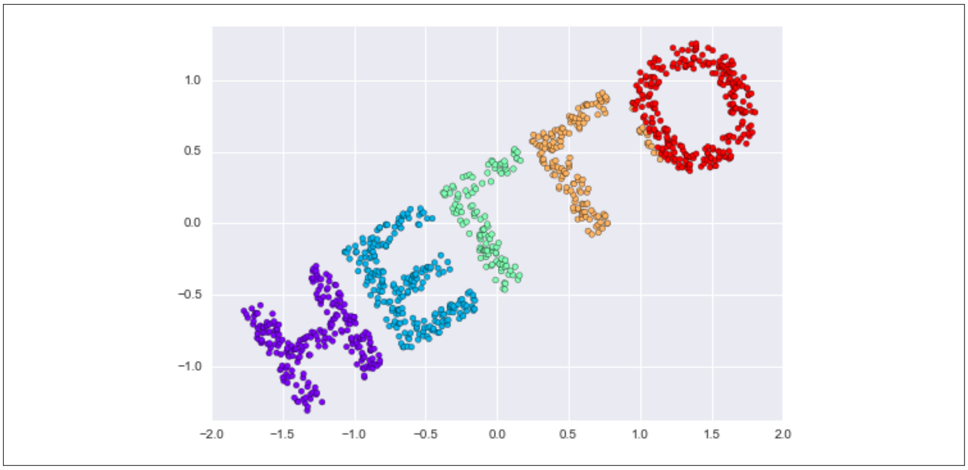

Figure 5-97. An MDS embedding computed from the pairwise distances

The MDS algorithm recovers one of the possible two-dimensional coordinate representations of our data, using *only* the $N \times N$ distance matrix describing the relationship between the data points.

MDS as Manifold Learning

The usefulness of this becomes more apparent when we consider the fact that distance matrices can be computed from data in *any* dimension. So, for example, instead of simply rotating the data in the two-dimensional plane, we can project it into three dimensions using the following function (essentially a three-dimensional generalization of the rotation matrix used earlier):

```
In[9]: def random_projection(X, dimension=3, rseed=42):
           assert dimension >= X.shape[1]
           rng = np.random.RandomState(rseed)
           C = rng.randn(dimension, dimension)
           e, V = np.linalg.eigh(np.dot(C, C.T))
           return np.dot(X, V[:X.shape[1]])

       X3 = random_projection(X, 3)
       X3.shape

Out[9]: (1000, 3)
```

Let's visualize these points to see what we're working with (Figure 5-98):

```
In[10]: from mpl_toolkits import mplot3d
        ax = plt.axes(projection='3d')
        ax.scatter3D(X3[:, 0], X3[:, 1], X3[:, 2],
                     **colorize)
        ax.view_init(azim=70, elev=50)
```