Python for Machine Learning and Data Analysis

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Spring 2019

Machine Learning 05

Lecture 11

Content

- Principal Component Analysis

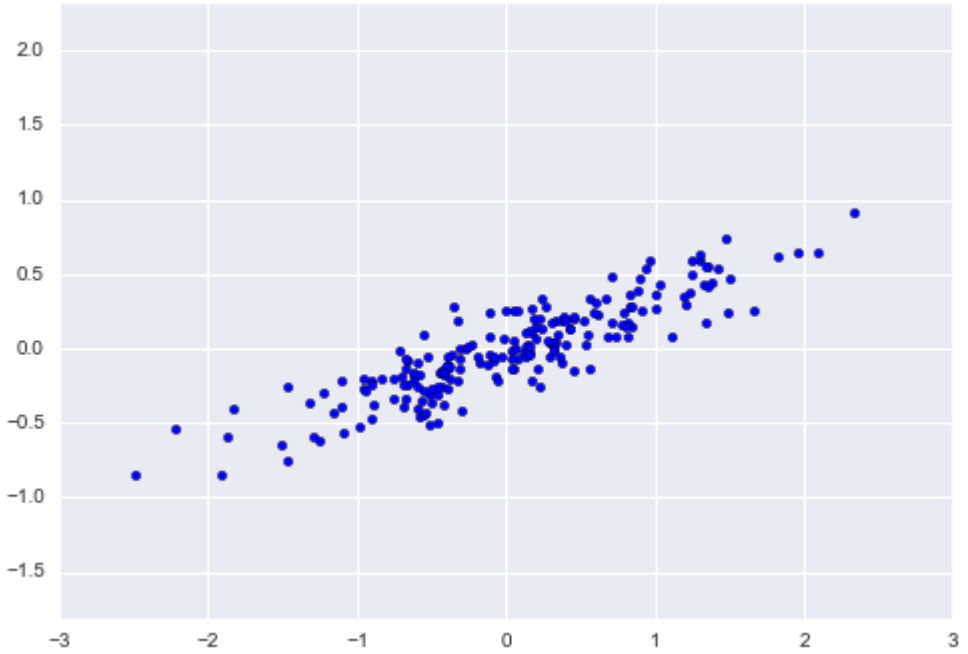
- Manifold Learning

Principal Component Analysis

1. Introducing Principal Component Analysis

- Principal Component Analysis (PCA) is perhaps one of the most broadly used of unsupervised algorithms.
- PCA is fundamentally a dimensionality reduction algorithm, but it can also be useful as a tool for visualization, for noise filtering, for feature extraction and engineering, and much more.
- After a brief conceptual discussion of the PCA algorithm, we will see a couple examples of these further applications.

- Principal component analysis is a fast and flexible unsupervised method for dimensionality reduction in data.
- Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider the following 200 points:



It is clear that there is a nearly linear relationship between the *x* and *y* variables. This is reminiscent of the linear regression data we explored in “Linear Regression”, but the problem setting here is slightly different: rather than attempting to predict the y values from the x values, the unsupervised learning problem attempts to learn about the relationship between the x and y values.

- In principal component analysis, this relationship is quantified by finding a list of the principal axes in the data, and using those axes to describe the dataset. Using Scikit-Learn's PCA estimator, we can compute this.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
```

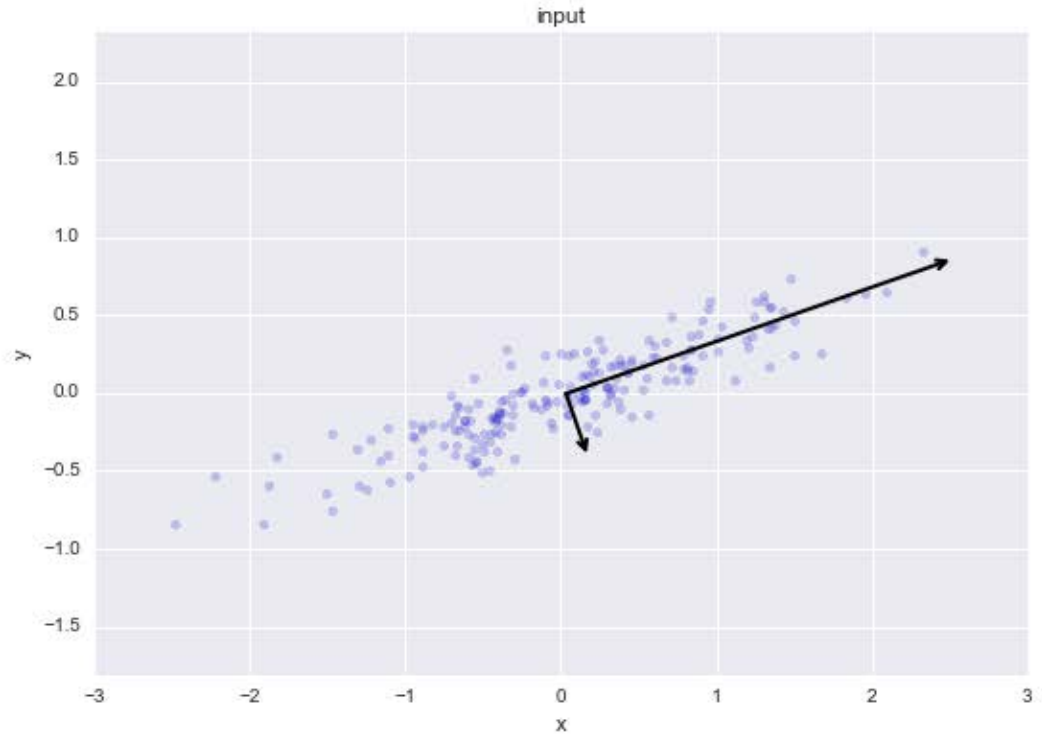
- The fit learns some quantities from the data, most importantly the "components" and "explained variance":

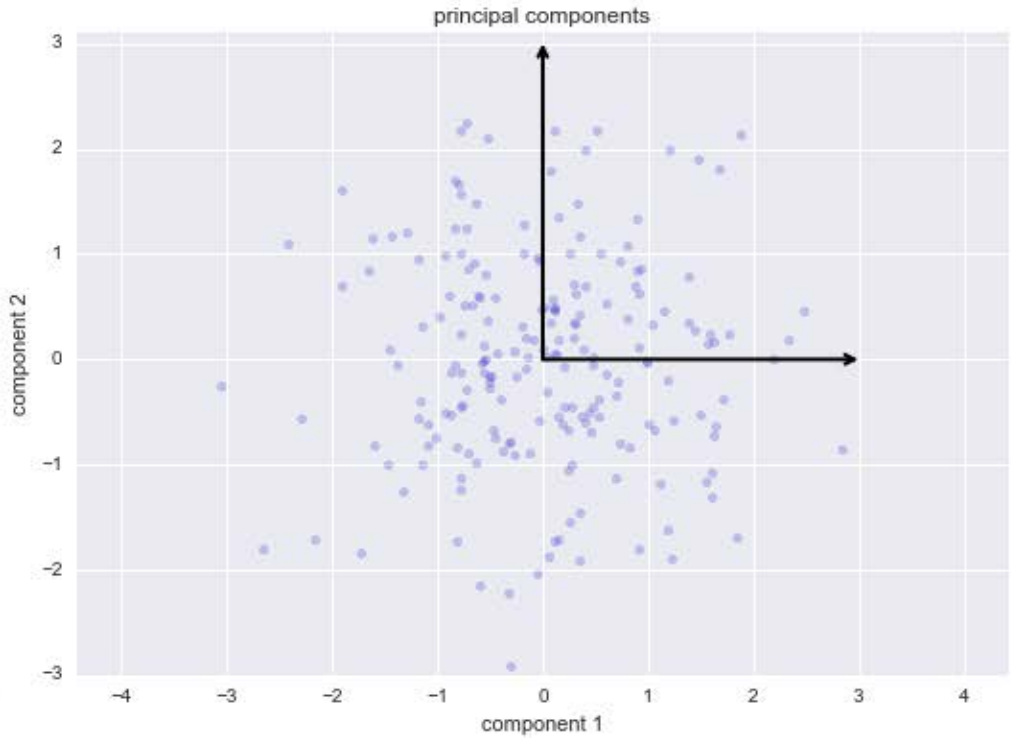
```
print(pca.components_)
[[ 0.94446029  0.32862557]
 [ 0.32862557 -0.94446029]]

print(pca.explained_variance_)
[ 0.75871884  0.01838551]
```

To see what these numbers mean, let's visualize them as vectors over the input data, using the "components" to define the direction of the vector, and the "explained variance" to define the squared-length of the vector:

- These vectors represent the principal axes of the data, and the length of the vector is an indication of how “important” that axis is in describing the distribution of the data。
- More precisely, it is a measure of the variance of the data when projected onto that axis. The projection of each data point onto the principal axes are the "principal components" of the data.





This transformation from data axes to principal axes is an affine transformation, which basically means it is composed of a translation, rotation, and uniform scaling.

While this algorithm to find principal components may seem like just a mathematical curiosity, it turns out to have very far-reaching applications in the world of machine learning and data exploration.

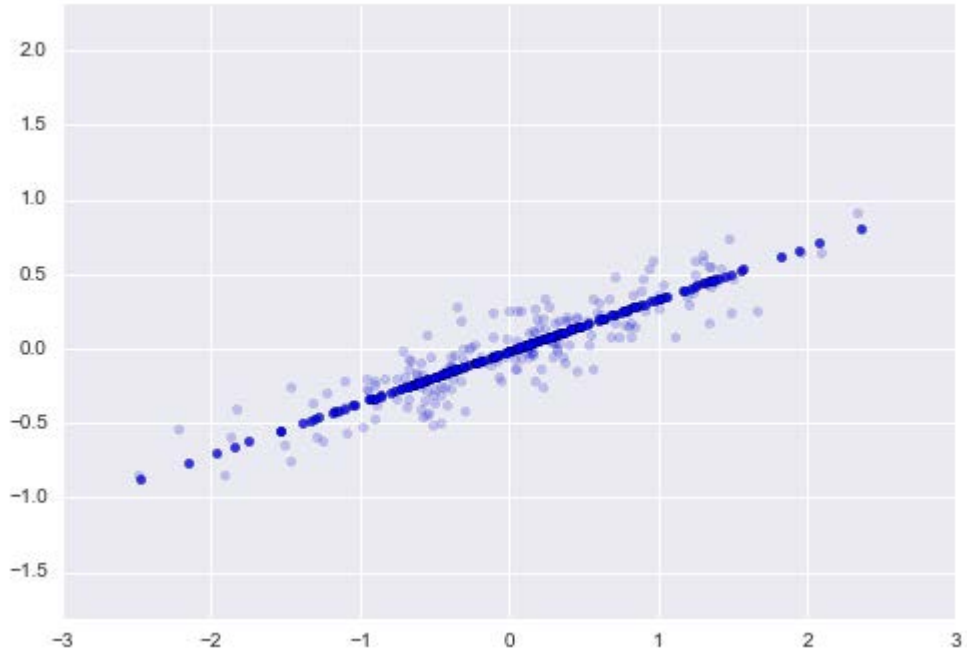
1.1 PCA as dimensionality reduction

- Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of the data that preserves the maximal data variance.

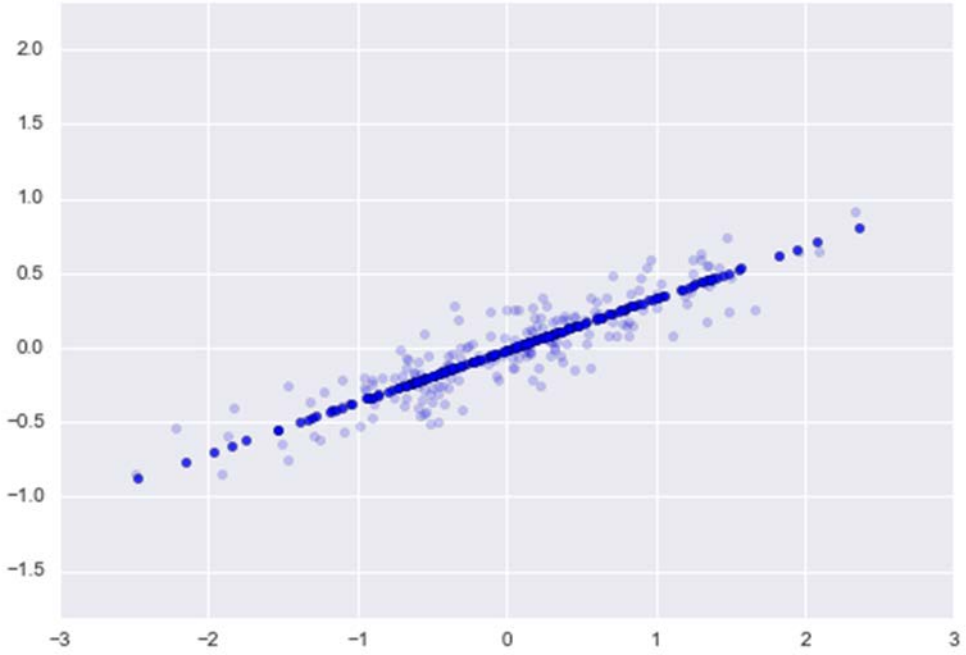
```
pca = PCA(n_components=1)
pca.fit(X);  X_pca = pca.transform(X)
original shape:    (200, 2)       transformed shape: (200, 1)
```

- The transformed data has been reduced to a single dimension. To understand the effect of this dimensionality reduction, we can perform the inverse transform of this reduced data and plot it along with the original data:

```
X_new = pca.inverse_transform(X_pca)
```



- The light points are the original data, while the dark points are the projected version.
- This makes clear what a PCA dimensionality reduction means: the information along the least important principal axis or axes is removed, leaving only the component(s) of the data with the highest variance.
- The fraction of variance that is cut out is roughly a measure of how much "information" is discarded in this reduction of dimensionality.
- The fraction of variance is proportional to the spread of points about the line formed in this figure.



This reduced-dimension dataset is in some senses "good enough" to encode the most important relationships between the points: despite reducing the dimension of the data by 50%, the overall relationship between the data points are mostly preserved.

1.2 PCA for visualization

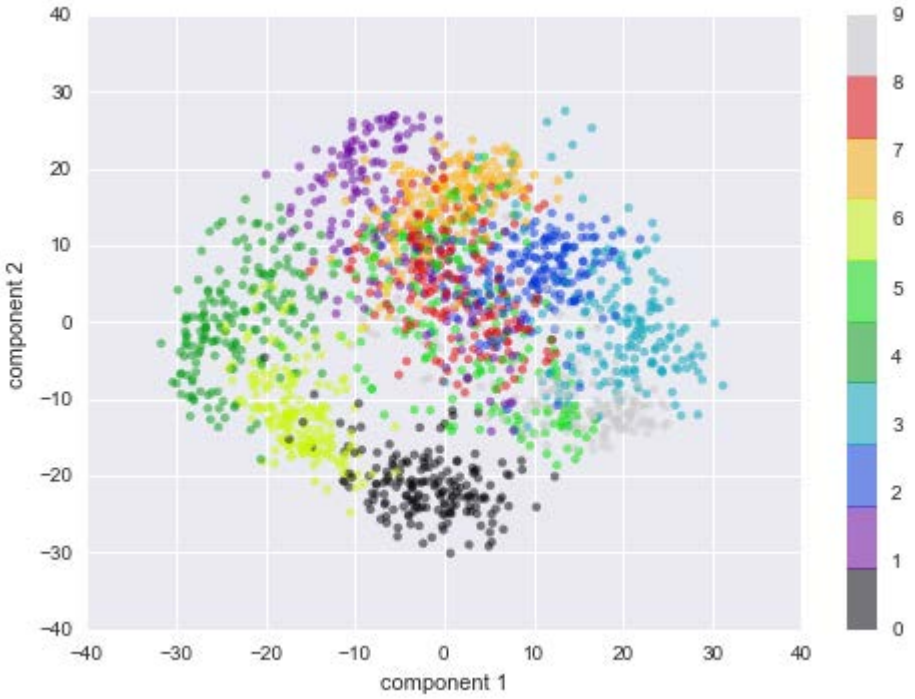
- The usefulness of the dimensionality reduction may not be entirely apparent in only two dimensions, but becomes much more clear when looking at high-dimensional data.

Example: Hand-written digits

- Loading the data: shape is (1797, 64)
- An image is 8x8 pixel as 64-dimension, use PCA to project them to 2-dimensions

We can now plot the first two principal components of each point to learn about the data:

- Recall what these components mean: the full data is a 64-dimensional point scatter, and these points are the projection of each data point along the directions with the largest variance.
- we have found the optimal stretch and rotation in 64-dimensional space that allows us to see the layout of the digits in two dimensions, and have done this in an unsupervised manner—that is, without reference to the labels.



What do the components mean?

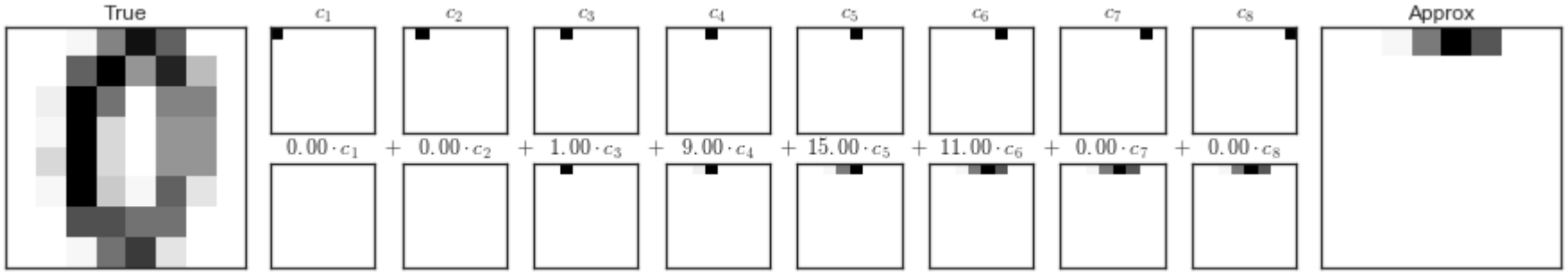
- The meaning can be understood in terms of combinations of basis vectors.

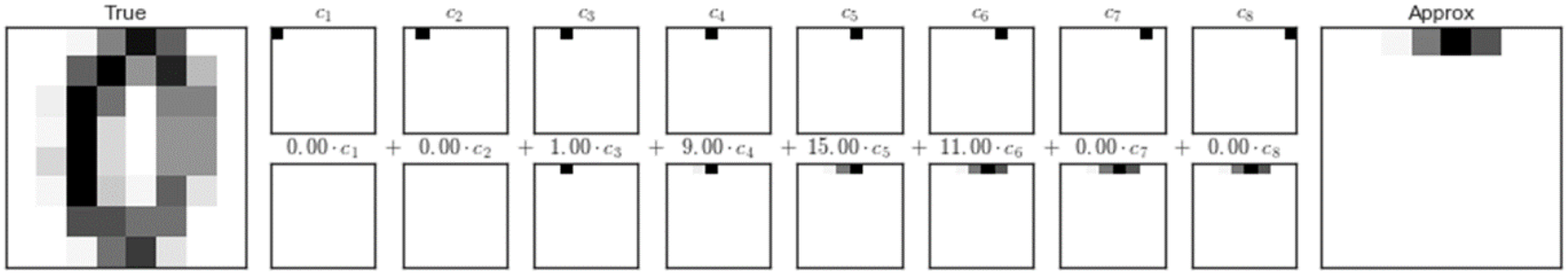
$$x = [x_1, x_2, x_3, \cdots, x_{64}]$$

- One way we can think about this is in terms of a pixel basis. That is, to construct the image, we multiply each element of the vector by the pixel it describes, and then add the results together to build the image:

$$image(x) = x_1 \cdot (pixel{\sim}1) + x_2 \cdot (pixel{\sim}2) + x_3 \cdot (pixel{\sim}3) \cdots x_{64} \cdot (pixel{\sim}64)$$

- One way we might imagine reducing the dimension of this data is to zero out all but a few of these basis vectors. For example, if we use only the first eight pixels, we get an eight-dimensional projection of the data, but it is not very reflective of the whole image: we've thrown out nearly 90% of the pixels!



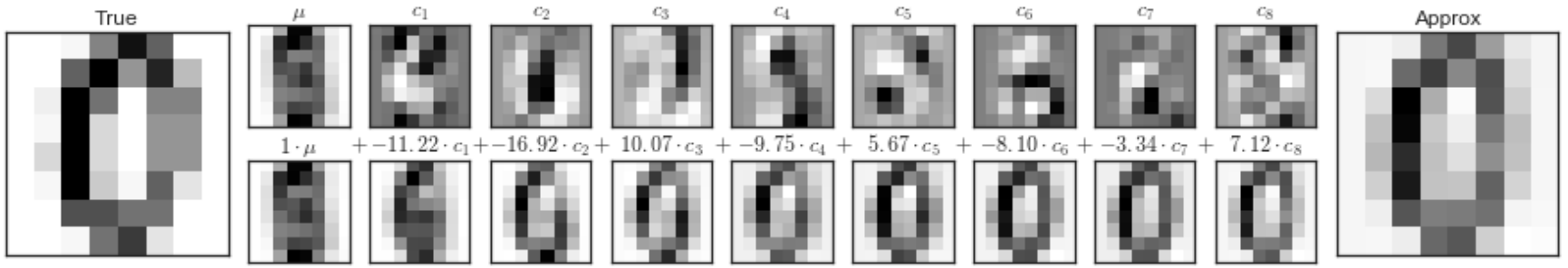


The upper row of panels shows the individual pixels, and the lower row shows the cumulative contribution of these pixels to the construction of the image. Using only eight of the pixel-basis components, we can only construct a small portion of the 64-pixel image. Were we to continue this sequence and use all 64 pixels, we would recover the original image.

But the pixel-wise representation is not the only choice of basis. We can also use other basis functions, which each contain some pre-defined contribution from each pixel, and write something like

$$image(x) = mean + x_1 \cdot (basis{\sim}1) + x_2 \cdot (basis{\sim}2) + x_3 \cdot (basis{\sim}3) \cdots$$

- PCA can be thought of as a process of choosing optimal basis functions, such that adding together just the first few of them is enough to suitably reconstruct the bulk of the elements in the dataset.
- The principal components, which act as the low-dimensional representation of our data, are simply the coefficients that multiply each of the elements in this series. This figure shows a similar depiction of reconstructing this digit using the mean plus the first eight PCA basis functions:



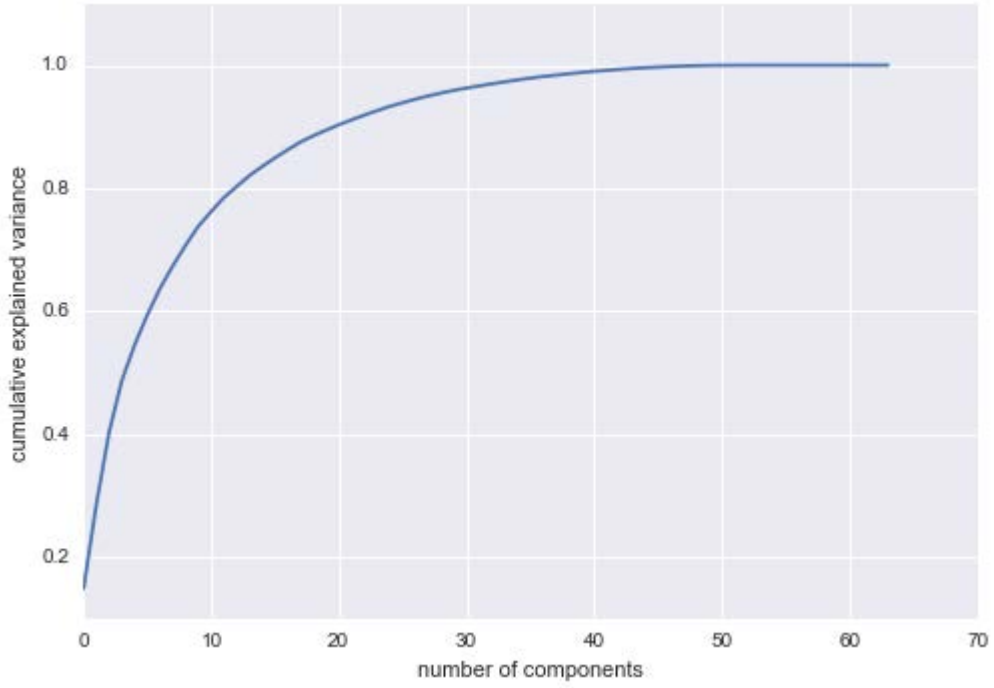
- Unlike the pixel basis, the PCA basis allows us to recover the salient features of the input image with just a mean plus eight components!
- The amount of each pixel in each component is the corollary of the orientation of the vector in our two-dimensional example.
- This is the sense in which PCA provides a low-dimensional representation of the data: it discovers a set of basis functions that are more efficient than the native pixel-basis of the input data.

Choosing the number of components

- A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data.
- This can be determined by looking at the cumulative explained variance ratio as a function of the number of components:

This curve quantifies how much of the total, 64-dimensional variance is contained within the first N components.

For example, we see that with the digits the first 10 components contain approximately 75% of the variance, while you need around 50 components to describe close to 100% of the variance.



1.3 PCA as Noise Filtering

- PCA can also be used as a filtering approach for noisy data.

The idea is this:

- any components with variance much larger than the effect of the noise should be relatively unaffected by the noise. So if you reconstruct the data using just the largest subset of principal components, you should be preferentially keeping the signal and throwing out the noise.

This attribute of signal preserving/noise filtering makes PCA a very useful feature-selection routine.

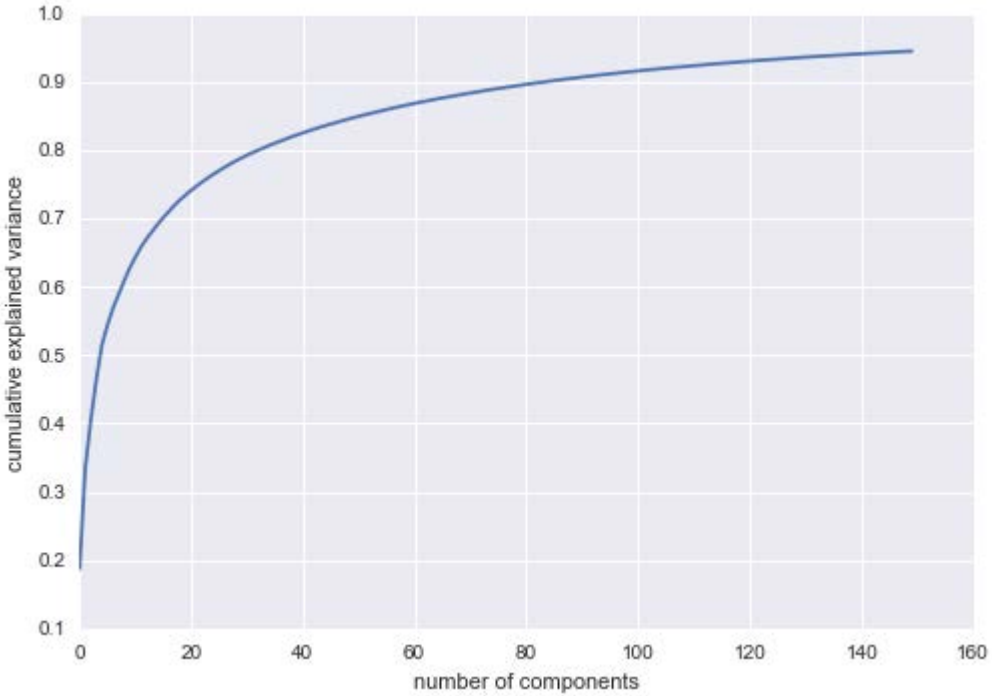
- for example, rather than training a classifier on very high-dimensional data, you might train the classifier on the lower-dimensional representation, which will automatically serve to filter out random noise in the inputs.

Example: Eigenfaces

- We've explored an example of using a PCA projection as a feature selection for facial recognition with a SVM.
- Recall that we were using the Labeled Faces in the Wild dataset made available through Scikit-Learn.
- In this case, it can be interesting to visualize the images associated with the first several principal components. These components are technically known as "eigenvectors," so these types of images are often called "eigenfaces".
- Because this is a large dataset, use RandomizedPCA — it contains a randomized method to approximate the first *N* principal components much more quickly than the standard PCA estimator, and thus is very useful for high-dimensional data (here, a dimensionality of nearly 3,000). We will take a look at the first 150 components:

The results are very interesting, and give us insight into how the images vary: for example,

- the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking out certain features, such as eyes, noses, and lips.
- Note that: the cumulative variance of these components to see how much of the data information the projection is preserving:



The 150 components account for just over 90% of the variance. That would lead us to believe that using these 150 components, we would recover most of the essential characteristics of the data.

To make this more concrete, we can compare the input images with the images reconstructed from these 150 components:

- The top row here shows the input images, while the bottom row shows the reconstruction of the images from just 150 of the ~3,000 initial features.
- This visualization makes clear why the PCA feature selection used in SVM was so successful:
  - Although it reduces the dimensionality of the data by nearly a factor of 20, the projected images contain enough information that we might easily recognize the individuals in the image.
  - What this means is that our classification algorithm needs to be trained on 150-dimensional data rather than 3,000-dimensional data, which depending on the particular algorithm we choose, can lead to a much more efficient classification.

Summary of Principal Component Analysis

- In this section we have discussed the use of principal component analysis for dimensionality reduction, for visualization of high-dimensional data, for noise filtering, and for feature selection within high-dimensional data.
- Because of the versatility and interpretability of PCA, it has been shown to be effective in a wide variety of contexts and disciplines. Given any high-dimensional dataset, I tend to start with PCA in order to visualize the relationship between points (as we did with the digits), to understand the main variance in the data (as we did with the eigenfaces), and to understand the intrinsic dimensionality (by plotting the explained variance ratio).
- Certainly PCA is not useful for every high-dimensional dataset, but it offers a straightforward and efficient path to gaining insight into high-dimensional data.

Cont'd: weakness

- PCA's tends to be highly affected by outliers in the data.
- For this reason, many robust variants of PCA have been developed, many of which act to iteratively discard data points that are poorly described by the initial components.
- Scikit-Learn contains a couple interesting variants on PCA, including RandomizedPCA and SparsePCA, both also in the sklearn.decomposition submodule.
  - RandomizedPCA uses a non-deterministic method to quickly approximate the first few principal components in very high-dimensional data;
  - SparsePCA introduces a regularization term (see in "Linear Regression") that serves to enforce sparsity of the components.

Manifold Learning

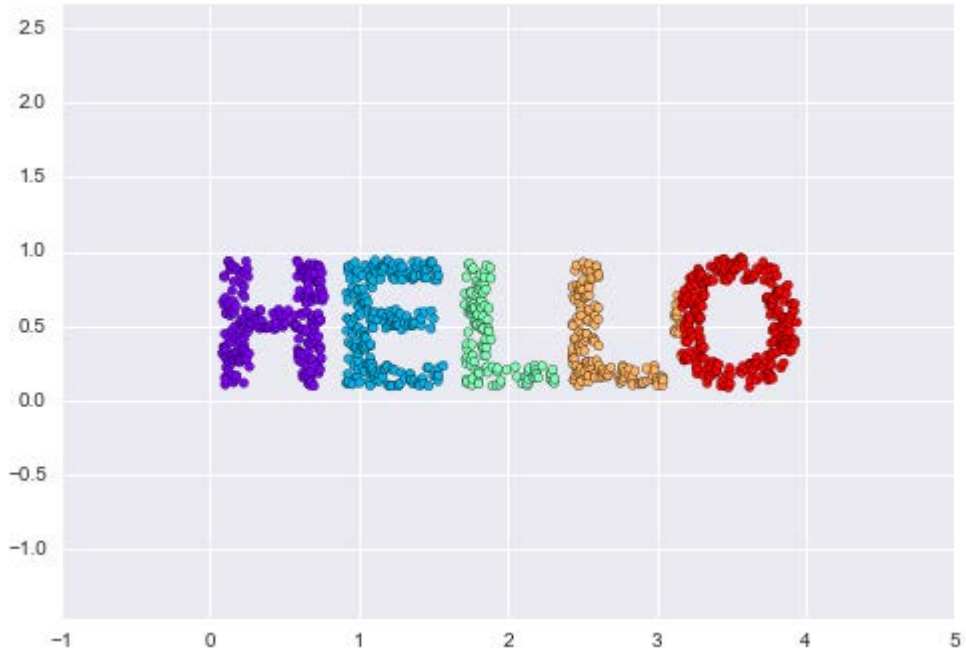
Address the nonlinear relationships within the data.

Manifold Learning

- Manifold learning is a class of unsupervised estimators that seeks to describe datasets as low-dimensional manifolds embedded in high-dimensional spaces.
- Imagining a sheet of paper: this is a two-dimensional object that lives in our three-dimensional world, and can be bent or rolled in that two dimensions.
  - Rotating, re-orienting, or stretching the piece of paper in three-dimensional space doesn't change the flat geometry of the paper: such operations are akin to linear embeddings.
  - If you bend, curl, or crumple the paper, it is still a two-dimensional manifold, but the embedding into the three-dimensional space is no longer linear.
  - Manifold learning algorithms would seek to learn about the fundamental two-dimensional nature of the paper, even as it is contorted to fill the three-dimensional space.
- Here we will demonstrate a number of manifold methods, going most deeply into a couple techniques: multidimensional scaling (MDS), locally linear embedding (LLE), and isometric mapping (IsoMap).

2.1 Manifold Learning: "HELLO"

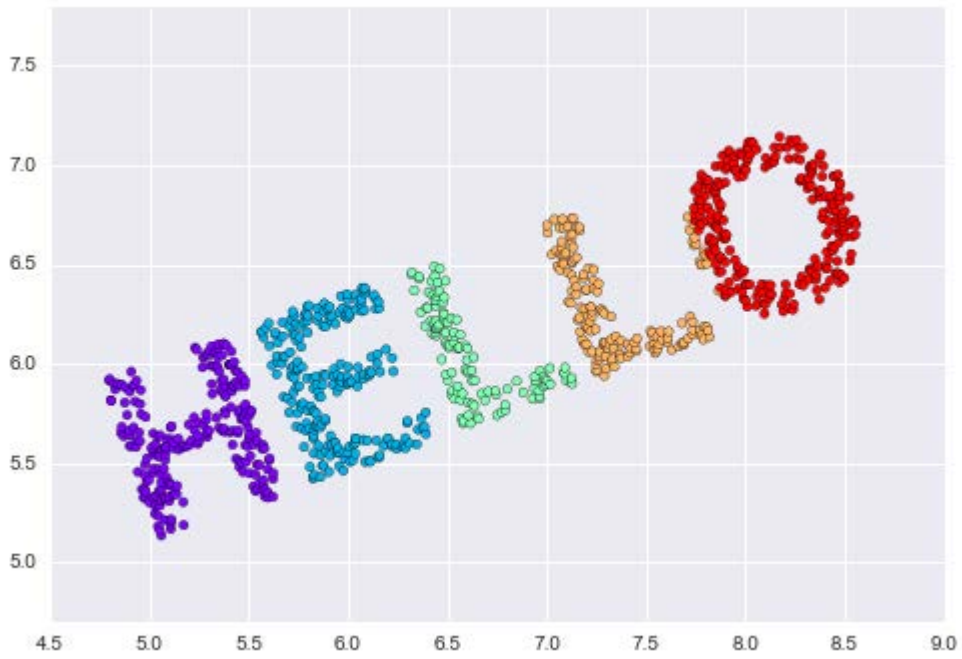
Create data in the shape of the word "HELLO“ and visualize



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.

2.2 Multidimensional Scaling (MDS)

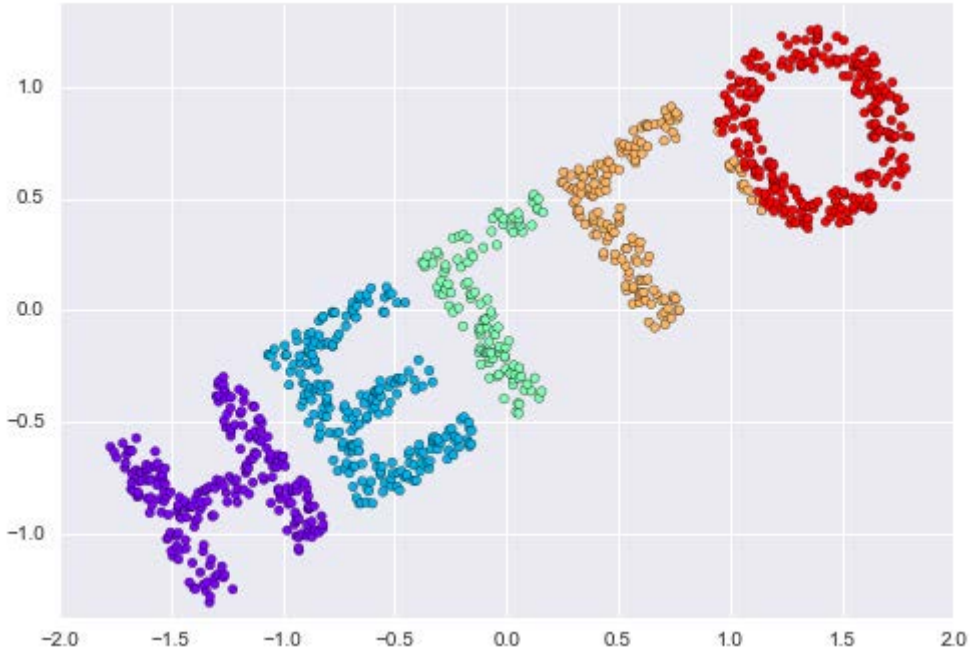
- The particular choice of x and y values of the dataset are not the most fundamental description of the data: we can scale, shrink, or rotate the data, and the "HELLO" will still be apparent.
- For example, if we use a rotation matrix to rotate the data, the x and y values change, but the data is still fundamentally the same:



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* x *N* array such that entry (*i, j*) contains the distance between point *i* and point *j*.
- Use Scikit-Learn's efficient pairwise_distances function to do this for our original data:

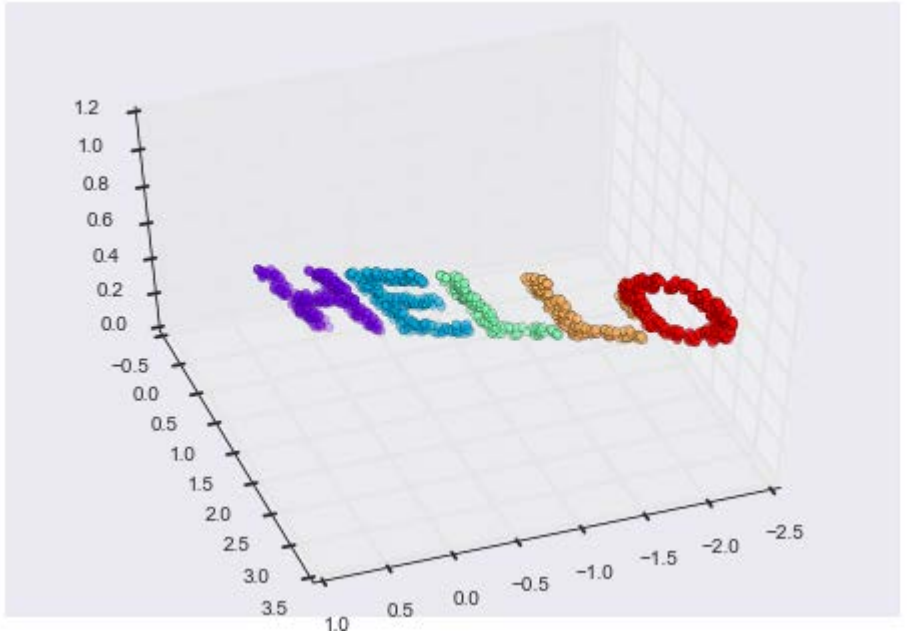
- Further, while computing this distance matrix from the (*x, y*) coordinates is straightforward, transforming the distances back into x and y coordinates is rather difficult.
- This is exactly what the multidimensional scaling algorithm aims to do: given a distance matrix between points, it recovers 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:



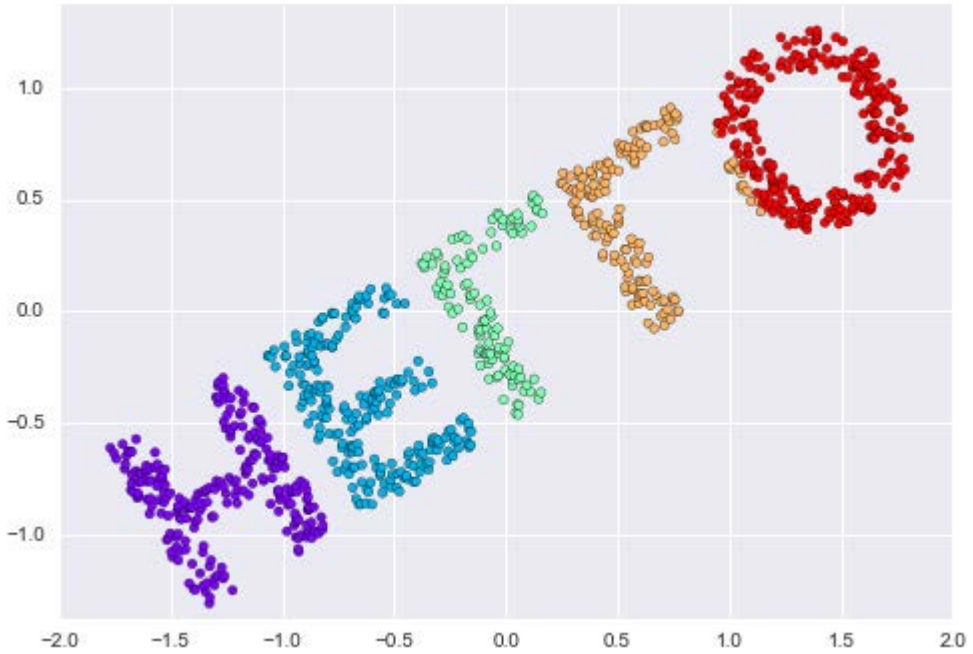
The MDS algorithm recovers one of the possible two-dimensional coordinate representations of our data, using only the *N* x *N* distance matrix describing the relationship between the data points.

2.2 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.
- For example, instead of simply rotating the data in the two-dimensional plane, we can project it into three dimensions and visualize:



- Use MDS estimator to input this three-dimensional data, compute the distance matrix, and then determine the optimal two-dimensional embedding for this distance matrix. The result recovers a representation of the original data:



This is essentially the goal of a manifold learning estimator:

- given high-dimensional embedded data, it seeks a low-dimensional representation of the data that preserves certain relationships within the data.

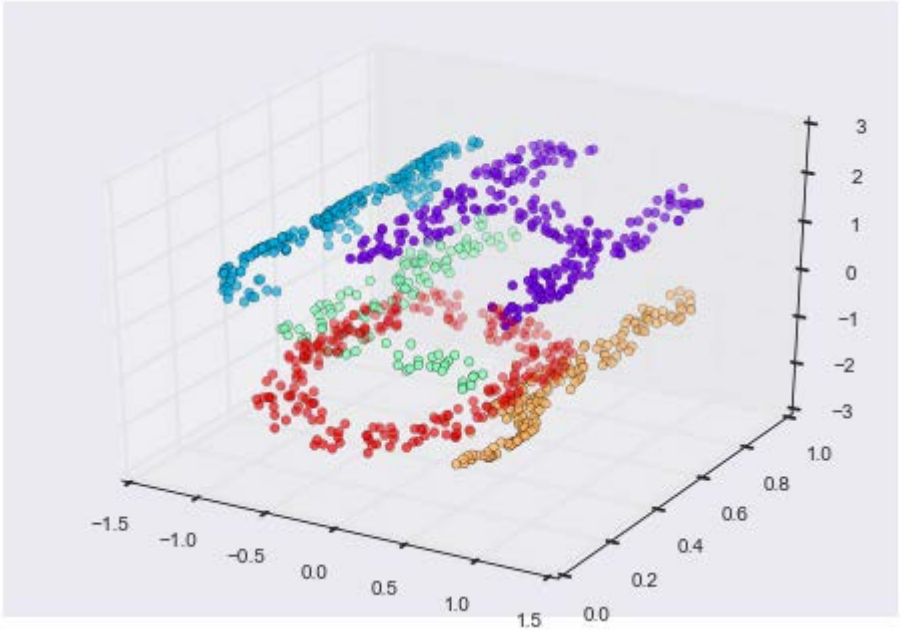
In the case of MDS, the quantity preserved is the distance between every pair of points.

So far, the discussion has considered linear embeddings:

essentially consist of rotations, translations, and scalings of data into higher-dimensional spaces.

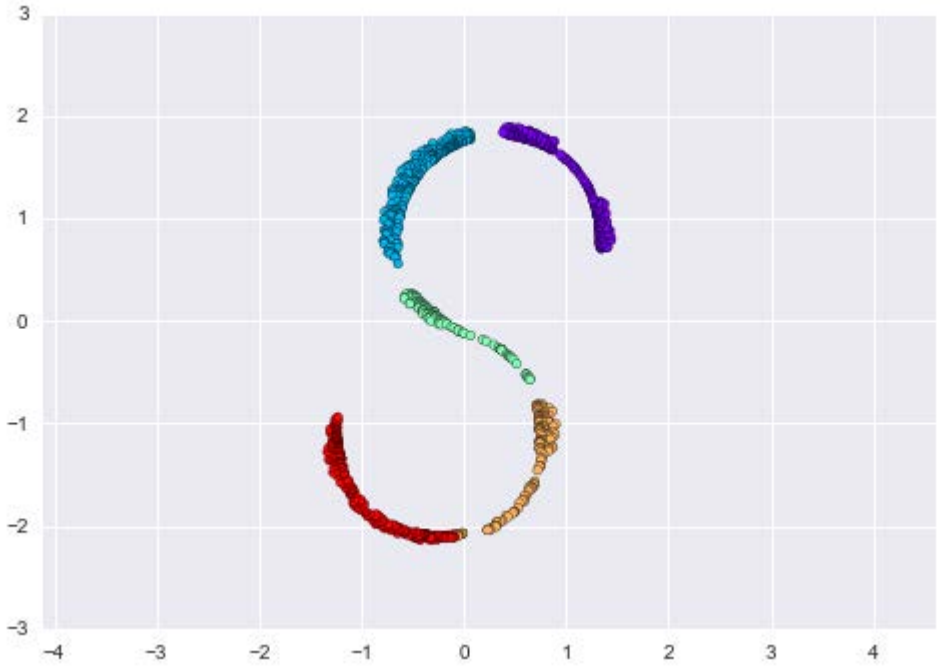
2.3 Nonlinear Embeddings: Where MDS Fails

- Where MDS breaks down is when the embedding is nonlinear.
- Consider the following embedding, which takes the input and contorts it into an "S" shape in three dimensions:



The fundamental relationships between the data points are still there, but this time the data has been transformed in a nonlinear way: it has been wrapped-up into the shape of an "S."

If we try a simple MDS algorithm on this data, it is not able to "unwrap" this nonlinear embedding, and we lose track of the fundamental relationships in the embedded manifold:

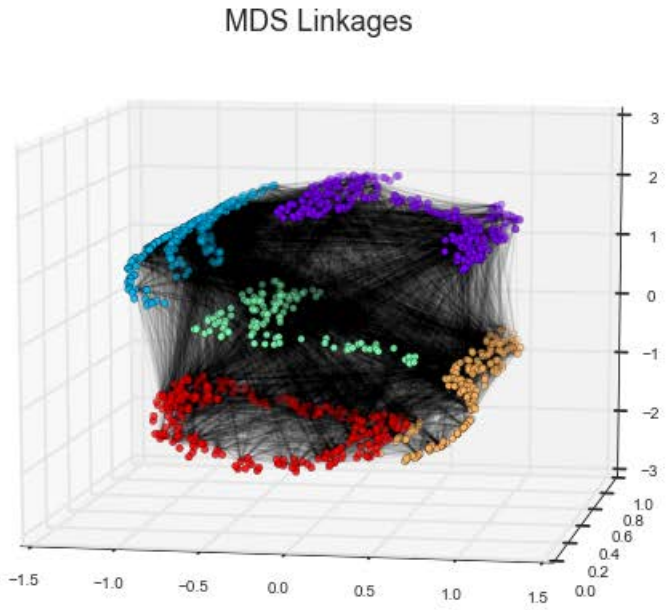


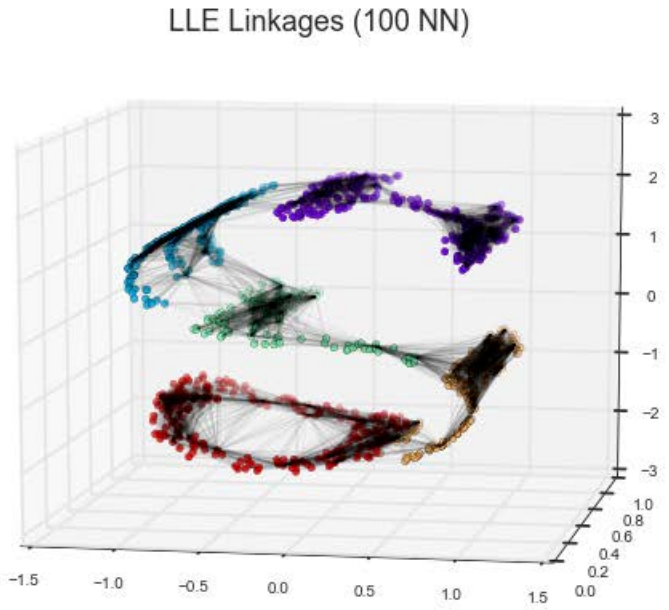
The best two-dimensional linear embedding does not unwrap the S-curve, but instead throws out the original y-axis.

The source of the problem is that MDS tries to preserve distances between faraway points when constructing the embedding.

But what if we instead modified the algorithm such that it only preserves distances between nearby points? → The resulting embedding would be closer to what we want.

2.4 Nonlinear Manifolds: Locally Linear Embedding (LLE)





Each faint line represents a distance that should be preserved in the embedding.

MDS model: it tries to preserve the distances between each pair of points in the dataset.

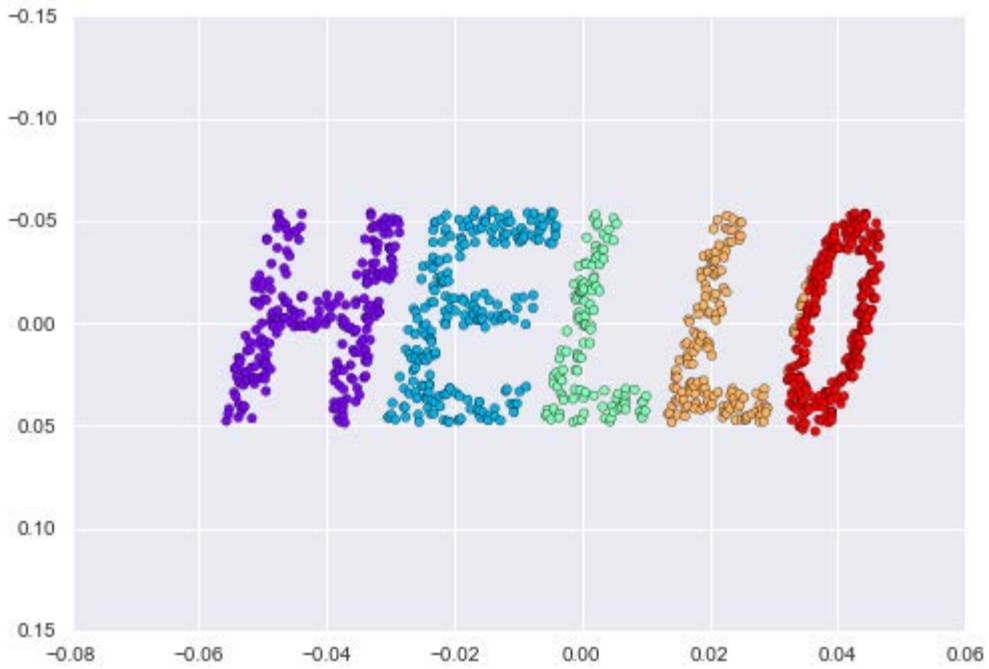
Why MDS fails: there is no way to flatten this data while adequately preserving the length of every line drawn between the two points.

LLE: rather than preserving *all* distances, it instead tries to preserve only the distances between *neighboring points*:

The data is unrolled in a way that keeps the lengths of the lines approximately the same. This is precisely what LLE does, through a global optimization of a cost function reflecting this logic.

LLE comes in a number of flavors; here we will use the modified LLE algorithm to recover the embedded two-dimensional manifold.

In general, modified LLE does better than other flavors of the algorithm at recovering well-defined manifolds with very little distortion:



The result remains somewhat distorted compared to our original manifold, but captures the essential relationships in the data!

Some Thoughts on Manifold Methods

- Though this motivation is compelling, in practice manifold learning techniques tend to be picky enough that they are rarely used for anything more than simple qualitative visualization of high-dimensional data.
- The following are some of the particular challenges of manifold learning, which all contrast poorly with PCA:
  - In manifold learning, there is no good framework for handling missing data. In contrast, there are straightforward iterative approaches for missing data in PCA.
  - In manifold learning, the presence of noise in the data can "short-circuit" the manifold and drastically change the embedding. In contrast, PCA naturally filters noise from the most important components.
  - The manifold embedding result is generally highly dependent on the number of neighbors chosen, and there is generally no solid quantitative way to choose an optimal number of neighbors. In contrast, PCA does not involve such a choice.

Cont'd

- In manifold learning, the globally optimal number of output dimensions is difficult to determine. In contrast, PCA lets you find the output dimension based on the explained variance.
- In manifold learning, the meaning of the embedded dimensions is not always clear. In PCA, the principal components have a very clear meaning.
- In manifold learning the computational expense of manifold methods scales as O[N^2] or O[N^3]. For PCA, there exist randomized approaches that are generally much faster (though see the megaman package for some more scalable implementations of manifold learning).

- With all that on the table, the only clear advantage of manifold learning methods over PCA is their ability to preserve nonlinear relationships in the data; for that reason I tend to explore data with manifold methods only after first exploring them with PCA.

Scikit-Learn implements several common variants of manifold learning beyond Isomap and LLE: the Scikit-Learn documentation has a nice discussion and comparison of them. Some recommendations below based on my experiences:

- For toy problems such as the S-curve we saw before, locally linear embedding (LLE) and its variants (especially modified LLE), perform very well. This is implemented in sklearn.manifold.LocallyLinearEmbedding.
- For high-dimensional data from real-world sources, LLE often produces poor results, and isometric mapping (IsoMap) seems to generally lead to more meaningful embeddings. This is implemented in sklearn.manifold.Isomap
- For data that is highly clustered, t-distributed stochastic neighbor embedding (t-SNE) seems to work very well, though can be very slow compared to other methods. This is implemented in sklearn.manifold.TSNE.

If you're interested in getting a feel for how these work, I'd suggest running each of the methods on the data in this section.

The End!