Visualising high-dimensional datasets using PCA and t-SNE in Python

Luuk Derksen Follow
Oct 29, 2016 · 9 min read

The first step around any data related challenge is to start by exploring the data itself. This could be by looking at, for example, the distributions of certain variables or looking at potential correlations between variables.

The problem nowadays is that most datasets have a large number of variables. In other words, they have a high number of dimensions along which the data is distributed. Visually exploring the data can then become challenging and most of the time even practically impossible to do manually. However, such visual exploration is incredibly important in any data-related problem. Therefore it is key to understand how to visualise high-dimensional datasets. This can be achieved using techniques known as dimensionality reduction. This post will focus on two techniques that will allow us to do this: PCA and t-SNE.

More about that later. Lets first get some (high-dimensional) data to work with.

MNIST dataset

We will use the MNIST-dataset in this write-up. There is no need to download the dataset manually as we can grab it through using Scikit Learn.

```
import numpy as np
from sklearn.datasets import fetch_mldata

mnist = fetch_mldata("MNIST original")
X = mnist.data / 255.0
y = mnist.target

print X.shape, y.shape

[out] (70000, 784) (70000,)
```

We are going to convert the matrix and vector to a Pandas DataFrame. This is very similar to the DataFrames used in R and will make it easier for us to plot it later on.

```
import pandas as pd

feat_cols = [ 'pixel'+str(i) for i in range(X.shape[1]) ]

df = pd.DataFrame(X,columns=feat_cols)
df['label'] = y
df['label'] = df['label'].apply(lambda i: str(i))

X, y = None, None

print 'Size of the dataframe: {}'.format(df.shape)

[out] Size of the dataframe: (70000, 785)
```

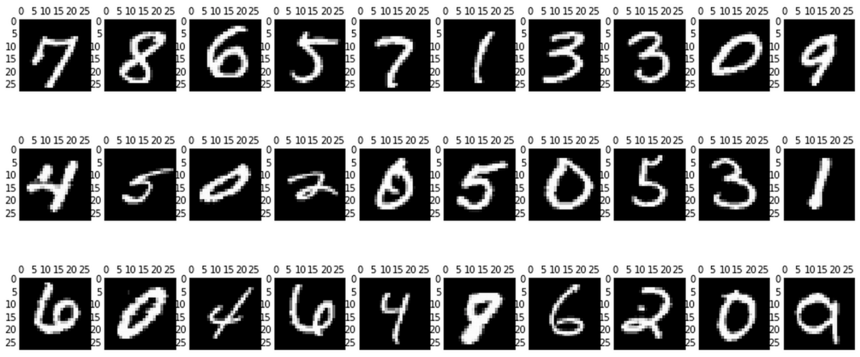
Because we dont want to be using 70,000 digits in some calculations we'll take a random subset of the digits. The randomisation is important as the dataset is sorted by its label (i.e., the first seven thousand or so are zeros, etc.). To ensure randomisation we'll create a random permutation of the number 0 to 69,999 which allows us later to select the first five or ten thousand for our calculations and visualisations.

```
rndperm = np.random.permutation(df.shape[0])
```

We now have our dataframe and our randomisation vector. Lets first check what these numbers actually look like. To do this we'll generate 30 plots of randomly selected images.

```
%matplotlib inline
import matplotlib.pyplot as plt

# Plot the graph
plt.gray()
```

```
fig = plt.figure( figsize=(16,7) )
for i in range(0,30):
    ax = fig.add_subplot(3,10,i+1, title='Digit: ' +
str(df.loc[rndperm[i],'label']) )

ax.matshow(df.loc[rndperm[i],feat_cols].values.reshape((28,2
8)).astype(float))

plt.show()
```



Now we can start thinking about how we can actually distinguish the zeros from the ones and two's and so on. If you were, for example, a post office such an algorithm could help you read and sort the handwritten envelopes using a machine instead of having humans do that. Obviously nowadays we have very advanced methods to do this, but this dataset still provides a very good testing ground for seeing how specific methods for dimensionality reduction work and how well they work.

The images are all essentially 28-by-28 pixel images and therefore have a total of 784 'dimensions', each holding the value of one specific pixel.

What we can do is reduce the number of dimensions drastically whilst trying to retain as much of the 'variation' in the information as possible. This is where we get to dimensionality reduction. Lets first take a look at something known as **Principal Component Analysis**.

Dimensionality reduction using PCA

PCA is a technique for reducing the number of dimensions in a dataset whilst retaining most information. It is using the correlation between some dimensions and tries to provide a minimum number of variables that keeps the maximum amount of variation or information about how the original data is distributed. It does not do this using guesswork but using hard mathematics and it uses something known as the eigenvalues and eigenvectors of the data-matrix. These eigenvectors of the covariance matrix have the property that they point along the major directions of variation in the data. These are the directions of maximum variation in a dataset.

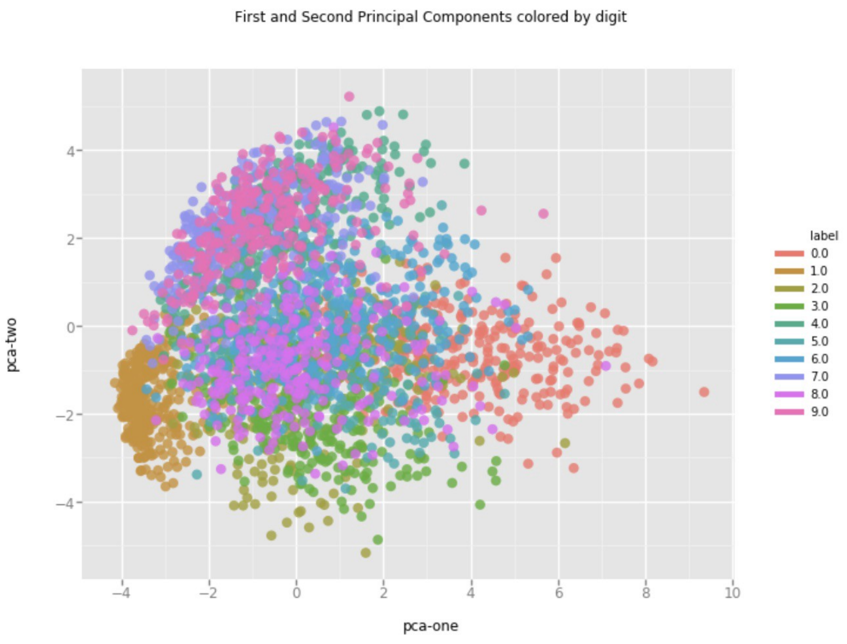
I am not going to get into the actual derivation and calculation of the principal components—if you want to get into the mathematics see this great page—instead we'll use the Scikit-Learn implementation of PCA.

Since we as humans like our two- and three-dimensional plots lets start with that and generate, from the original 784 dimensions, the first three principal components. And we'll also see how much of the variation in the total dataset they actually account for.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=3)
pca_result = pca.fit_transform(df[feat_cols].values)

df['pca-one'] = pca_result[:,0]
df['pca-two'] = pca_result[:,1]
df['pca-three'] = pca_result[:,2]

print 'Explained variation per principal component:
{}'.format(pca.explained_variance_ratio_)

[out] Explained variation per principal component: [
0.16756229  0.0826886   0.05374424]
```

Now, given that the first two components account for about 25% of the variation in the entire dataset lets see if that is enough to visually set the different digits apart. What we can do is create a scatterplot of the first and second principal component and color each of the different types of digits with a different color. If we are lucky the same type of digits will be positioned (i.e., clustered) together in groups, which would mean that the first two principal components actually tell us a great deal about the specific types of digits.

```
from ggplot import *

chart = ggplot( df.loc[rndperm[:3000],:], aes(x='pca-one',
y='pca-two', color='label') ) \
        + geom_point(size=75,alpha=0.8) \
        + ggtitle("First and Second Principal Components
colored by digit")
chart
```



From the graph we can see the two components definitely hold some information, especially for specific digits, but clearly not enough to set all of them apart. Luckily there is another technique that we can use to reduce the number of dimensions that may prove more helpful. In the next few paragraphs we are going to take a look at that technique and explore if it gives us a better way of reducing the dimensions for visualisation. The method we will be exploring is known as t-SNE (t-Distributed Stochastic Neighbouring Entities).

T-Distributed Stochastic Neighbouring Entities (t-SNE)

t-Distributed Stochastic Neighbor Embedding (t-SNE) is another technique for dimensionality reduction and is particularly well suited for the visualization of high-dimensional datasets. Contrary to PCA it is not a mathematical technique but a probablistic one. The original paper describes the working of t-SNE as:

“t-Distributed stochastic neighbor embedding (t-SNE) minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures

pairwise similarities of the corresponding low-dimensional points in the embedding”.

Essentially what this means is that it looks at the original data that is entered into the algorithm and looks at how to best represent this data using less dimensions by matching both distributions. The way it does this is computationally quite heavy and therefore there are some (serious) limitations to the use of this technique. For example one of the recommendations is that, in case of very high dimensional data, you may need to apply another dimensionality reduction technique before using t-SNE:

```
|  It is highly recommended to use another dimensionality
reduction
|  method (e.g. PCA for dense data or TruncatedSVD for
sparse data)
|  to reduce the number of dimensions to a reasonable amount
(e.g. 50)
|  if the number of features is very high.
```

The other key drawback is that it:

“Since t-SNE scales quadratically in the number of objects N, its applicability is limited to data sets with only a few thousand input objects; beyond that, learning becomes too slow to be practical (and the memory requirements become too large)”.

We will use the Scikit-Learn Implementation of the algorithm in the remainder of this writeup.

Contrary to the recommendation above we will first try to run the algorithm on the actual dimensions of the data (784) and see how it does. To make sure we don’t burden our machine in terms of memory and power/time we will only use the first 7,000 samples to run the algorithm on.

```
import time

from sklearn.manifold import TSNE

n_sne = 7000

time_start = time.time()
tsne = TSNE(n_components=2, verbose=1, perplexity=40,
n_iter=300)
```

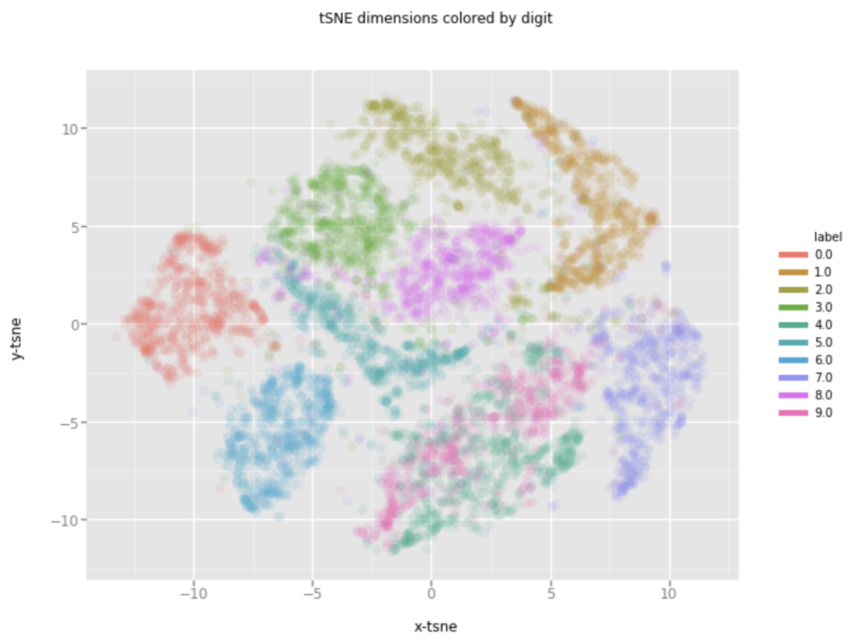
```
tsne_results =
tsne.fit_transform(df.loc[rndperm[:n_sne],feat_cols].values)
```

```
print 't-SNE done! Time elapsed: {}
seconds'.format(time.time()-time_start)
```

```
[out]
[t-SNE] Computing pairwise distances...
[t-SNE] Computed conditional probabilities for sample 1000 /
7000
[t-SNE] Computed conditional probabilities for sample 2000 /
7000
[t-SNE] Computed conditional probabilities for sample 3000 /
7000
[t-SNE] Computed conditional probabilities for sample 4000 /
7000
[t-SNE] Computed conditional probabilities for sample 5000 /
7000
[t-SNE] Computed conditional probabilities for sample 6000 /
7000
[t-SNE] Computed conditional probabilities for sample 7000 /
7000
[t-SNE] Mean sigma: 2.170716
[t-SNE] Error after 97 iterations with early exaggeration:
17.891132
[t-SNE] Error after 300 iterations: 2.206017
t-SNE done! Time elapsed: 813.213096142 seconds
```

Now that we have the two resulting dimensions we can again visualise them by creating a scatter plot of the two dimensions and coloring each sample by its respective label.

```
df_tsne = df.loc[rndperm[:n_sne],:].copy()
df_tsne['x-tsne'] = tsne_results[:,0]
df_tsne['y-tsne'] = tsne_results[:,1]

chart = ggplot( df_tsne, aes(x='x-tsne', y='y-tsne',
color='label') ) \
        + geom_point(size=70,alpha=0.1) \
        + ggtitle("tSNE dimensions colored by digit")
chart
```



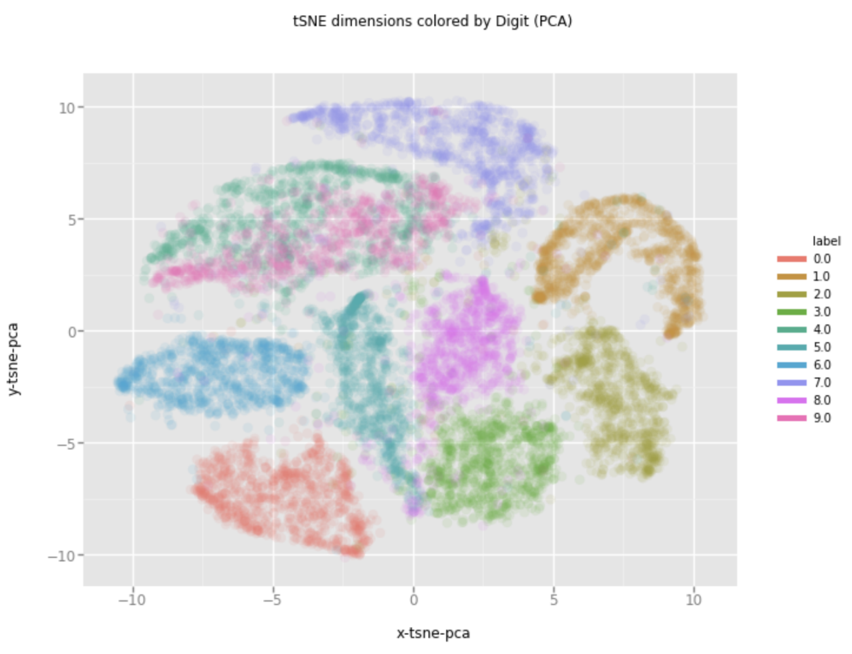
This is already a significant improvement over the PCA visualisation we used earlier. We can see that the digits are very clearly clustered in their own little group. If we would now use a clustering algorithm to pick out the seperate clusters we could probably quite accurately assign new points to a label.

We'll now take the recommandations to heart and actually reduce the number of dimensions before feeding the data into the t-SNE algorithm. For this we'll use PCA again. We will first create a new dataset containing the fifty dimensions generated by the PCA reduction algorithm. We can then use this dataset to perform the t-SNE on

```
pca_50 = PCA(n_components=50)
pca_result_50 = pca_50.fit_transform(df[feat_cols].values)

print 'Cumulative explained variation for 50 principal
components:
{}'.format(np.sum(pca_50.explained_variance_ratio_))

[out] Cumulative explained variation for 50 principal
components: 84.6676222833%
```

Amazingly, the first 50 components roughly hold around 85% of the total variation in the data.

Now lets try and feed this data into the t-SNE algorithm. This time we'll use 10,000 samples out of the 70,000 to make sure the algorithm does not take up too much memory and CPU. Since the code used for this is very similar to the previous t-SNE code I have moved it to the Appendix: Code section at the bottom of this post. The plot it produced is the following one:



From this plot we can clearly see how all the samples are nicely spaced apart and grouped together with their respective digits. This could be an amazing starting point to then use a clustering algorithm and try to identify the clusters or to actually use these two dimensions as input to another algorithm (e.g., something like a Neural Network).

So we have explored using various dimensionality reduction techniques to visualise high-dimensional data using a two-dimensional scatter plot. We have not gone into the actual mathematics involved but instead relied on the Scikit-Learn implementations of all algorithms.

Roundup Report

Before closing off with the appendix…

Together with some likeminded friends we are sending out weekly newsletters with some links and notes that we want to share amongst ourselves (why not allow others to read them as well?).

Appendix: Code

Code: t-SNE on PCA-reduced data

```
n_sne = 10000

time_start = time.time()

tsne = TSNE(n_components=2, verbose=1, perplexity=40,
n_iter=300)
tsne_pca_results =
tsne.fit_transform(pca_result_50[rndperm[:n_sne]])
```

```
print 't-SNE done! Time elapsed: {}
seconds'.format(time.time()-time_start)
```

```
[out]
[t-SNE] Computing pairwise distances...
[t-SNE] Computed conditional probabilities for sample 1000 /
10000
[...]
[t-SNE] Computed conditional probabilities for sample 10000
/ 10000
[t-SNE] Mean sigma: 1.814452
[t-SNE] Error after 100 iterations with early exaggeration:
18.725542
[t-SNE] Error after 300 iterations: 2.657761
t-SNE done! Time elapsed: 1620.80310392 seconds
```

And for the visualisation

```
df_tsne = None
df_tsne = df.loc[rndperm[:n_sne],:].copy()
df_tsne['x-tsne-pca'] = tsne_pca_results[:,0]
df_tsne['y-tsne-pca'] = tsne_pca_results[:,1]

chart = ggplot( df_tsne, aes(x='x-tsne-pca', y='y-tsne-pca',
color='label') ) \
        + geom_point(size=70,alpha=0.1) \
        + ggtitle("tSNE dimensions colored by Digit (PCA)")
chart
```