Dimensionality Reduction on MNIST dataset

This notebook discusses the importance of dimensionality reduction as a preprocessing step. We will show that high dimensional datasets can sometimes be expressed using only a few dimensions and that reducing dimensionality can make our datasets easier to work with and decrease the training time of our models.

What is dimensionalty and why is it important?

In simplistic terms, it is just the number of columns in the dataset, but it has significant downstream effects on the eventual models. The concept of the "curse of dimensionality" indicates that in high-dimensional spaces the proximity between objects have diminished differentiation effects. Even in relatively low dimensional problems, a dataset with more dimensions requires more parameters for the model to understand, and that means more rows to reliably learn those parameters. If the number of rows in the dataset is fixed, addition of extra dimensions without adding more information for the models to learn from can have a detrimental effect on the eventual model accuracy.

Dataset used for this activity:

The MNIST dataset is composed of 28x28 pixel images of handwriten digits from zero through nine.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training data set, (train.csv), has 785 columns. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image.

References:

https://www.kaggle.com/c/digit-recognizer/data (https://www.kaggle.com/c/digit-recognizer/data)
http://www.eggie5.com/69-dimensionality-reduction-using-pca (http://www.eggie5.com/69-dimensionality-reduction-using-pca)

In [1]:
```
import pandas as pd
import numpy as np
import time

from sklearn.model_selection import train_test_split

train = pd.read_csv('data/mnist_train.csv')

# Separate labels from the data
y = train['label']
# Drop the label feature
X = train.drop("label",axis=1)

# Split the train data into X_train and y_train datasets in 80:20 ratio.
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

print("Train data shape : " + str(X_train.shape))
print("Test data shape : " + str(X_test.shape))
X_train.head()
```

```
Train data shape : (33600, 784)
Test data shape : (8400, 784)
```

Out[1]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pi
34941	0	0	0	0	0	0	0	0	0	0	...	0	
24433	0	0	0	0	0	0	0	0	0	0	...	0	
24432	0	0	0	0	0	0	0	0	0	0	...	0	
8832	0	0	0	0	0	0	0	0	0	0	...	0	
30291	0	0	0	0	0	0	0	0	0	0	...	0	

5 rows × 784 columns

Applying KNN classifier on MNIST dataset without applying Dimensionality Reduction

Let's take a look at how long it takes to train a KNN Classifier on the MNIST dataset:

In [2]:
```
from sklearn.neighbors import KNeighborsClassifier

start = time.time()
clf = KNeighborsClassifier(n_neighbors=3, algorithm='ball_tree')
clf.fit(X_train, y_train)
y = clf.predict(X_test)

# Calculate error in prediction
errors = (y_test != y).sum()
total = X_test.shape[0]
error_rate_without_dr = (errors/float(total)) * 100
print("Error rate without dimensionality reduction: %d/%d * 100 = %f" % (errors,

end = time.time()
duration_without_dr = end-start
print("Time taken to train a KNN Classifier without DR: %d seconds" %duration_wit
```

```
Error rate without dimensionality reduction: 280/8400 * 100 = 3.333333
Time taken to train a KNN Classifier without DR: 711 seconds
```

Applying SVD transform

Singular Value Decomposition (SVD) is a matrix factorization technique that factors a matrix M into the three matrices U, Σ, and V. This is very similar to PCA, except that the factorization for SVD is done on the data matrix, whereas for PCA, the factorization is done on the covariance matrix. Typically, SVD is used under the hood to find the principle components of a matrix.

In [3]:
```
from sklearn.decomposition import TruncatedSVD

start = time.time()
svd = TruncatedSVD(n_components=150)
svd.fit(X_train)

X_train_svd = svd.transform(X_train)
X_test_svd = svd.transform(X_test)
print("SVD transformation time: %d seconds" % (time.time()-start))

start = time.time()
clf.fit(X_train_svd, y_train)
y = clf.predict(X_test_svd)

errors = (y_test != y).sum()
total = X_test_svd.shape[0]
error_rate_with_svd = (errors/float(total)) * 100
print("Error rate with SVD: %d/%d * 100 = %f" % (errors, total, error_rate_with_s

end = time.time()
duration_with_svd = end-start
print("Time taken to train a KNN Classifier with SVD: %d seconds" %duration_with_
```

```
SVD transformation time: 10 seconds
Error rate with SVD: 247/8400 * 100 = 2.940476
Time taken to train a KNN Classifier with SVD: 157 seconds
```

In [ ]:

```
```