School of Engineering and Applied Science (SEAS), Ahmedabad University

B.Tech (CSE Semester VI)/M.Tech/PhD:
Machine Learning (CSE 523)

Project Submission #3: Principal Component Analysis

- **Group No.:** S_ECC4
- **Project Area:** Climate and Environment
- **Project Title:** Landslide Susceptibility Assessment by Novel Hybrid Machine Learning Algorithms
- **Name of the group members :**

1. Jainam Chhatbar (1741002)
2. Dhairya Dudhatra (1741058)
3. Charmil Gandhi (1741059)
4. Jinesh Patel (1741076)

Principal Component Analysis For Dimension Reduction

Case 1: Considering that dimensions of data set is less than number of samples

Landslide prediction and susceptibility assessment depends on variety of factors such as:

- Population, Death and Injuries Count
- Slope Aspect and Location Accuracy
- Lithology and Landslide Type
- Distance To Faults
- Drainage Density
- Distance To Geological Boundaries
- Latitude and Longitude
- Various Triggers like rainfall, snow, mining etc.

As landslide depends on so many factors, it is obvious that dimensions of data set is smaller than the number of samples. We have considered **Global Landslide Catalog (GLC)** here. In GLC , we first mapped all the text fields to integer values. We have considered 2000 samples and 11 features. So dimension of our matrix **X** (which is data set) is $[2000 \times 11]$. As all our features are not in the same range, normalization is required to change the values of numeric columns in data set to common scale and then **Covariance Matrix (S)** was formed :

$$S = \frac{1}{N} \sum (X^T \times X)$$

The dimensions of **Covariance Matrix (S)** is $[11 \times 11]$. The next step is to **Eigen Values** and **Eigen Vectors** related to number of features i.e 11. Next we formed the **Projection Matrix** given by $(B^T.B)$. Matrix **B** is formed by the eigen vectors of all components. Dimensions of **B** is $[11 \times 11]$.

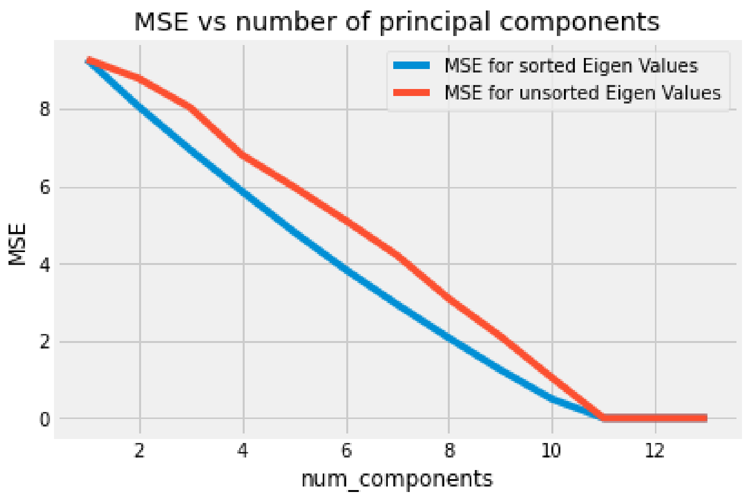


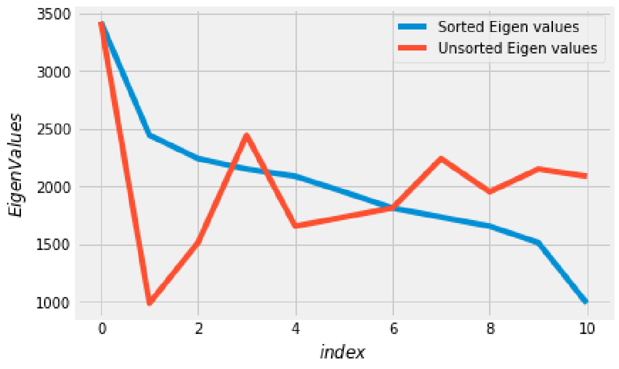
Figure 1: MSE vs Number Of Components

As seen in *Figure 1*, ***MSE*** decreases as the number of components increases. Thus, as the dimensions increases, we are covering maximum spread and the error i.e loss in information decreases. Also, significant change is observed in both the case :

- Sorted Eigen Values
- Unsorted Eigen Values

Though, the trend remains the same : MSE decreases with increase in number of components. Also we can observe from *Figure 3*, that variance increases as number of principal components increases. As shown in figure, we have considered 11 components, so variance increases till 11 and then becomes constant.

Also as seen in **Figure 2**, As the both: sorted and unsorted eigen values are displayed. We are observing such graphs because in unsorted eigen values, for same index, it's value may differ from the value in unsorted eigen value.



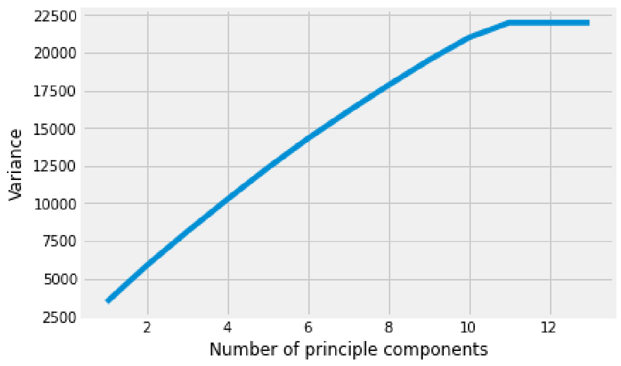


Figure 2: Eigen Values vs Index

Figure 3: Variance vs Number Of Principal Components

Case 2: Considering that dimensions of data set is larger than number of samples In this case, we have assumed that dimensions of data set i.e features are more than the number of samples. As mentioned in **Case 1**, we have 11 features. So, for **Case 2**, we have just considered 7 samples of data. So for this case, dimensions of **X** is $[7 \times 11]$, though, the performance will not be great. The **Covariance Matrix (S)** is given by:

$$S = \frac{1}{N} \sum (X \times X^T)$$

Thus, dimensions of **Covariance Matrix (S)** is $[7 \times 7]$. The next step is to **Eigen Values** and **Eigen Vectors** related to number of features i.e 11. Next we formed the **Projection Matrix** given by $(B^T.B)$. Matrix **B** is formed by the eigen vectors of all components. Dimensions of **B** is $[11 \times 11]$.

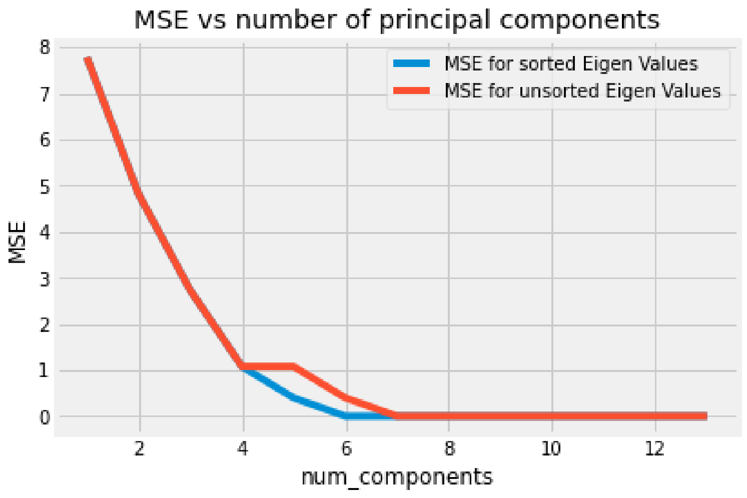
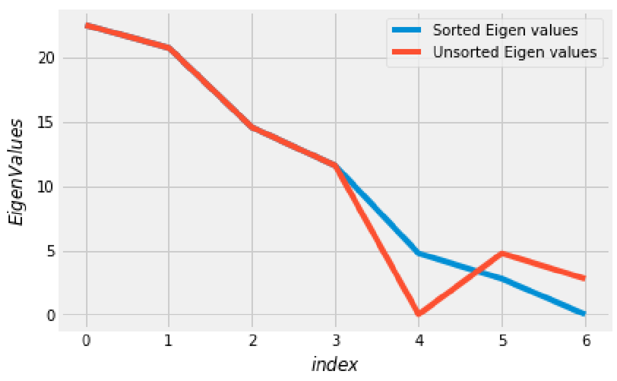


Figure 4: MSE vs Number Of Components

The trend followed here is very similar to the one in **Case 1**. MSE decreases as the number of component increases which signifies that the loss in information also decreases. From **Figure 6**, it can be noted that variance increases with the increase in number of principal components which is the similar trend followed in **Case 1**.



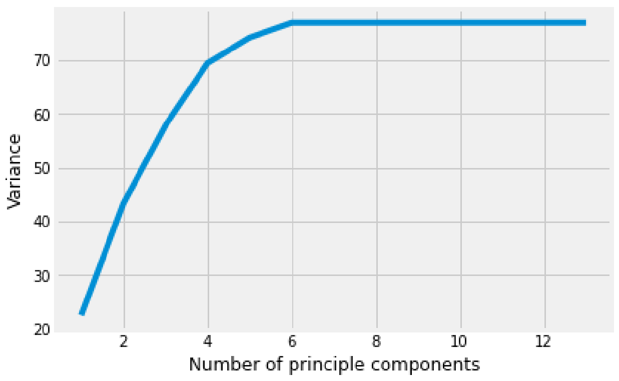


Figure 5: Eigen Values vs Index

Figure 6: Variance vs Number Of Principal Components

Implementation code:

```
import pandas as pd
import numpy as np
import timeit
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from ipywidgets import interact

%matplotlib inline


def normalize(X):
    mu = np.mean(X, axis=0)       # Calculating the mean of X
    std = np.std(X, axis=0)       # Calculating the standard deviation of X
    std_filled = std.copy()
    std_filled[std==0] = 1.
    # Calculating the normalized data Xbar
    Xbar = (X - mu)/std_filled
    return Xbar, mu, std


df2 = pd.read_csv("/content/GLC03122015_latest.csv", header='infer')
X = df2.values
print(X.shape)              # Display dimension of X
Xbar, mu, std = normalize(X)
print(Xbar.shape)           # Display dimension of normalized X
print(mu.shape)
print(std.shape)
print(np.trace(Xbar))
```

```
def eig_sorted(S):
    # Finding the eigenvalues and eigenvectors
    eigvals, eigvecs = np.linalg.eig(S)
    # Sorting the eigenvalues and corresponding eigenvectors
    idx = eigvals.argsort()[::-1]
    eigvals = eigvals[idx]
    eigvecs = eigvecs[:,idx]
    return (eigvals, eigvecs)


def eig_unsorted(S):
    # Finding the eigenvalues and eigenvectors
    eigvals, eigvecs = np.linalg.eig(S)
    return (eigvals,eigvecs)


def projection_matrix(B):
    P = np.matmul(B,B.T)      # Computing the projection matrix
    return P


def PCA_sorted(X, num_components):
    # Computing PCA on given input matrix for sorted eigenvalues
    Xbar,mu,std=normalize(X)
    covariance=np.matmul(Xbar.T,Xbar)
    S=covariance
    eigvals,eigvecs=eig_sorted(S)

    # To find the variance of eigenvalues
    eigenSum = eigvals[:num_components].sum()

    B = np.stack(eigvecs[:,:num_components])
    P = projection_matrix(B)      # Compute projection matrix
    X_reconstruct=np.matmul(P,X.T)      # Reconstuct the X
    X_reconstruct=X_reconstruct.T
    return eigenSum, X_reconstruct


def PCA_unsorted(X, num_components):
    # Computing PCA on given input matrix for unsorted eigenvalues
    Xbar,mu,std=normalize(X)
    covariance=np.matmul(Xbar.T,Xbar)
    S=covariance
    eigvals,eigvecs=eig_unsorted(S)
    B = np.stack(eigvecs[:,:num_components])
    P = projection_matrix(B)      # Compute projection matrix
    X_reconstruct=np.matmul(P,X.T)      # Reconstruct the X
    X_reconstruct=X_reconstruct.T
    return X_reconstruct
```

```
covariance = np.matmul(Xbar.T,Xbar)
print(covariance.shape)
print(np.trace(covariance))
S = covariance


# Plotting sorted and unsorted eigenvalues against index
sorted_eigvals, sorted_eigvecs = eig_sorted(S)
unsorted_eigvals, unsorted_eigvecs = eig_unsorted(S)
plt.plot(sorted_eigvals, label = 'Sorted Eigen values')
plt.plot(unsorted_eigvals, label = 'Unsorted Eigen values')
plt.xlabel("$index$")
plt.ylabel("$EigenValues$")
plt.legend()


def mse(predict, actual):
    # Computing MSE
    return np.square(predict - actual).sum(axis=1).mean()


loss_1 = []
variance = []
reconstructions_1 = []
# iterating over different numbers of principal components, and
# finding the MSE for sorted eigenvalues
for num_component in range(1, 14):
    eigenSum, reconst_1 = PCA_sorted(Xbar, num_component)
    error_1 = mse(reconst_1, Xbar)      # Computing MSE
    reconstructions_1.append(reconst_1)
    variance.append((num_component, eigenSum))
    loss_1.append((num_component, error_1))

reconstructions_1 = np.asarray(reconstructions_1)

# "unnormalize" the reconstructed data
reconstructions_1 = reconstructions_1 * std + mu

loss_1 = np.asarray(loss_1)
variance = np.asarray(variance)


loss_2 = []
reconstructions_2 = []
# iterate over different numbers of principal components, and
# finding the MSE for unsorted eigenvalues
for num_component in range(1, 14):
    reconst_2 = PCA_unsorted(Xbar, num_component)
    error_2 = mse(reconst_2, Xbar)    # Computing MSE
    reconstructions_2.append(reconst_2)
    loss_2.append((num_component, error_2))

reconstructions_2 = np.asarray(reconstructions_2)

# "unnormalize" the reconstructed image
reconstructions_2 = reconstructions_2 * std + mu

loss_2 = np.asarray(loss_2)


# Plotting the MSE of sorted and unsorted eigenvalues against
# number of principal components
fig, ax = plt.subplots()
ax.plot(loss_1[:,0], loss_1[:,1], label = 'MSE for sorted Eigen Values');
ax.plot(loss_2[:,0], loss_2[:,1], label = 'MSE for unsorted Eigen Values');
ax.set(xlabel='num_components', ylabel='MSE', title=
          'MSE vs number of principal components');
plt.legend()


# Plotting graph of variance against number of principal components
plt.plot(variance[:,0], variance[:,1])
plt.xlabel("Number of principal components")
plt.ylabel("Variance");

def correlation_matrix(df):
    from matplotlib import pyplot as plt
    from matplotlib import cm as cm

    fig = plt.figure(figsize=(16,12))
    ax1 = fig.add_subplot(111)
    cmap = cm.get_cmap('jet', 30)
    cax = ax1.imshow(df2.corr(), interpolation="nearest", cmap=cmap)
    ax1.grid(True)
    plt.title('Features correlation\n',fontsize=15)
    labels=df.columns
    ax1.set_xticklabels(labels,fontsize=9)
    ax1.set_yticklabels(labels,fontsize=9)
    # Add colorbar, make sure to specify tick locations to match desired tickl
    fig.colorbar(cax, ticks=[0.1*i for i in range(-11,11)])
    plt.show()

correlation_matrix(df2)

def time(f, repeat=10):
    times = []
```

```
        for _ in range(repeat):
            start = timeit.default_timer()
            f()
            stop = timeit.default_timer()
            times.append(stop-start)
        return np.mean(times), np.std(times)

    times_mm0 = []
    times_mm1 = []

    # iterate over datasets of different size
    for datasetsize in np.arange(4, 784, step=20):
        XX = Xbar[:datasetsize] # select the first `datasetsize` samples in the da
        # record the running time for computing X.T @ X
        mu, sigma = time(lambda : XX.T @ XX)
        times_mm0.append((datasetsize, mu, sigma))

        # record the running time for computing X @ X.T
        mu, sigma = time(lambda : XX @ XX.T)
        times_mm1.append((datasetsize, mu, sigma))

    times_mm0 = np.asarray(times_mm0)
    times_mm1 = np.asarray(times_mm1)

    fig, ax = plt.subplots()
    ax.set(xlabel='size of dataset', ylabel='running time')
    bar1 = ax.errorbar(times_mm0[:, 0], times_mm0[:, 1], times_mm0[:, 2], label="$
    bar2 = ax.errorbar(times_mm1[:, 0], times_mm1[:, 1], times_mm1[:, 2], label="$
    plt.ylim(-0.00001,0.0003)
    plt.xlim(0,300)
    ax.legend();



    # For Case 2 :

    import pandas as pd
    import numpy as np
    import timeit
    import matplotlib as mpl
    mpl.use('Agg')
    import matplotlib.pyplot as plt
    plt.style.use('fivethirtyeight')
    from ipywidgets import interact

    %matplotlib inline


    def normalize(X):
```

```
    mu = np.mean(X, axis=0)          # Compute the mean of X
    std = np.std(X, axis=0)          # Compute the std of X
    std_filled = std.copy()
    std_filled[std==0] = 1.
    Xbar = (X - mu)/std_filled       # Compute the normalized data Xbar
    return Xbar, mu, std


df2 = pd.read_csv("/content/GLC03122015_latest.csv", header='infer')

x = df2.iloc[:7,:]
X = x.values
print(X.shape)
Xbar, mu, std = normalize(X)
print(Xbar.shape)
print(mu.shape)
print(std.shape)
print(np.trace(Xbar))


def eig_sorted(S):
    eigvals, eigvecs = np.linalg.eig(S)
    idx = eigvals.argsort()[::-1]
    eigvals = eigvals[idx]
    eigvecs = eigvecs[:,idx]
    return (eigvals, eigvecs)


def eig_unsorted(S):
    eigvals, eigvecs = np.linalg.eig(S)
    return (eigvals,eigvecs)


def projection_matrix(B):
    P = np.matmul(B,B.T)
    return P


def PCA_sorted(X, num_components):
    Xbar,mu,std=normalize(X)
    covariance=np.matmul(Xbar,Xbar.T)
    S=covariance
    eigvals,eigvecs=eig_sorted(S)
    eigenSum = eigvals[:num_components].sum()
    B = np.stack(eigvecs[:,:num_components])
    P = np.matmul(B,B.T)
    X_reconstruct=np.matmul(P,X)
    return eigenSum, X_reconstruct
```

```
def PCA_unsorted(X, num_components):
    Xbar,mu,std=normalize(X)
    covariance=np.matmul(Xbar,Xbar.T)
    S=covariance
    eigvals,eigvecs=eig_unsorted(S)
    B = np.stack(eigvecs[:,:num_components])
    P = np.matmul(B,B.T)
    X_reconstruct=np.matmul(P,X)
    return X_reconstruct


covariance = np.matmul(Xbar,Xbar.T)
print(covariance.shape)
print(np.trace(covariance))
S = covariance


sorted_eigvals, sorted_eigvecs = eig_sorted(S)
unsorted_eigvals, unsorted_eigvecs = eig_unsorted(S)
plt.plot(sorted_eigvals, label = 'Sorted␣Eigen␣values')
plt.plot(unsorted_eigvals, label = 'Unsorted␣Eigen␣values')
plt.xlabel("$index$")
plt.ylabel("$EigenValues$")
plt.legend()


def mse(predict, actual):
    return np.square(predict - actual).sum(axis=1).mean()


loss_1 = []
variance = []
reconstructions_1 = []
# iterate over different numbers of principal components, and compute the MSE
for num_component in range(1, 14):
    eigenSum, reconst_1 = PCA_sorted(Xbar, num_component)
    error_1 = mse(reconst_1, Xbar)
    reconstructions_1.append(reconst_1)
    variance.append((num_component, eigenSum))
    loss_1.append((num_component, error_1))

reconstructions_1 = np.asarray(reconstructions_1)
reconstructions_1 = reconstructions_1 * std + mu # "unnormalize" the reconstru
loss_1 = np.asarray(loss_1)

variance = np.asarray(variance)
```

```
loss_2 = []
reconstructions_2 = []
# iterate over different numbers of principal components, and compute the MSE
for num_component in range(1, 14):
    reconst_2 = PCA_unsorted(Xbar, num_component)
    error_2 = mse(reconst_2, Xbar)
    reconstructions_2.append(reconst_2)
    loss_2.append((num_component, error_2))

reconstructions_2 = np.asarray(reconstructions_2)
reconstructions_2 = reconstructions_2 * std + mu # "unnormalize" the reconstru
loss_2 = np.asarray(loss_2)


fig, ax = plt.subplots()
ax.plot(loss_1[:,0], loss_1[:,1], label = 'MSE for sorted Eigen Values');
ax.plot(loss_2[:,0], loss_2[:,1], label = 'MSE for unsorted Eigen Values');
ax.set(xlabel='num_components', ylabel='MSE', title='MSE vs number of principa
plt.legend()


plt.plot(variance[:,0], variance[:,1])
plt.xlabel("Number of principle components")
plt.ylabel("Variance");
```

URL links: Click here to go to our project drive

Inference:
Working directly with high-dimensional data, such as images, comes with some difficulties:

- It is hard to analyze
- Interpretation is difficult
- Visualization is nearly impossible
- Storage of the data vectors can be expensive.

Many dimensions are redundant and can be explained by a combination of other dimensions. High-dimensional data are often correlated. Correlation is a statistical measure that indicates the extent to which two or more variables fluctuate together. **Figure 7** shows the correlation between different feature components. It can be seen that there are some good amount of correlation between features i.e. they are not independent of each other.

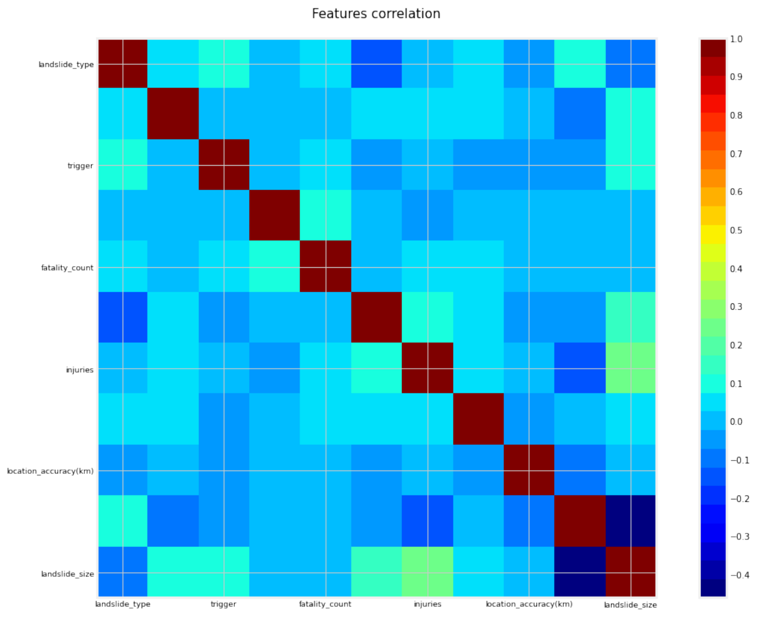


Figure 7: Correlation between features

Dimensionality reduction exploits structure and correlation and allows us to work with a more compact representation of the data without losing much information. **Principal Component Analysis (PCA)** is one of the many methods for Dimensionality Reduction. To find PCA on data set **X**:

- Normalise **X**
- Find Covariance Matrix **S**
- Find **Eigen Values** and **Eigen Vectors**
- Find matrix **B** which is formed by eigen vectors of all components
- Find **Projection matrix (P)**
- Find **Reconstructed Matrix**$(X.P^T)$

As seen above in both the cases, MSE decreases with increase in number of componentswhich signifies decrease in loss of information as number of components increase. Also, the eigen values also decrease with increase in number of components. Variance increases with the increase in number of principal components. We have also compared the running time of both cases. As

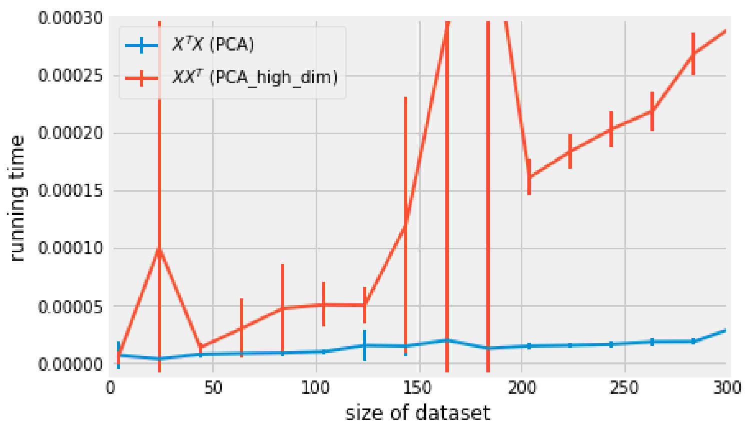


Figure 8: Run Time vs Size of Dataset

observed in above figure, run time for PCA of higher dimension data is comparatively larger than run time for PCA with more samples.