Computer Vision
Assignment # 2

Submitted by

Ali Akram 260336
Mirza Ahmed Aftab 258495
Muhammad Umer Ahsan 243538

DEPARTMENT OF ELECTRICAL ENGINEERING

College of Electrical and Mechanical Engineering (CEME)

Assignment 2

Face Recognition using K Nearest Neighbours and PCA

Objective:

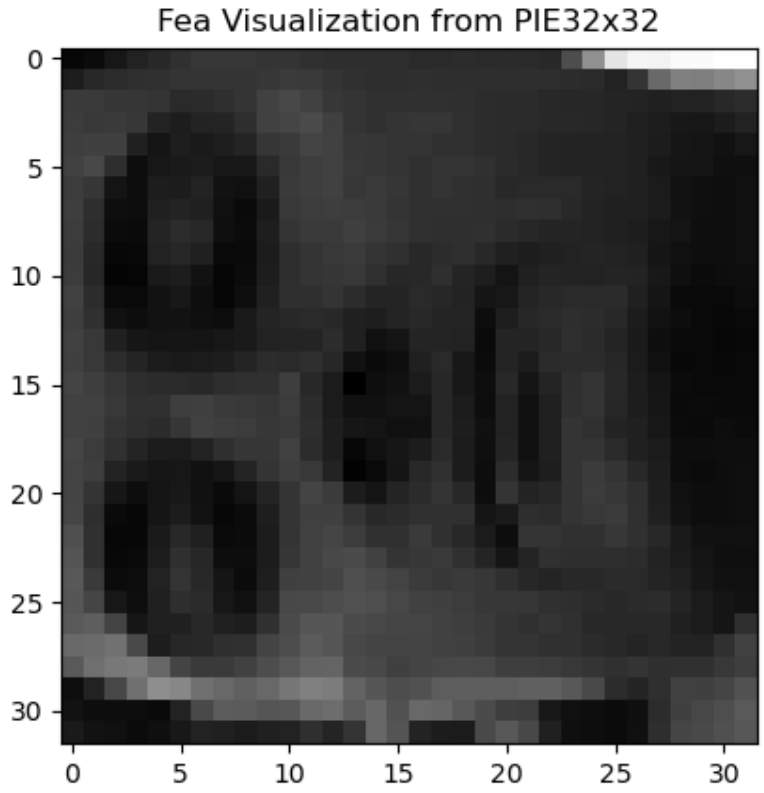
To perform face recognition using KNN and repeat it after dimensionality reduction using PCA and compare the results.

The CMU PIE Dataset:

Original dataset has large number of images including different PIE (Pose, Illumination, expression) for a person.

Now, there are 1700 images. We have 10 subjects i.e. 10 classes 170 images for each. Gnd contains labels 1 to 10, 1 for 1st 170 images 2 for next 170 images and so on.

If we reshape the fea vector to 32x32, we can visualize the image as following:



Libraries Used:

```
import numpy as np
import scipy
from sklearn.model_selection import train_test_split
import random as randd
import time
from matplotlib import pyplot as plt
```

Procedure:

- Loading data from mat file and normalizing it.
- Splitting the data randomly using random library and train_test_split.
- Implementing KNN Function with 3 different distance measures.
- Tune Value of K (Nearest Neighbors).
- Check their accuracy, Standard Deviation and Time.
- Implement PCA function.
- Perform Dimensionality reduction using PCA.
- Use KNN to classify the reduced dimensions data.
- Check their accuracy, Standard Deviation and Time.

1-Loading Data from mat file:

- We read datasets using pandas. After that we extract feature and ground truth from the data and then we normalize the feature by dividing each vector with its magnitude.

```
fea = pd.read_csv('fea.csv')

fea = fea/np.linalg.norm(fea, axis = 1, keepdims=True)
```

2-Splitting Data into test and train:

- For this step we have used function from sklearn library.
- For different random sets, we have used the random state argument of test_train_split. This argument controls the shuffling applied to data. We use this so we can check if the results are stable across different shuffles and splits. Its value normally is between 0 and 42 so that is why we have used randint from random library it will generate a random number each time the function is run.
- The function has 6 arguments as shown below

```
def Split(fea,dimensions,train_imgn,test_imgn,Total,Classes):
```

- Fea is feature vector, dimensions are number of dimensions which we need to change after PCA and are required as we have to create an empty array to store the data. Train_imgn and test_imgn are numbers for test and train images which is 150 and 20 and others values are also used for results. Total and classes are used if we want to have different classes for example for 8 classes, we will pass 1360 as total and 8 as classes
- We split the data using for loop with range of 170 and assign them label using gnd.
- So, the whole function basically divides the data into test and train based on the arguments.

```
def Split(fea,dimensions,train_imgn,test_imgn,Total,Classes):
  training_data = np.empty([train_imgn * Classes, dimensions], dtype=float)
  training_label = np.empty([train_imgn * Classes, 1], dtype=int)
  test_label = np.empty([test_imgn * Classes, 1], dtype=int)
  test_data = np.empty([test_imgn * Classes, dimensions], dtype=float)
  random=randd.randint(0,42)
  i=0
  for x in range(0,Total,170):
    featur=fea[x:x+170]
    fea_tra,fea_test=train_test_split(featur,test_size=test_imgn,train_size=train_imgn,random_state=random)
    training_data[i*train_imgn:(i*train_imgn)+train_imgn]=fea_tra
    test_data[i*test_imgn:(i*test_imgn)+test_imgn]=fea_test
    training_label[i*train_imgn:(i*train_imgn)+train_imgn]=int(gnd[x])
    test_label[i*test_imgn:(i*test_imgn)+test_imgn]=int(gnd[x])
    i=i+1
  return training_data,training_label,test_data,test_label
```

K Nearest Neighbors Function:

- We know that in KNN no learning is involved as the model stores our entire dataset and classifies the test data based on points like them. It makes its predictions based on training data only.
- Now the main thing in KNN is the distance measure we use, here we have used 3 different distance measure i.e. Euclidean, MahalaNobis, and Cosine Similarity. Results are displayed with all of them being used.
- Now Based on the distance measure used we calculate(distance between a test vector and each vector of training data) and store these distances in an array(appending the distance array in our case).
- Then we sort them in ascending order, we also preserve the index by using Np.argsort.
- This gives us the K indexes of smallest distances in the distance list (stored in indexes)
- We then extract the labels from training data which corresponds to these indexes. (labels)
- Then using mode from scipy.stats we find the maximum repeating label and then return it as it is our prediction

Input of KNN is whole training data and labels, 1 instance of test data, number of K neighbors and distance measure that we want to use

- Code is shown below
- Euclidean Distance formula was simply implemented, and distance was returned, same is the case for mahalanobis distance. But for the Cosine Similarity we look for greater values so that is why we have returned 1- distance

```python
def KNN(training_data,training_labels,test_data,k,measure):
    distance=[]
    if(measure=='Euclidean'):
        for i in range(len(training_data)):
            dist = np.sqrt(np.sum((training_data[i] - test_data) ** 2))
            distance.append(dist)
    if (measure == 'Mahalanobis'):
        inv_cov = np.linalg.inv(np.cov(training_data, rowvar=False))
        for i in range(len(training_data)):
            dist = np.matmul(np.matmul(np.transpose(training_data[i] - test_data), inv_cov),
                             (training_data[i] - test_data))
            distance.append(np.sqrt(dist))
    if (measure == 'Cosine Similarity'):
        for i in range(len(training_data)):
            dist = np.dot(training_data[i], test_data) / (np.linalg.norm(training_data[i]) * np.linalg.norm(test_data))
            distance.append(1-dist)
    indexes=np.argsort(distance)[:k]
    labels=training_labels[indexes]
    mode=scipy.stats.mode(labels)
    label=mode[0]
    return label
```

Principal Component Analysis PCA:

- PCA is a very famous technique for dimensionality reduction. It is an unsupervised method. It is very difficult to visualize if the data has more than 3 dimensions. This is commonly referred as curse of dimensionality as well. What PCA does is that it reduces the data with many dimensions to a lower number of data while capturing the maximum variability of data as well. It also makes the training time of any classifier or algorithm faster as we must deal with less dimensions
- 1st of all we subtract the mean of each variable from the dataset. We do this so that dataset is centered on origin. It helps us calculating covariance matrix
- We calculate covariance matrix by using np.cov with rowvar as false. By default, the rowvar is true but for getting correct dimensions we need to set it to false.
- Next we compute the eigen values and eigen vectors of the covariance matrix for this we use np.linalg.eigh. The eigen vectors of covariance matrix are orthogonal and the highest eigen value represents the higher variability
- Next, we sort them in descending order using argsort and giving -1
- Then we extract the first 64 or any number of dimensions we need because the array is in descending order the maximum varying eigen vectors are on tip.
- Then we transform it back by using the dot product between the transpose of both new eigenvectors and mean centered data.

Input of PCA is feature vector and new number of dimensions that we want and the output is reduced features, and the covariance matrices before and after dimensionality reduction

```python
def PCA(X, new_dim):
    # Subtracting Mean from data
    X_meaned = X - np.mean(X, axis=0)
    # Finding Covariance Matrix
    cov_mat = np.cov(X_meaned, rowvar=False)
    # Finding eigen Vales and vectors
    eigen_values, eigen_vectors = np.linalg.eigh(cov_mat)
    # Sorting those eigen values
    sorted_index = np.argsort(eigen_values)[::-1]
    sorted_eigenvalue = eigen_values[sorted_index]
    sorted_eigenvectors = eigen_vectors[:, sorted_index]
    # Picking first (new dim) vectors as they are principal compenents
    eigenvector_subset = sorted_eigenvectors[:, 0:new_dim]
    # Transforming data
    X_reduced = np.dot(eigenvector_subset.transpose(), X_meaned.transpose()).transpose()
    # Calcuting covariance matrix of data
    cov_mat_PCA = np.cov(X_reduced, rowvar=False)

    return X_reduced, cov_mat,cov_mat_PCA
```

Mainn Function:

- **This function basically calculates the accuracy**
- **For each correct prediction it adds 1 and then divide with total test instances to get accuracy**
- **We have also used time library here to calculate time for different instances**

```python
def Mainn(training_data, training_label,test_data,test_label,K,measure):
    S=time.time()
    x=0
    for i in range(len(test_data)):
        prediction = KNN(training_data, training_label, test_data[i], K,measure)
        if prediction == test_label[i]:
            x=x+1
    accuracy=(x/len(test_data))*100
    T=time.time()
    return accuracy,(T-S)
```

Executing the Code and Results

```
for i in range(0,5):
    #calcaccuracy(fea,dim,train_size,testsize,total_images,classes)
    training_data,training_label,test_data,test_label=Split(fea,1024,150,20,1700,10)
    accuracy_euc,time_euc=Mainn(training_data,training_label,test_data,test_label,3,'Euclidean')
    accuracy_mahal,time_mahal = Mainn(training_data,training_label,test_data, test_label, 3, 'Mahalanobis')
    accuracy_cosine,time_cosine = Mainn(training_data,training_label,test_data, test_label, 3, 'Cosine Similarity')
    a_euc.append(accuracy_euc)
    a_mahal.append(accuracy_mahal)
    a_cos.append(accuracy_cosine)
    t_euc.append(time_euc)
    t_mahal.append(time_mahal)
    t_cos.append(time_cosine)
```

Code is executed and time and accuracy are stored in respective arrays

Above snip shows accuracy and time calculation and below one shows the code for plots. Same code is used for all plots. Py file is attached

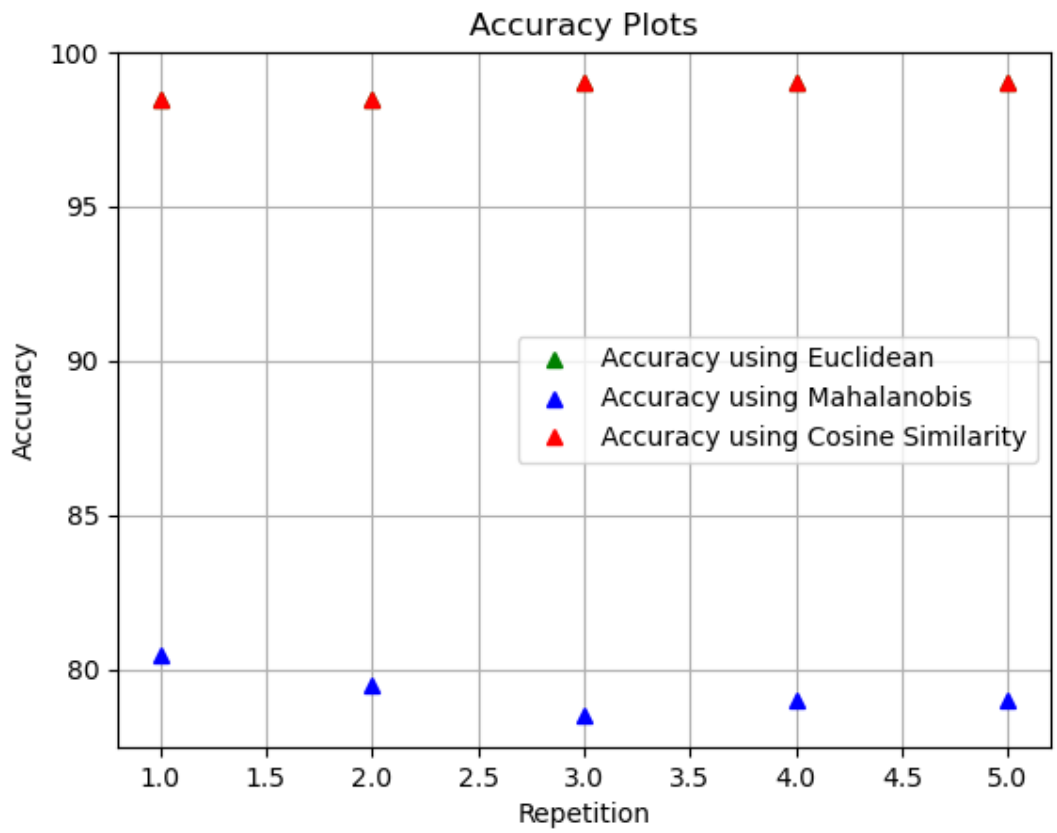
```
x=[1,2,3,4,5]
plt.plot(x,a_euc,'g^',label='Accuracy using Euclidean')
plt.plot(x,a_mahal,'b^',label='Accuracy using Mahalanobis')
plt.plot(x,a_cos,'r^',label='Accuracy using Cosine Similarity')
plt.xlabel('Repetition')
plt.ylabel('Accuracy')
plt.title('Accuracy Plots')
plt.legend()
plt.grid()
plt.show()

plt.plot(x,t_euc,'g^',label='Time using Euclidean')
plt.plot(x,t_mahal,'b^',label='Time using Mahalanobis')
plt.plot(x,t_cos,'r^',label='Time using Cosine Similarity')
plt.xlabel('Repetition')
plt.ylabel('Time')
plt.title('Time Plots')
plt.legend()
plt.grid()
plt.show()

plt.figure(figsize=(9, 9))
plt.imshow(covariance, cmap='gray');
plt.title('Covariance Matrix before PCA')
plt.show()
# Now plotting Covariance Matrix after Dimensionality reduction
plt.imshow(covariance_PCA, cmap='gray');
plt.title('Covariance Matrix after PCA')
plt.show()
```

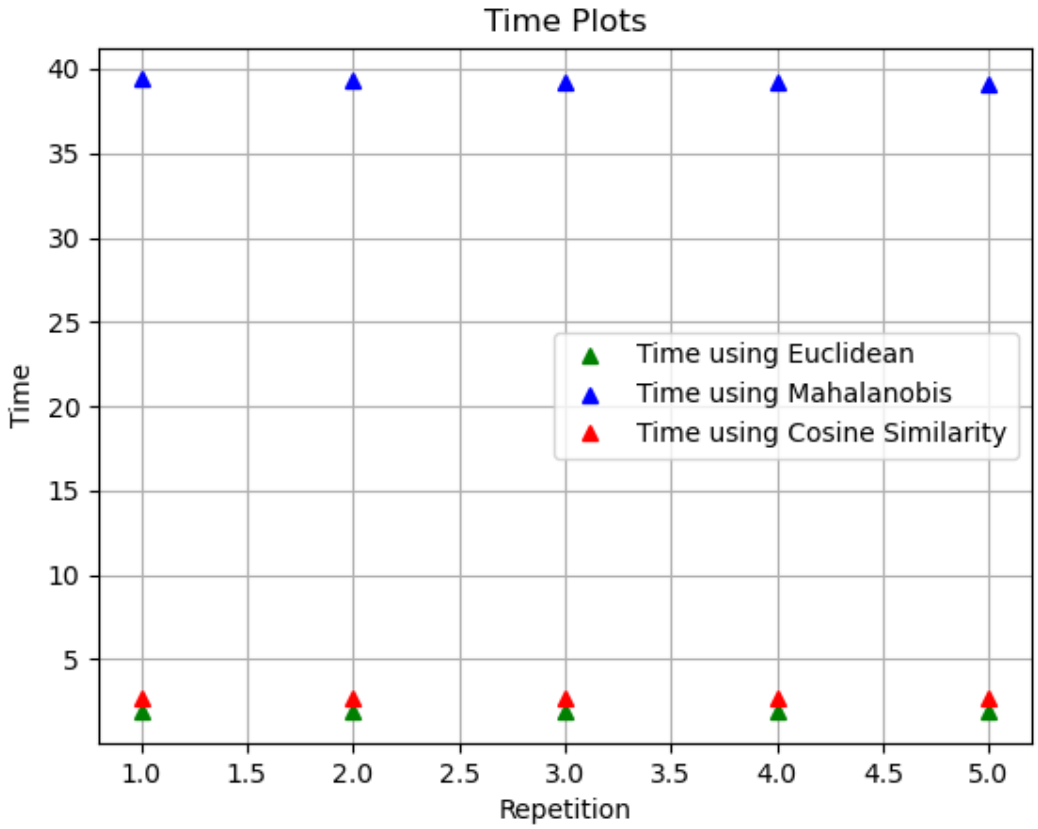
Results:

Using KNN with 3 Neighbors, 5 random repetitions and original 1024 dimensions

Results:

```
Mean Time Taken for KNN using Euclidean 1.8990610122680665
Mean Time Taken for KNN using Mahalanobis 39.240791130065915
Mean Time Taken for KNN using Cosine Similarity 2.678644227981567
Accuracy after random splits 5 time using Euclidean [98.5, 98.5, 99.0, 99.0, 99.0]
Mean accuracy using Euclidean 98.8
Std Deviation using Euclidean 0.2449489742783178
Accuracy after random splits 5 time using Mahalanobis [80.5, 79.5, 78.5, 79.0, 79.0]
Mean accuracy using Mahalanobis 79.3
Std Deviation using Mahalanobis 0.6782329983125268
Accuracy after random splits 5 time using Cosine Similarity [98.5, 98.5, 99.0, 99.0, 99.0]
Mean accuracy using Cosine Similarity 98.8
Std Deviation using Cosine Similarity 0.2449489742783178
```





Now if we look at these results and plots we say that accuracy and time for Euclidean and Cosine is almost same around 98- 99 percent and 3 to 4 seconds. And for mahalanobis we have 79 percent accuracy and around 40 sec time.

Now the reason for this maybe that Mahalanobis distance performs poorly for high dimensional datasets because the complexity of calculating the inverse of covariance matrix increases drastically

Standard Deviation is also more for the case of mahalanobis distance as compared to Euclidean and Cosine

Using KNN with varying Neighbors(K), Same Split and original 1024 dimensions.

Results:

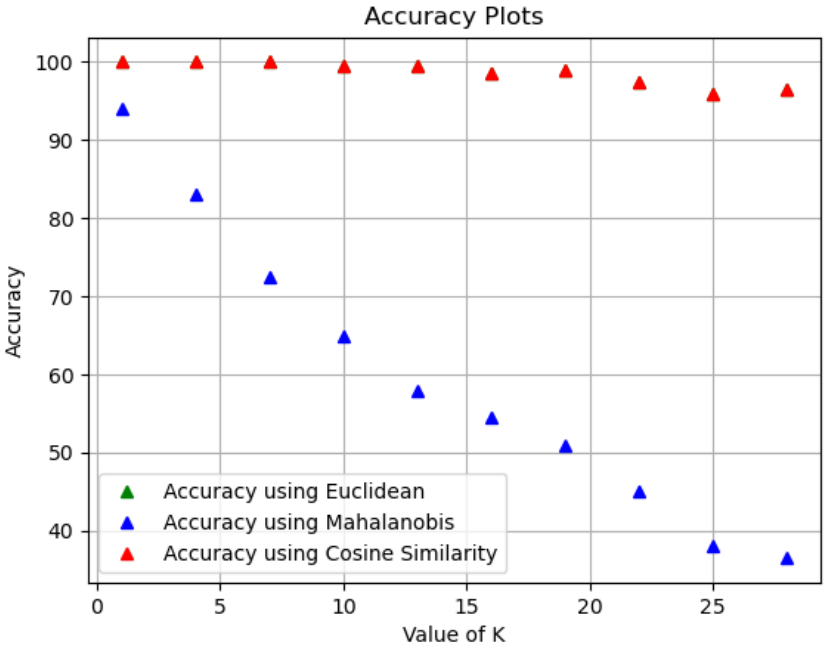
Now I have varied the value of k from 1 to 30. 30 seems a bit too much but again this was for experimenting and visualization. Results are displayed below

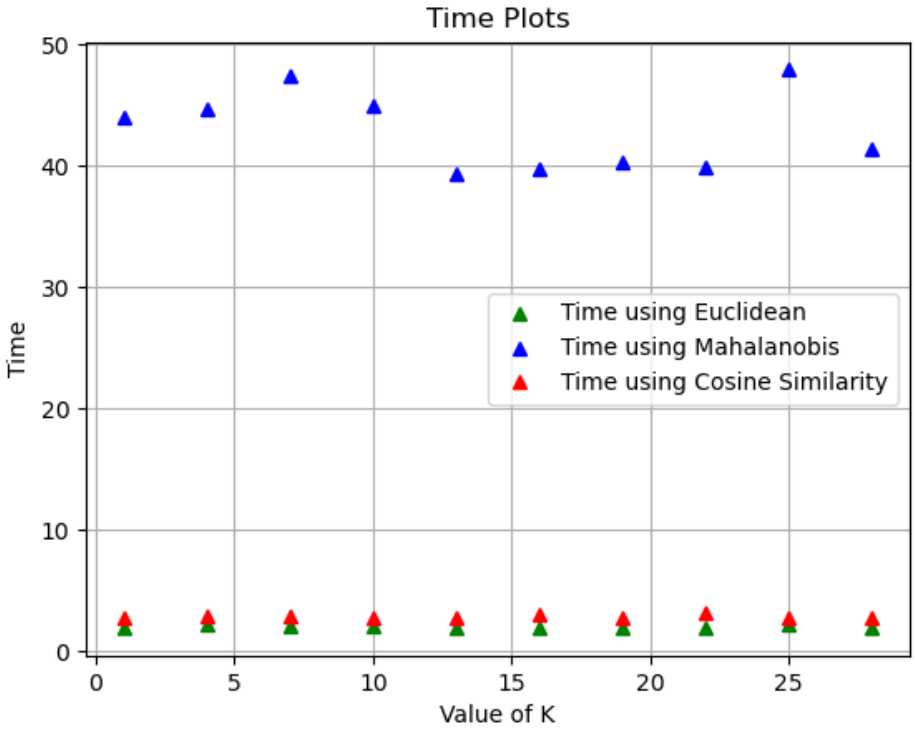
Now there are no pre-defined methods that can be used to calculate the value of k. small values sometimes lead to unstable boundaries. Large values can be good if we get noisy data. But in our case our data was quite refined. So the best value can be find by a plot between accuracy and K which is shown below.

Now for Euclidean and Cosine the accuracy remains almost same for even large number of K but decreases too much for Mahalanobis distance.

Looking at the plots the value of K= 3 seems fine

Time does not change too much for the value of K because we have not used loop for extracting indexes. The use of argsort helps there a lot.

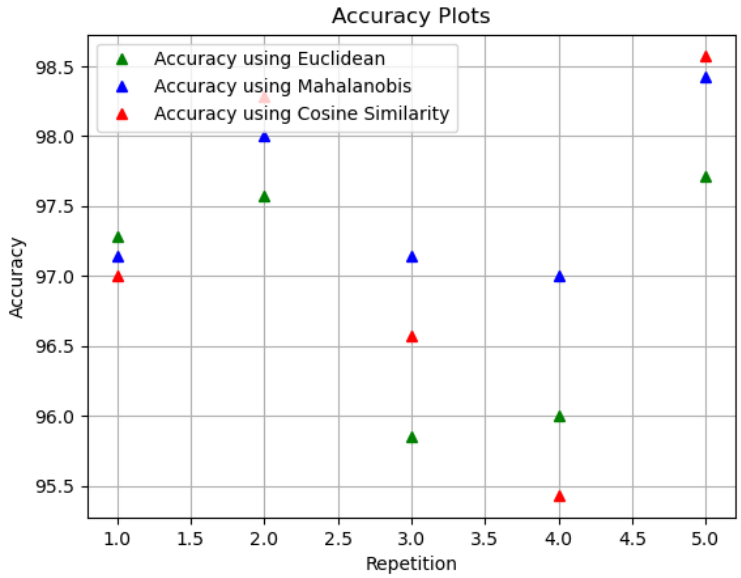


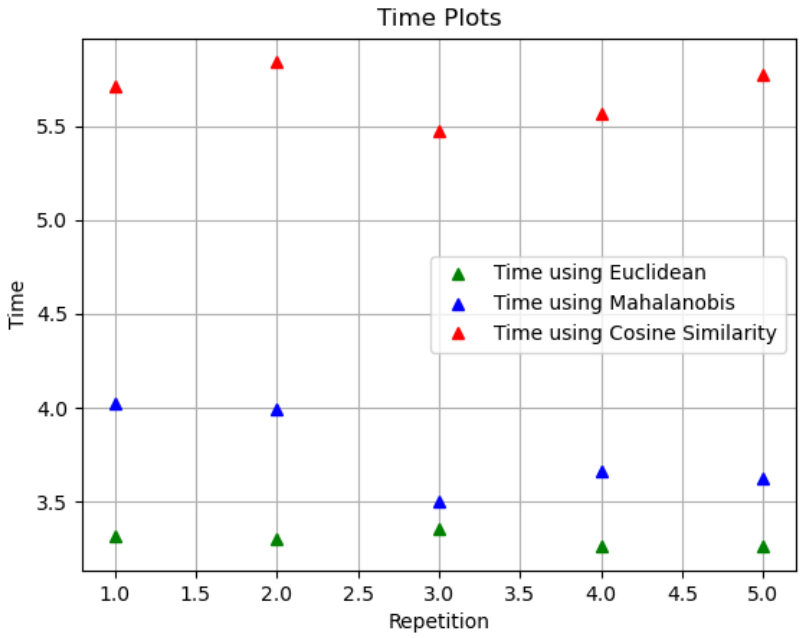


KNN Using 100 training images and 70 test images with original 64 dimensions and 5 random splits:

Now we can see if we compare it to 64 dimensions KNN after PCA with 150 training images

We find that if we increase test images we get less accuracy and time also increases that is why knn is also called lazy

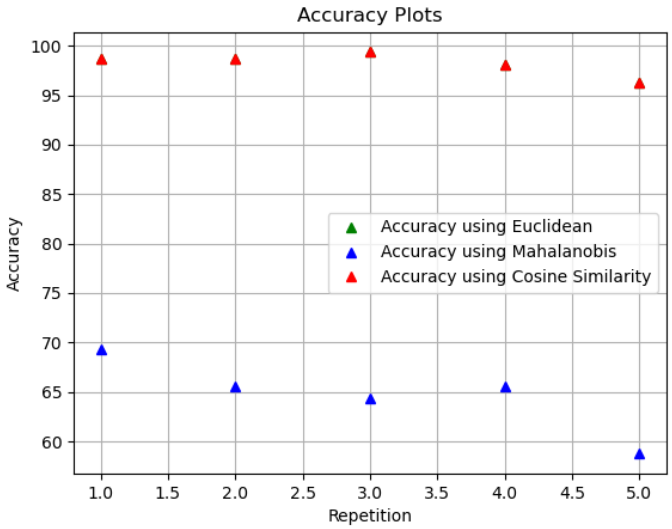


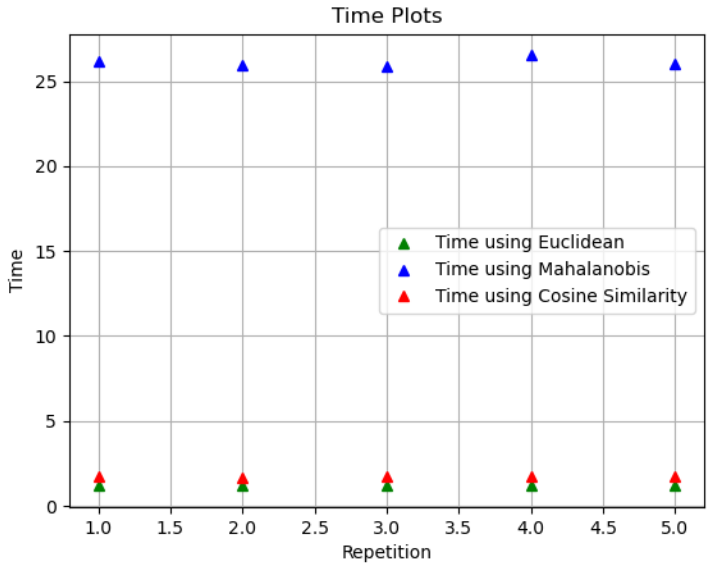


Using Less Classes

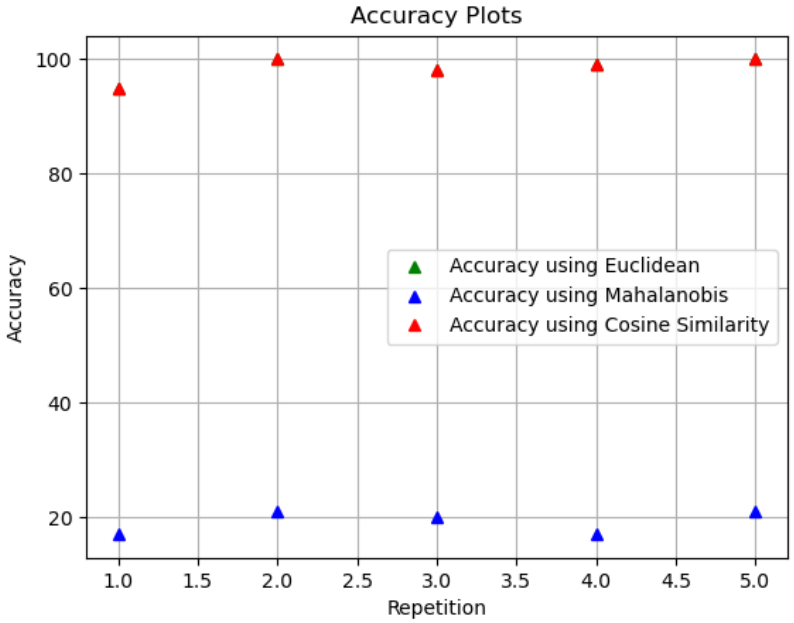
Using less classes, we can see that accuracy for Mahalanobis distance has decreased more. Time however for 8 classes has come from 40 to 25sec. Accuracy for 5 classes in case of mahalanobis has decreased to 20 and time has reduced to 12 sec

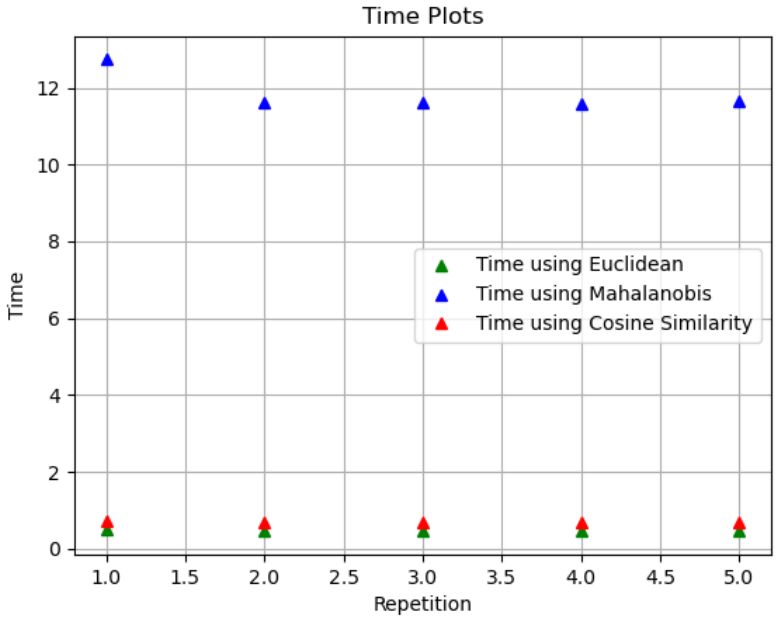
Classes=8





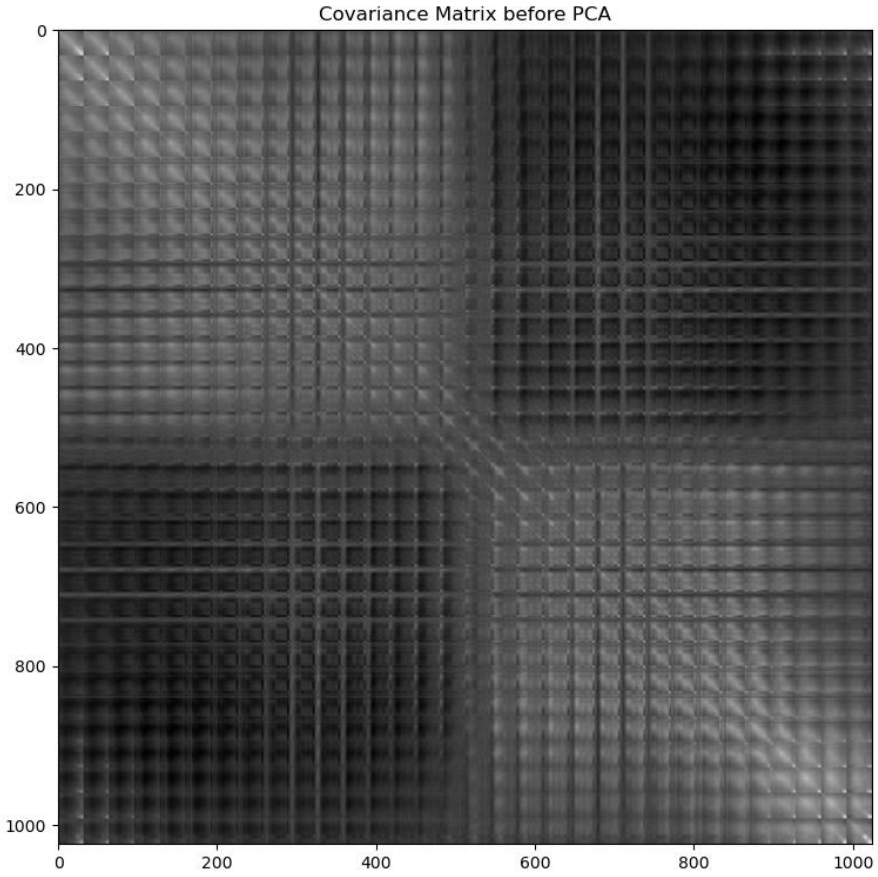
Classes=5


Accuracy Plots
Accuracy
Repetition
Accuracy using Euclidean
Accuracy using Mahalanobis
Accuracy using Cosine Similarity
100
80
60
40
20
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


Time Plots
Time
Repetition
Time using Euclidean
Time using Mahalanobis
Time using Cosine Similarity
12
10
8
6
4
2
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

PART 3

Covariance Matrix Before PCA



We can see from the above visualization that features are corelated

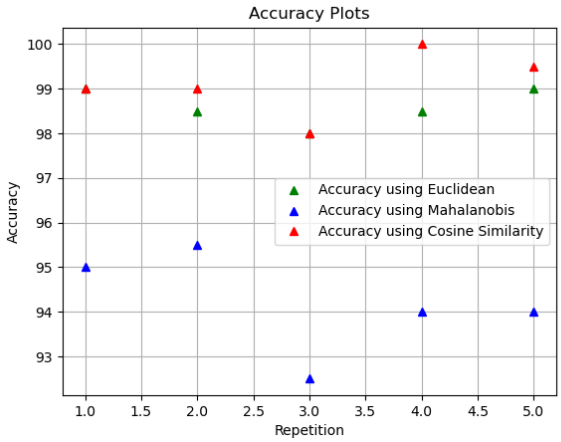
Performing PCA:

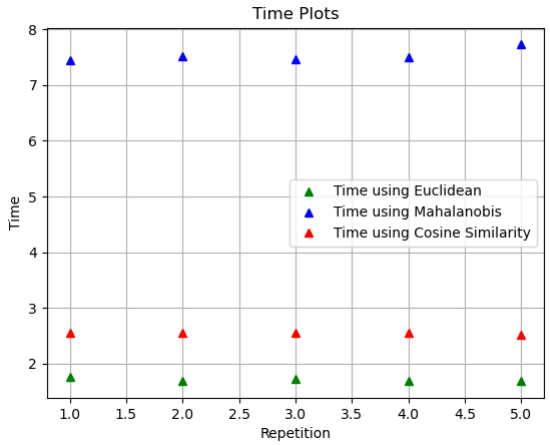
Now PCA has been performed and we use KNN with 170 train images and 20 Test Images

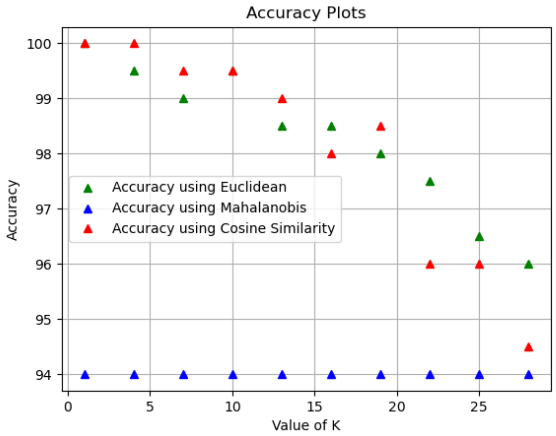
Results are displayed below results contain accuracy, time, std deviation and affect of K.

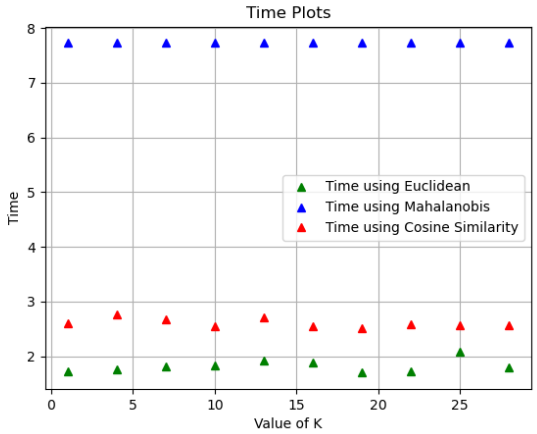
New Dim=512:

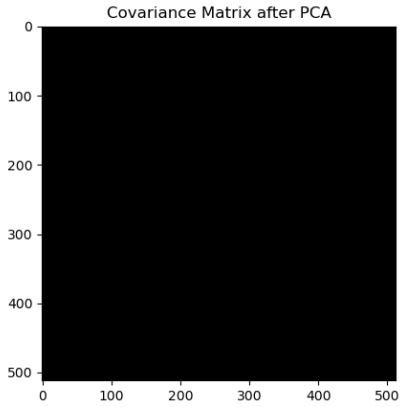
```
Time Taken for PCA 0.1255812644958496
Mean Time Taken for KNN using Euclidean 1.70908579826355
Mean Time Taken for KNN using Mahalanobis 7.532416486740113
Mean Time Taken for KNN using Cosine Similarity 2.5516684532165526
Accuracy after random splits 5 time using Euclidean [99.0, 98.5, 98.0, 98.5, 99.0]
Mean accuracy using Euclidean 98.6
Std Deviation using Euclidean 0.37416573867739417
Accuracy after random splits 5 time using Mahalanobis [95.0, 95.5, 92.5, 94.0, 94.0]
Mean accuracy using Mahalanobis 94.2
Std Deviation using Mahalanobis 1.0295630140987
Accuracy after random splits 5 time using Cosine Similarity [99.0, 99.0, 98.0, 100.0, 99.5]
Mean accuracy using Cosine Similarity 99.1
Std Deviation using Cosine Similarity 0.66332495807108
```





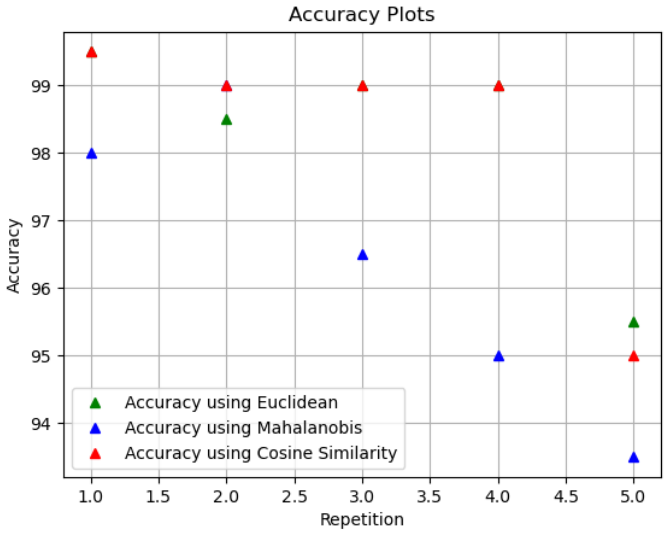

Accuracy Plots
Accuracy using Euclidean
Accuracy using Mahalanobis
Accuracy using Cosine Similarity
Accuracy
Value of K

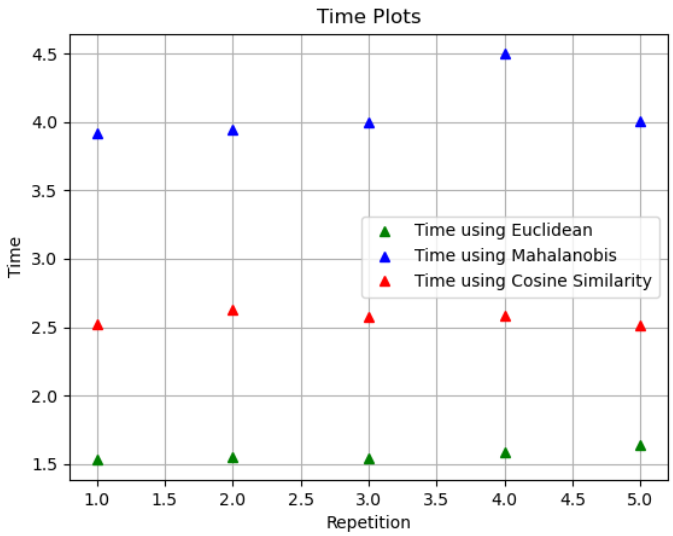

Time Plots
Time using Euclidean
Time using Mahalanobis
Time using Cosine Similarity
Time
Value of K

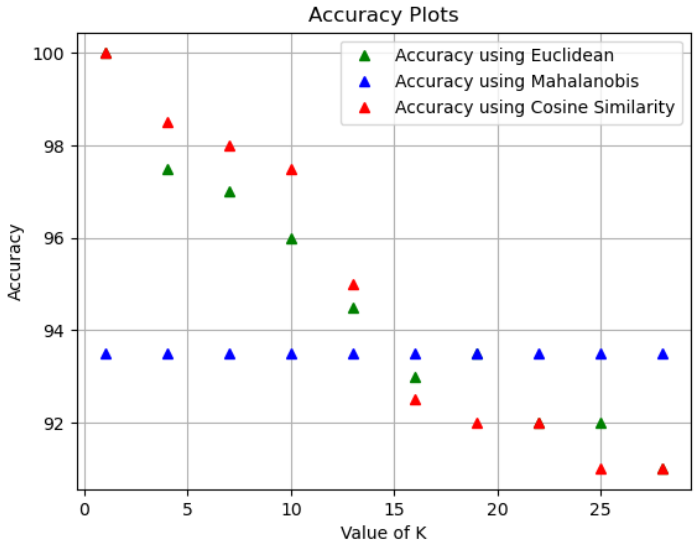

Covariance Matrix after PCA

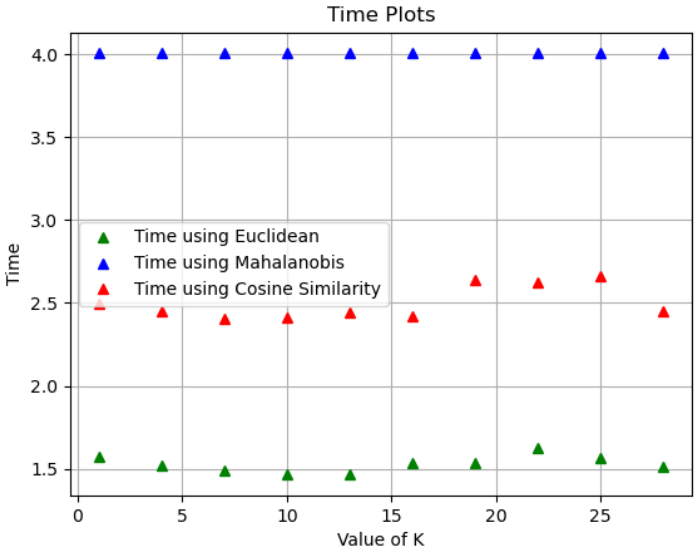
New Dim=256

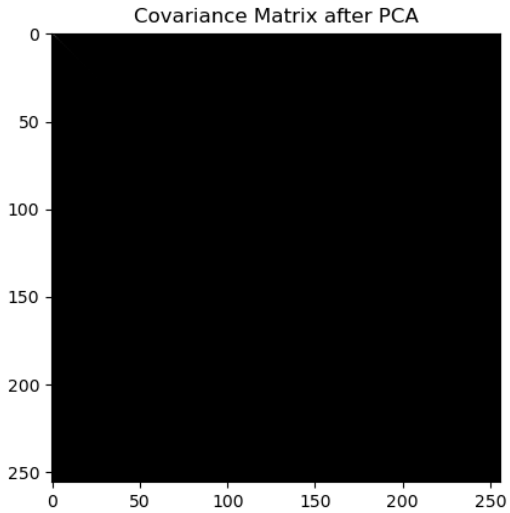
```
Time Taken for PCA 0.11557555198669434
Mean Time Taken for KNN using Euclidean 1.5703518867492676
Mean Time Taken for KNN using Mahalanobis 4.072773170471192
Mean Time Taken for KNN using Cosine Similarity 2.565818691253662
Accuracy after random splits 5 time using Euclidean [99.5, 98.5, 99.0, 99.0, 95.5]
Mean accuracy using Euclidean 98.3
Std Deviation using Euclidean 1.4352700094407325
Accuracy after random splits 5 time using Mahalanobis [98.0, 99.0, 96.5, 95.0, 93.5]
Mean accuracy using Mahalanobis 96.4
Std Deviation using Mahalanobis 1.9849433241279208
Accuracy after random splits 5 time using Cosine Similarity [99.5, 99.0, 99.0, 99.0, 95.0]
Mean accuracy using Cosine Similarity 98.3
Std Deviation using Cosine Similarity 1.661324772583615
```





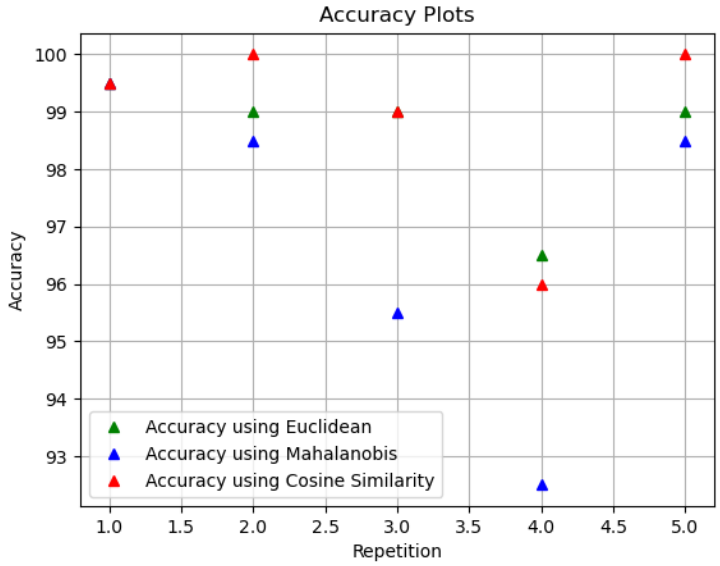

Accuracy Plots
Accuracy using Euclidean
Accuracy using Mahalanobis
Accuracy using Cosine Similarity
Accuracy
Value of K
100
98
96
94
92
0
5
10
15
20
25

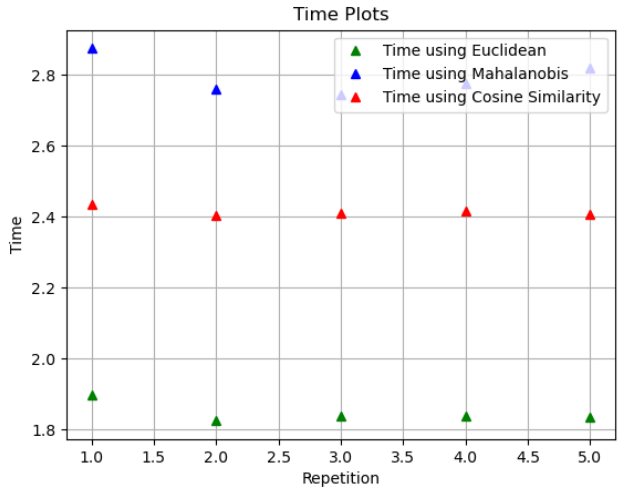

Time Plots
Time using Euclidean
Time using Mahalanobis
Time using Cosine Similarity
Time
Value of K
4.0
3.5
3.0
2.5
2.0
1.5
0
5
10
15
20
25

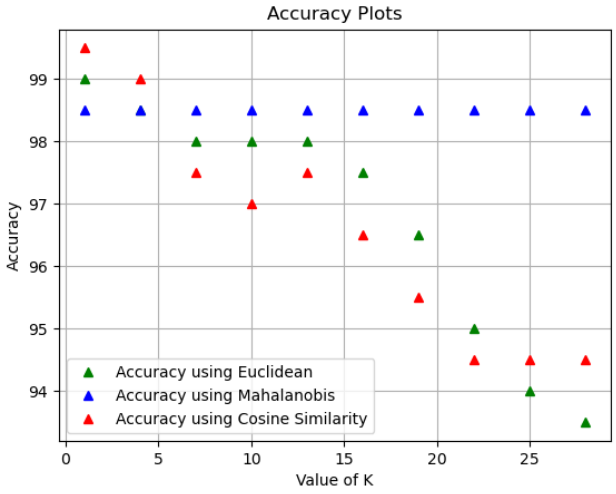

Covariance Matrix after PCA
0
50
100
150
200
250

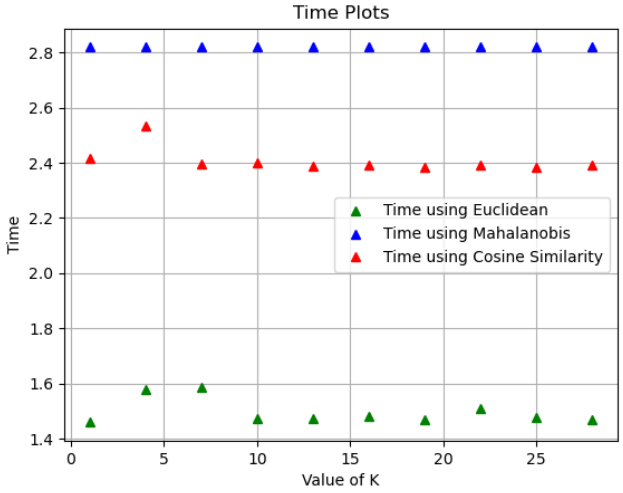
New Dim=128

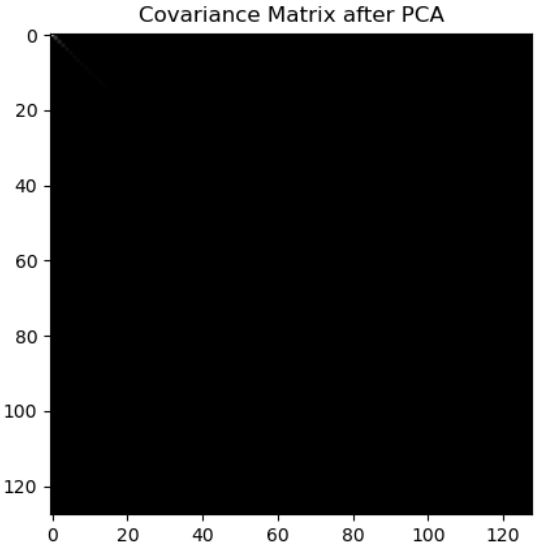
```
Time Taken for PCA 0.12268590927124023
Mean Time Taken for KNN using Euclidean 1.847426176071167
Mean Time Taken for KNN using Mahalanobis 2.7944531440734863
Mean Time Taken for KNN using Cosine Similarity 2.414527654647827
Accuracy after random splits 5 time using Euclidean [99.5, 99.0, 99.0, 96.5, 99.0]
Mean accuracy using Euclidean 98.6
Std Deviation using Euclidean 1.0677078252031311
Accuracy after random splits 5 time using Mahalanobis [99.5, 98.5, 95.5, 92.5, 98.5]
Mean accuracy using Mahalanobis 96.9
Std Deviation using Mahalanobis 2.5768197453450252
Accuracy after random splits 5 time using Cosine Similarity [99.5, 100.0, 99.0, 96.0, 100.0]
Mean accuracy using Cosine Similarity 98.9
Std Deviation using Cosine Similarity 1.4966629547095764
```





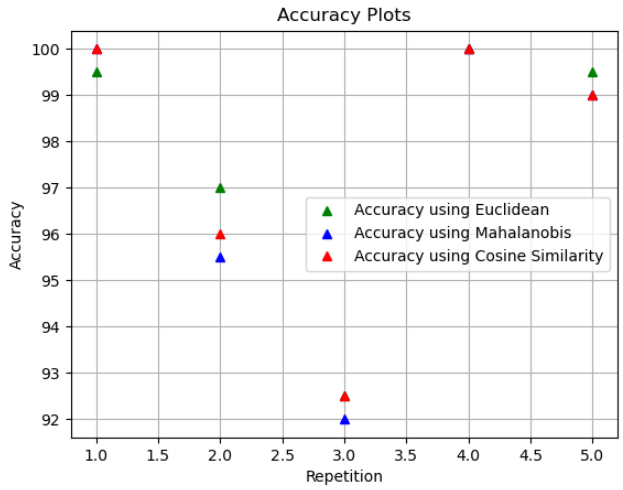

Accuracy Plots
Accuracy
Value of K
Accuracy using Euclidean
Accuracy using Mahalanobis
Accuracy using Cosine Similarity

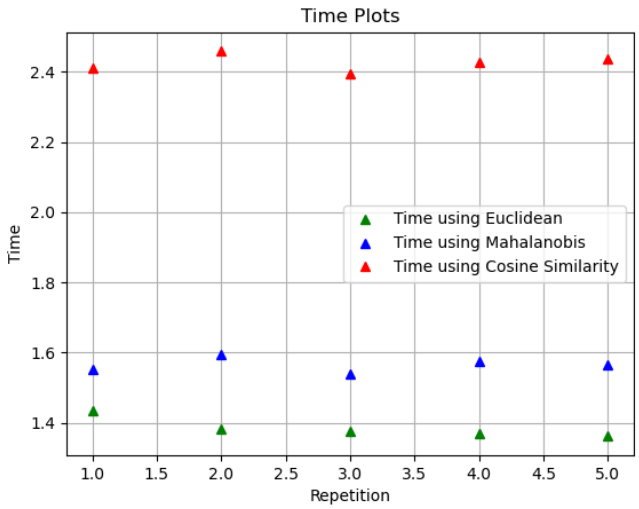

Time Plots
Time
Value of K
Time using Euclidean
Time using Mahalanobis
Time using Cosine Similarity

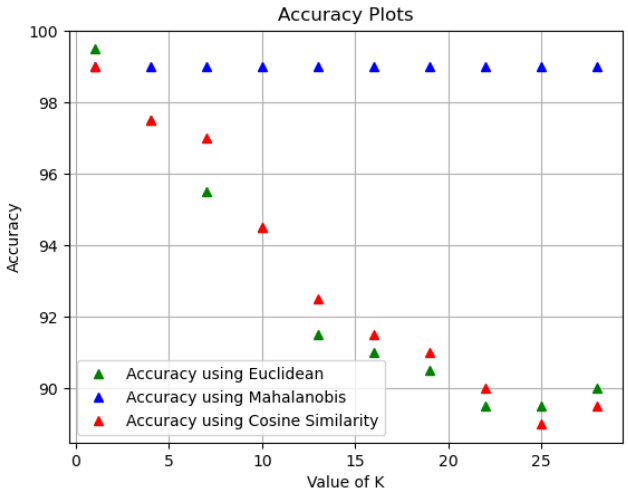

Covariance Matrix after PCA

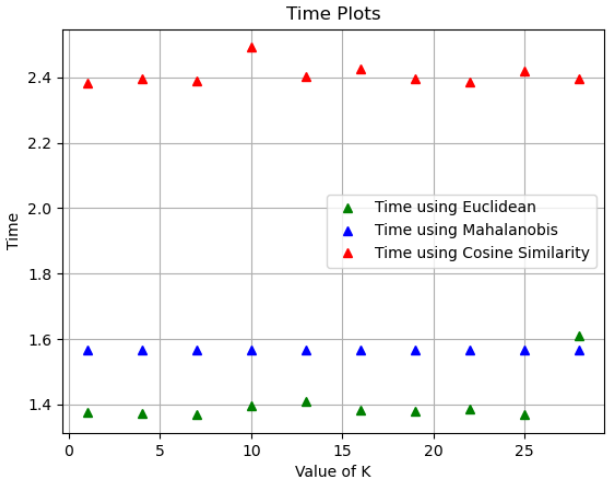
New Dim=64

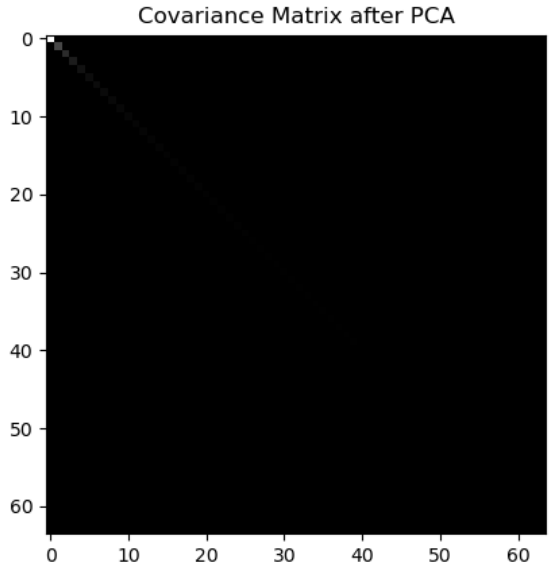
```
Time Taken for PCA 0.11505293846130371
Mean Time Taken for KNN using Euclidean 1.3843690395355224
Mean Time Taken for KNN using Mahalanobis 1.5659643173217774
Mean Time Taken for KNN using Cosine Similarity 2.4258918285369875
Accuracy after random splits 5 time using Euclidean [99.5, 97.0, 92.5, 100.0, 99.5]
Mean accuracy using Euclidean 97.7
Std Deviation using Euclidean 2.803569153775237
Accuracy after random splits 5 time using Mahalanobis [100.0, 95.5, 92.0, 100.0, 99.0]
Mean accuracy using Mahalanobis 97.3
Std Deviation using Mahalanobis 3.1240998703626617
Accuracy after random splits 5 time using Cosine Similarity [100.0, 96.0, 92.5, 100.0, 99.0]
Mean accuracy using Cosine Similarity 97.5
Std Deviation using Cosine Similarity 2.898275349237888
```





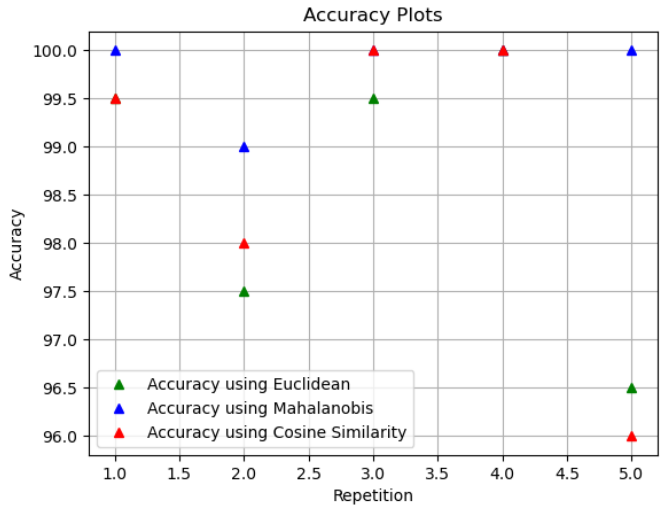

Accuracy Plots
Accuracy
Value of K
Accuracy using Euclidean
Accuracy using Mahalanobis
Accuracy using Cosine Similarity

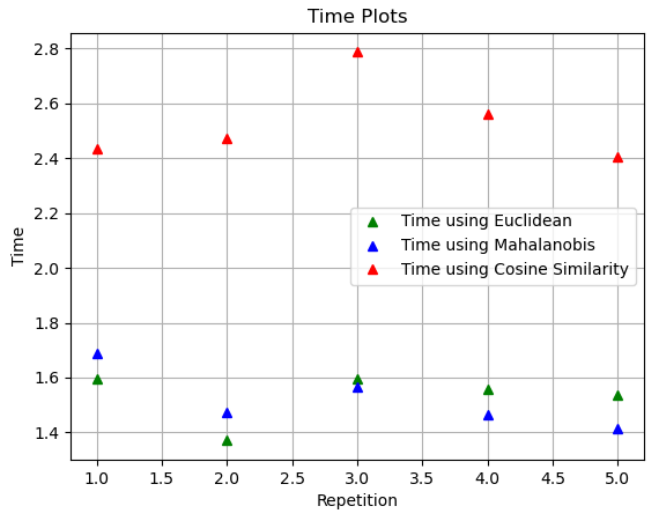

Time Plots
Time
Value of K
Time using Euclidean
Time using Mahalanobis
Time using Cosine Similarity

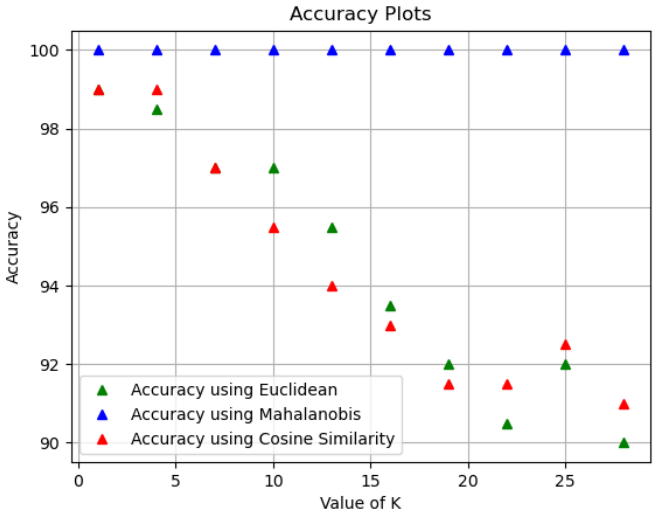

Covariance Matrix after PCA

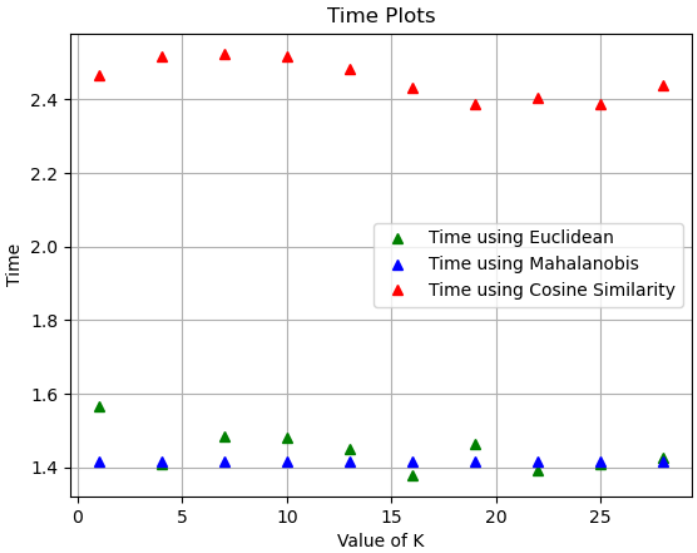
New Dim=32

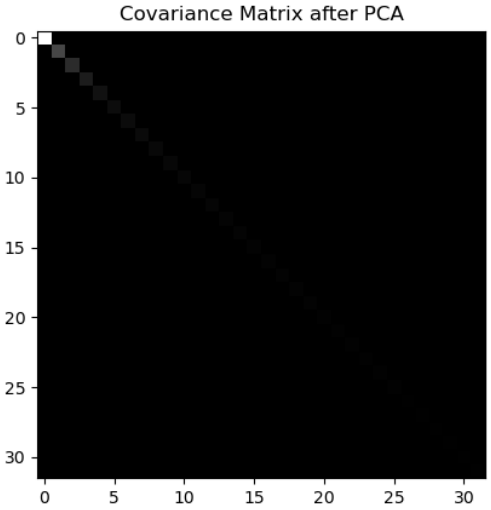
```
Time Taken for PCA 0.12059664726257324
Mean Time Taken for KNN using Euclidean 1.5300957679748535
Mean Time Taken for KNN using Mahalanobis 1.5213095664978027
Mean Time Taken for KNN using Cosine Similarity 2.532074213027954
Accuracy after random splits 5 time using Euclidean [99.5, 97.5, 99.5, 100.0, 96.5]
Mean accuracy using Euclidean 98.6
Std Deviation using Euclidean 1.3564659966250536
Accuracy after random splits 5 time using Mahalanobis [100.0, 99.0, 100.0, 100.0, 100.0]
Mean accuracy using Mahalanobis 99.8
Std Deviation using Mahalanobis 0.4
Accuracy after random splits 5 time using Cosine Similarity [99.5, 98.0, 100.0, 100.0, 96.0]
Mean accuracy using Cosine Similarity 98.7
Std Deviation using Cosine Similarity 1.5362291495737217
```






Accuracy Plots
Accuracy
Value of K
Accuracy using Euclidean
Accuracy using Mahalanobis
Accuracy using Cosine Similarity


Time Plots
Time
Value of K
Time using Euclidean
Time using Mahalanobis
Time using Cosine Similarity


Covariance Matrix after PCA

Explanation:

Now as visible from the PCA plots and results. We can see that accuracy and computation time has improved. This is significant for the case of mahalanobis distance which took around 45-50 sec and takes 1.5 to 2 seconds after PCA and performs way better. Our accuracy has quite increased.

Moreover, we also have visualized the Covariance matrix before and after PCA. We can see that after PCA covariance matrix is a diagonal one and there is no correlation between predictors in diagonal matrix. PCA has improved our results a lot

I have included the results of PCA from 512 dimensions to 32 dimensions. We can clearly see that even at 512 dimensions which is half of 1024. Time Taken by Classifier for Mahalanobis is reduced from 50 seconds to 7 secs which is quite impressive

And for 32 dimensions we have almost 1 sec for everything. Choice of K seems to have no relation with PCA as results remains almost more of the same

Covariance matrices of 512, 256 and 128 have white pixels in principal direction but due to resolution they are not visible we can visualize it properly. For the case of 32 and 64 you can see it properly.

So overall dimensionality reduction has helped us a lot.

Conclusion:

We used KNN for face recognition and also used PCA for dimensionality reduction. We drew following conclusion

Euclidean distance seems to perform best even with high dimensions

Mahala Nobis performs poor with high dimensions

Cosine has as good accuracy as Euclidean but sometimes Euclidean performs better

When test data increases it performs poorly. That is why it is called lazy

PCA helps quite a lot in dimensionality reduction and it increases accuracy and decreases the time

Decreasing classes decreases accuracy for mahalanobis while accuracy for Euclidean and cosine remains same and it also improves time