Dimensionality Reduction and Visualization Project

Caner Canlıer 21702121

Bilkent University

IE 461 Data Science

18th of April, 2022

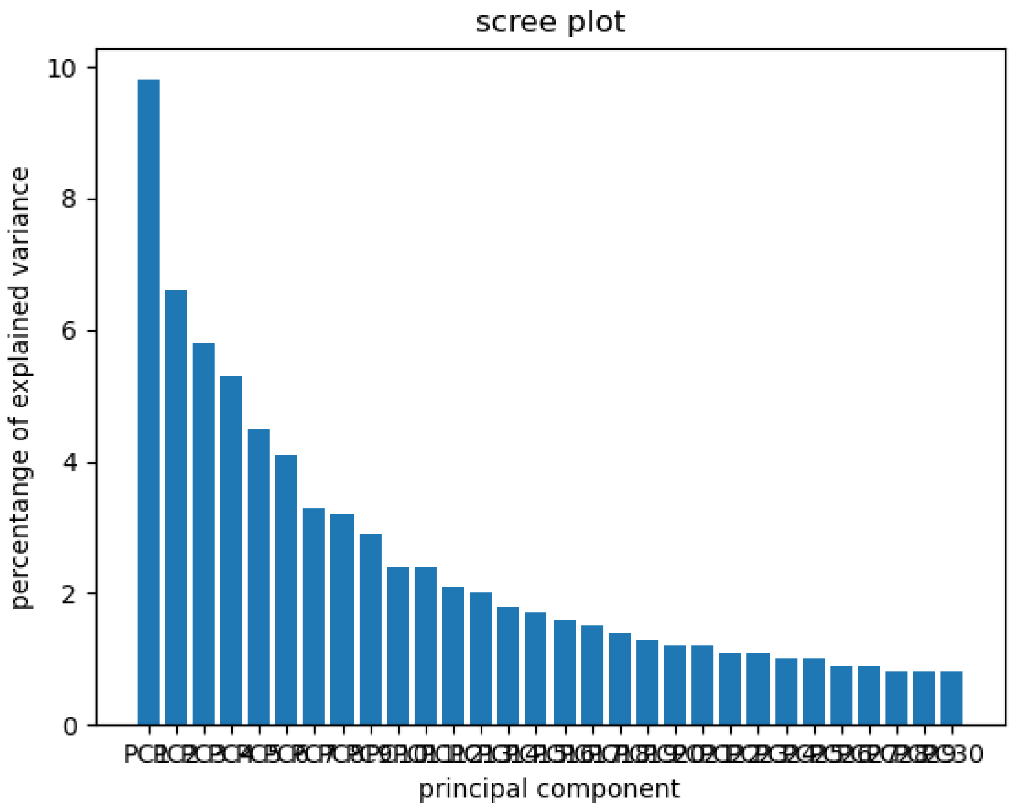
Question 1

In this question, I benefited from sklearn module to do PCA and GaussianNB and matplotlib to draw simple graphs and visualize data. Besides those I used numpy which is a common array module.

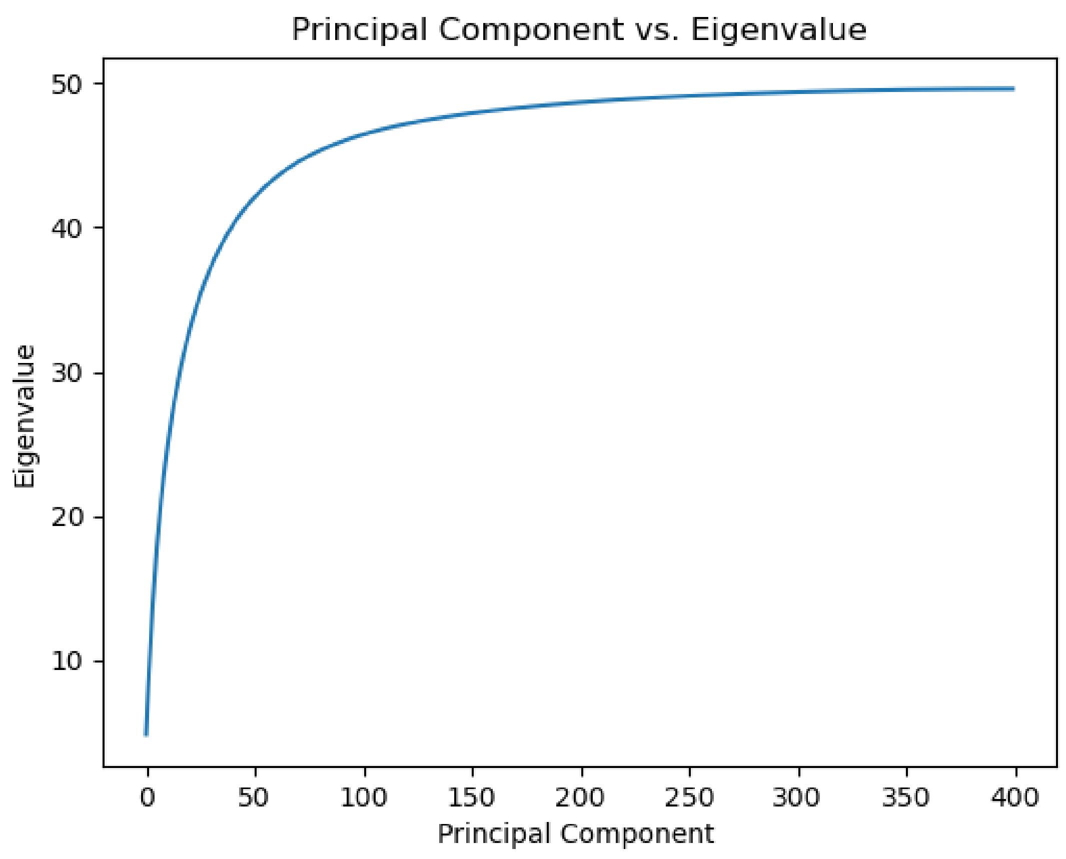
Firstly, I load the data and split the it set into two equal size, one part became my train data, the other one became my test data. To spilt I used train_test_split function from Scikit-learn library.

Question 1.1

For this part, to obtain new bases to project the training data, I do PCA on training data set which is the half of our total data. Since I have 400 features in data set, I found 400 eigenvector. PCA function automatically sorted them in descending order so I didn't have to do any sorting procedure and directly plot them. First, I checked the bar graph of first 30 principal component's percentage of explained variance.



This graph showed me that each principal component explains little of the data. After that I plot the eigenvalues in descending order.



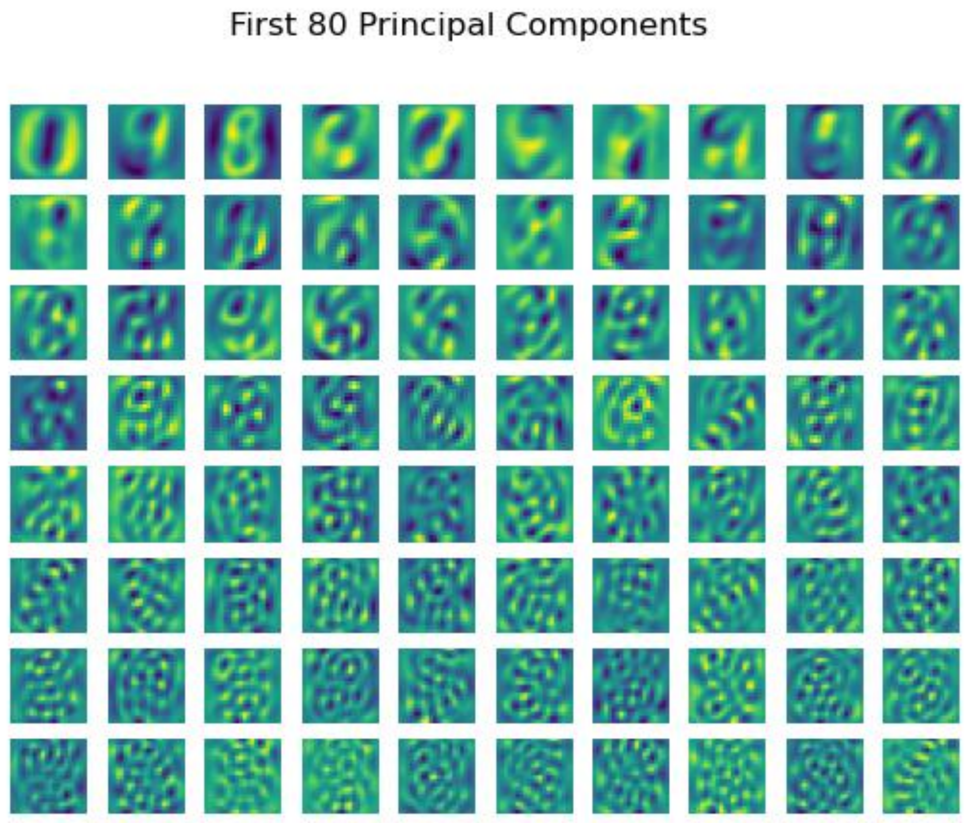
According to this plot, I think we need approximately 50-80 components. This graph suggests that at least 50 components should be chosen because there is a drastic change until that point. After 80 components, the line becomes straighter which indicates that number of principal components should between 50 and 80.

Question 1.2

In this part, to calculate the mean of training data set, I used mean_attiribute of PCA class. After that I transposed the data since it didn't mean anything until transpose it. The transposed image is below.

From this figure we can interpret that the most distinctive feature of the mean plot is circular shape. Although some numbers such as 1 and 4 has sharp edges, 3 and 9 has circular edges, in the mean figure upper and lower parts are dense yellow since all digits have a common feature both in upper part and bottom part. Also, those intensive yellow seems like circular since many numbers (except 1,4,7) have circular upper part (2) or bottom part (5) or both (0,3,6,8,9). Even though upper-bottom parts are common, middle part include less common points for each number so it is not clear in the figure.

Besides the mean figure, I displayed first 80 components. The reason why I choose 80 principal components have been already explained in the previous question but to repeat, we could choose at least 50 or at most 80 principal components according to previous question's plot. I choose the max point.



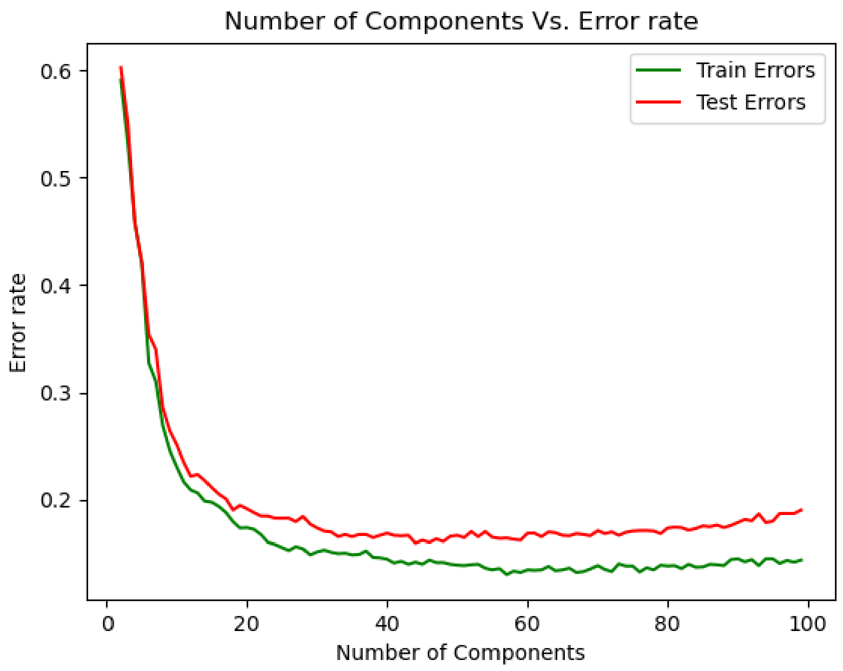
First 7 principal components can be recognized. For instance, first one is 0 then 9,8,3,7… However, after some point, it starts to become too complicated and unclear. It can be concluded that when we increase the principal components, image becomes complicated and hard to interpret.

Question 1.3

For this part, I choose 100 subspace dimensions to measure the impact of dimensionality to classification. I created new PCA with n components (n=1..100) and use fit function in pca to train data. After that I used transform function to project train data into new variables. Subsequently, I benefited from GaussianNB class and trained a Gaussian classifier with the projected train data. I used predict function in GaussianNB class and perform classification on train data. To find classification accuracy, I sum all matched predicts and divide to whole train data. After extracting that accuracy from 1, I achieved train errors and kept them in a list. I went through the same procedure to finding test errors. In the next question, the relationship between number of components and classification error will be showed.

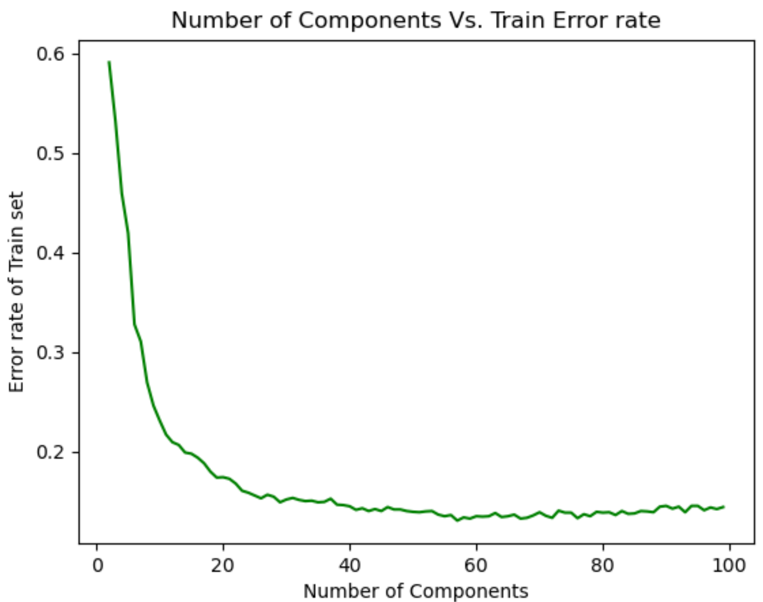
Question 1.4

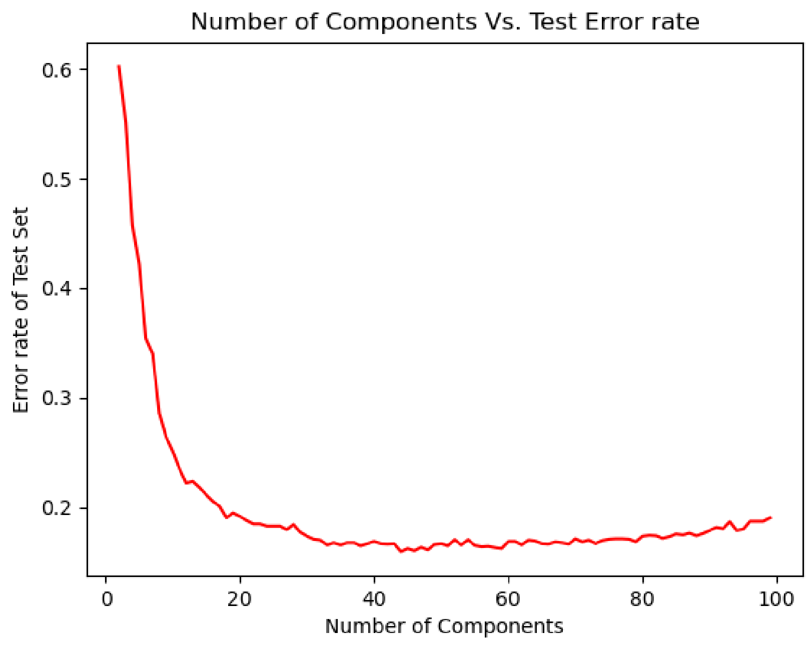
In this part, we can observe the classification error on training and test data.



From the graphs we can see that there is a drastic drop in the both Train error rate and Test error rate at approximately 15-25 components. As expected, test errors are higher than train errors since the model is trained with train data set, so it can predict more accurately the train set data. Even though train errors are lower than test errors, there is no significant difference between those two. To see each line clearly, lets investigate them in separate graphs.

Firstly, lets observe the performance of the model in train data set.





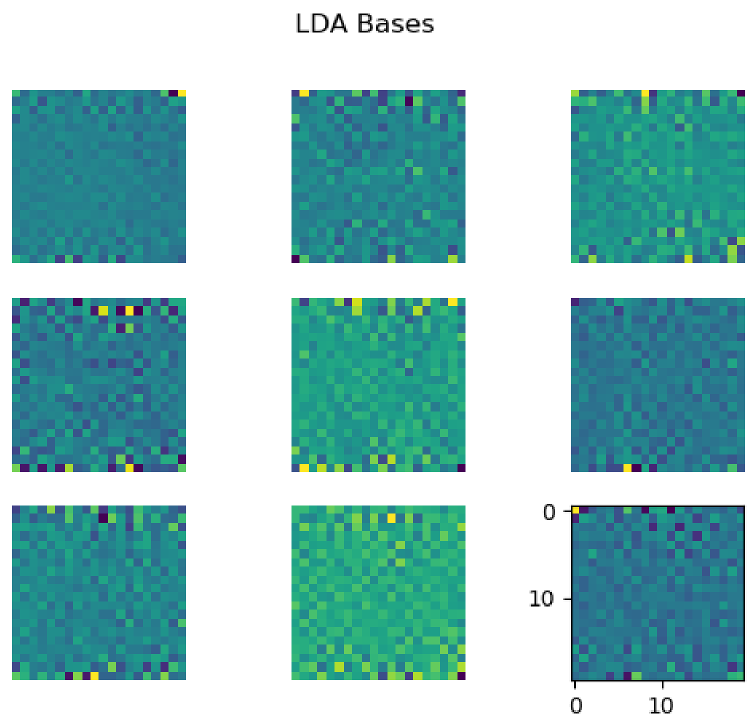
For both graphs, after 25 components, error rate fluctuates and slightly increases. While for test data error fluctuates between 0.15 and 0.10, for train data fluctuation range is smaller. Also in test data, after 80 components, error rate starts to increase to approximately 20%. It can be interpreted as the curse of dimensionality, adding too much feature can cause a drop in model's performance. On the other hand, although we observe a slight increase in error rate in train data set, we don't detect such a high error rate. The main reason of that is since the data is trained with train data, it is familiar to the model.

Question 2

In this question, I benefited from the previously split test and train data set. To apply Fisher linear discriminant analysis (LDA), I used sklearn.discriminant_analysis module and in that module I used LinearDiscriminantAnalysis class. Also I benefited from GaussianNB class of sklearn.naive_bayes module to train a gaussian classifier. I stored my error rates in lists and visualize them by using matlabplot library.

Question 2.1

For this question, I applied LinearDiscriminantAnalysis function on training data set and obtained a new bases to project. I achieve 9 LDA bases. These bases can be seen below as 20x20 images.



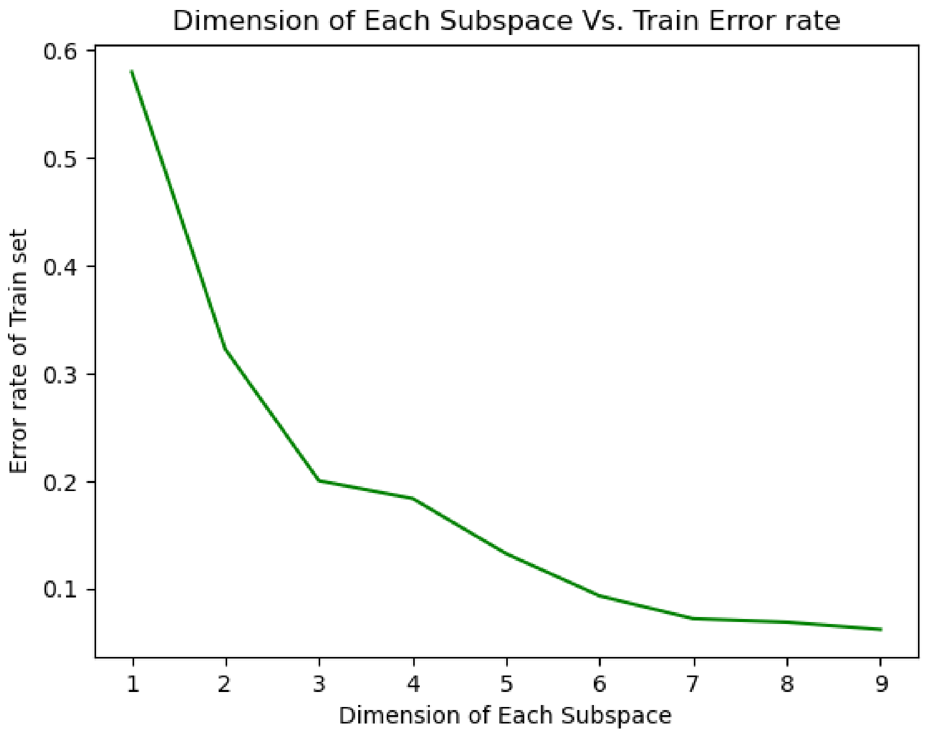
I was expecting to see different result than PCA since while PCA minimizing the reconstruction error, LDA maximizes the class separability. Because LDA bases reflect the vectors that separate the data best, instead of vectors that carry the most information, LDA bases couldn’t identify the numbers. While using PCA, we were able to identify some of the numbers, for LDA it becomes an impossible mission since LDA bases do not resemble any of the digits' patterns. New set of LDA bases can be seen in the above table.

Question 2.2

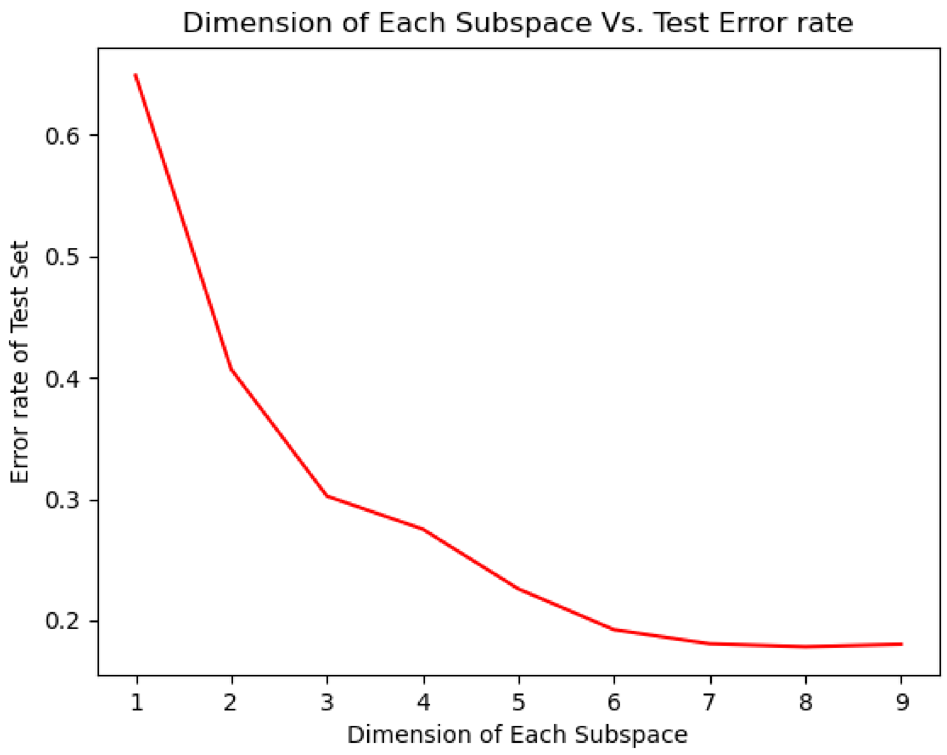
In this question, I did similar thing as the Question 1.3. While for question 1.3, I used PCA function, for this question I used LDA function and dimensions between 1 and 9 to measure the dimensionality impact. Then, I created new LDA with n components and use fit function in lda to train data. After that I used transform function to project train data into new variables. Subsequently, I benefited from GaussianNB class and trained a Gaussian classifier with the projected train data. I used predict function in GaussianNB class and perform classification on train data. I couldn’t find any function to calculate the error rates directly so I calculate them separately. To find classification accuracy, I sum all matched predicts and divide to whole train data. After extracting that accuracy from 1, I achieved train errors and kept them in a list. I went through the same procedure to finding test errors. In the next question, the relationship between number of components and classification error will be showed.

Question 2.3

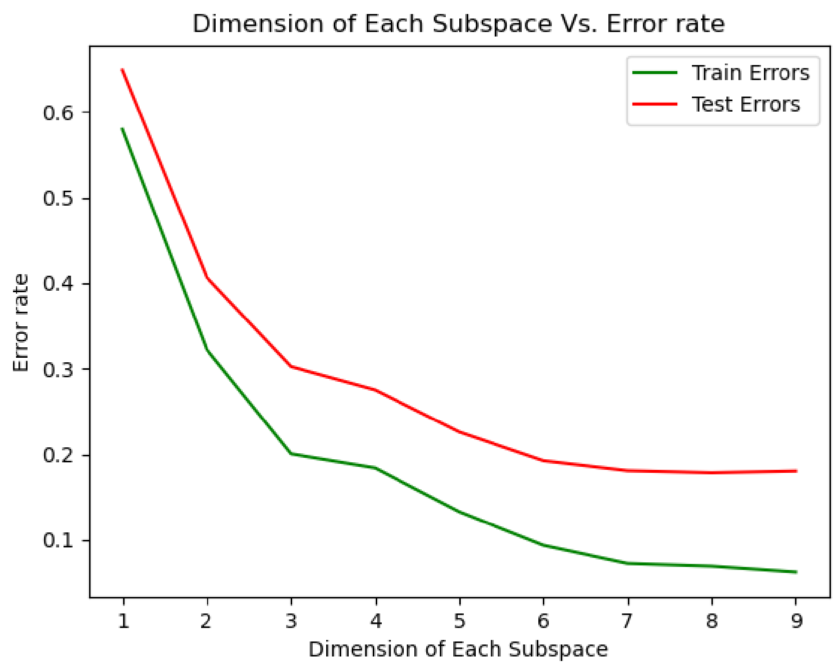
For this question, we should keep in mind that working principals of PCA and LDA are different each other. So we shouldn't expect to get same classification error values. I showed training data and test data errors on two separate plots. Addition to those plots, I also provide another plot which compares the training and test classification errors.



For training data set, as the dimension of the subspace rises, the error rate in classification reduces. The error rate goes down from approximately 58% to 5%. Error rate dramatically drop when we increase the dimension from 1 to 3, however, after 3-dimension error rate starts to decrease slower. Also, from this graph it can be concluded that more LDA components leads to more accurate predictions on training data set. In PCA, while dimension is increasing, error rates are decreasing at some point and then fluctuates, however for LDA, this is not the case. Also it can be seen that minimum and maximum error rate of both PCA and LDA are similar for train data set. While PCA reaches 20% error rate after approximately 20 components, LDA reach the same error rate after 3 dimension of subspace.



For test data set, we cannot such a claim that when the dimension of the subspace rises, the error rate in classification always reduces since after 7-dimension, the error rate stays stable. However, we can still say that generally more LDA components leads to more accurate predictions on test data. The error rate goes down from approximately 65% to 15%. Error rate dramatically drop when we increase the dimension from 1 to 3, however, after 3-dimension error rate starts to decrease slower. In PCA, while dimension is increasing, error rates are decreasing at some point and then fluctuates, however for LDA, this is not the case. Also, it can be seen that minimum and maximum error rate of both PCA and LDA are similar for test data set. While PCA reaches 20% error rate after approximately 20 components, LDA reach the same error rate after 6 dimensions of subspace. So, we can say that, until some point PCA may perform better.



When we compare the train error with the test error, we achieve the similar result as PCA's result. As expected, train error is less than test error. However, while the gap between train and test errors were smaller in PCA when we compare with the LDA.x

Question 3

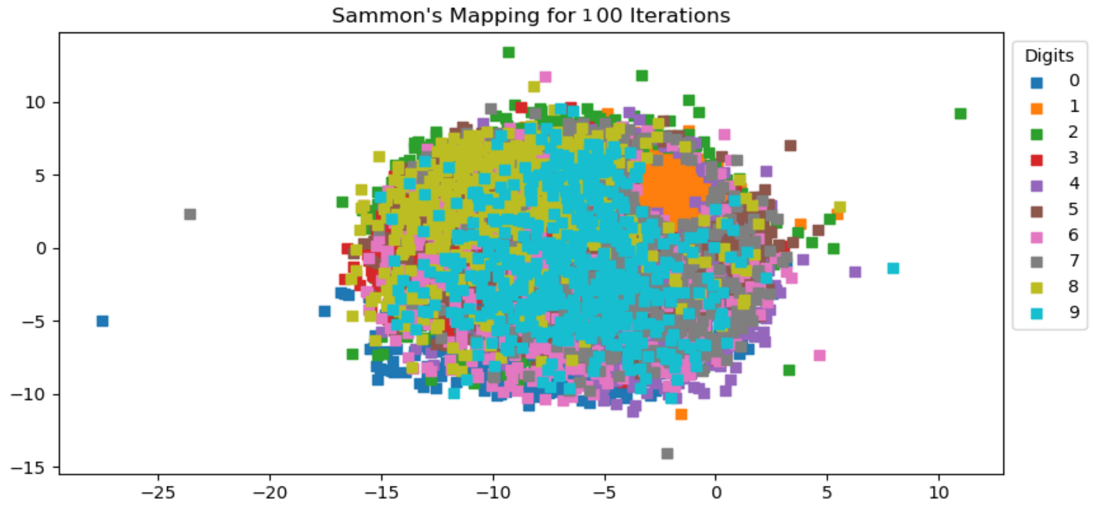
Firstly, I want to introduce and discuss about how did I use Sammon's mapping and t-SNE.

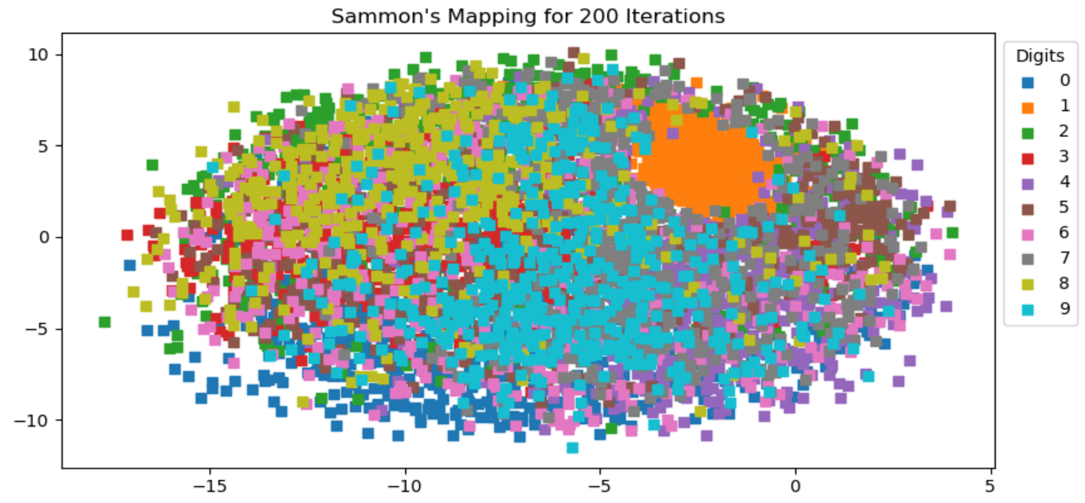
1- Sammon's Mapping

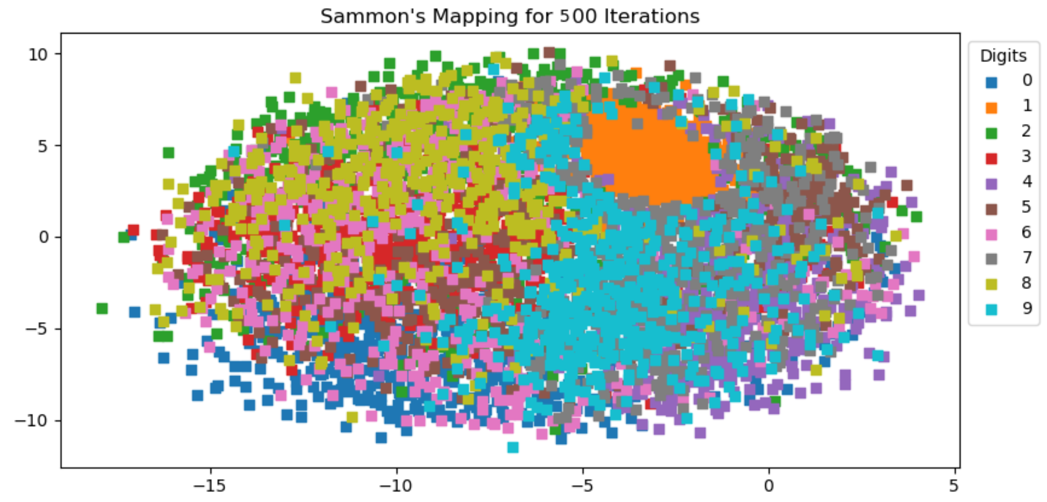
Sammon mapping is an algorithm for mapping a high-dimensional space to a lower-dimensional space while attempting to retain the structure of inter-point distances in the higher-dimensional region. Since I couldn't find any library in python, I tried to search an easy way in GitHub repository. I found what I searched for in GitHub. I want to describe the code that I used:

- **def sammon(x, n, display = 2, inputdist = 'raw', maxhalves = 20, maxiter = 500, tolfun = 1e-9, init = 'default')**
- **y = sammon(x)** - applies the Sammon nonlinear mapping procedure on multivariate data x
- **[y,E] = sammon(x)** also returns the value of the cost function in E (i.e the stress of the mapping).

For our problem, x was the features and maxiter value is the maximum number of iterations. I haven't changed other variables rather than maxiter because others were preventing algorithm to crash. Also, since projection recomputed for each data, algorithm took some time to give a result. After getting the result, I used matplotlib to visualize the result. I tried different numbers of iteration (maxiter value) such as 100,200 and 500. Since I change the code every time I run with different numbers, there is only a main code rather than 3 different codes with 3 different numbers.







From the plots, it can be seen that when iteration count is 100, digits distribution is dense than others. However, it doesn't imply that increasing the number of iterations lead to significant change. Even 500 iterations cannot separate digits properly. Even though after 500 iterations, the digits are still together, when we compare with 100 iterations, it seems like digits are started to appear outside and close to same digits. For instance, let's take a look at the digit 9 which I the blue dot on the graph. While after 100 iterations, it is all over the map, after 500 iterations it starts to gather right hand side of the graph.

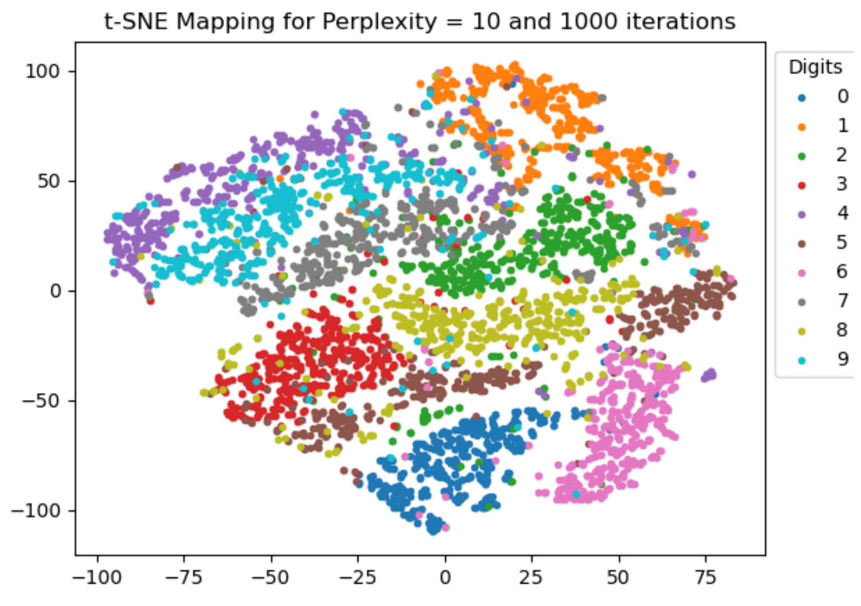
As a general concept, when we increase the iteration count, we gain better understanding. For this problem, we couldn't observe this fact clearly because of low iteration number. Maybe after 5000 iterations, we could observe a better graph.

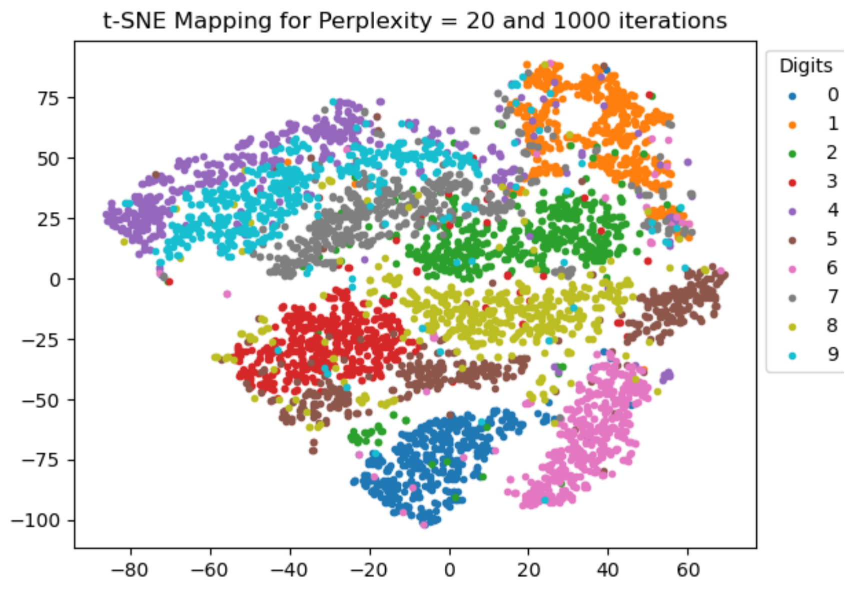
2- t-SNE

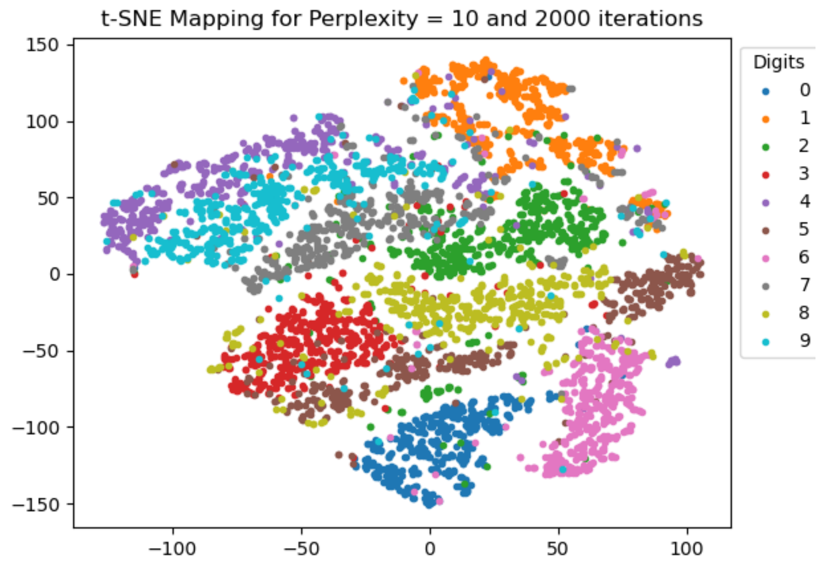
t-Distributed Stochastic Neighbor Embedding (t-SNE) is an unsupervised, non-linear technique is primarily used for data exploration and visualization of high-dimensional data. To put it another way, t-SNE offers a sense of how data is organized in a high-dimensional space. It resembles Sammon's Mapping since it tries to bring the same digits together and have equal number of neighbors around every data point. For t-SNE, I use TSNE class from sklearn library. In that library t-SNE is defined as:

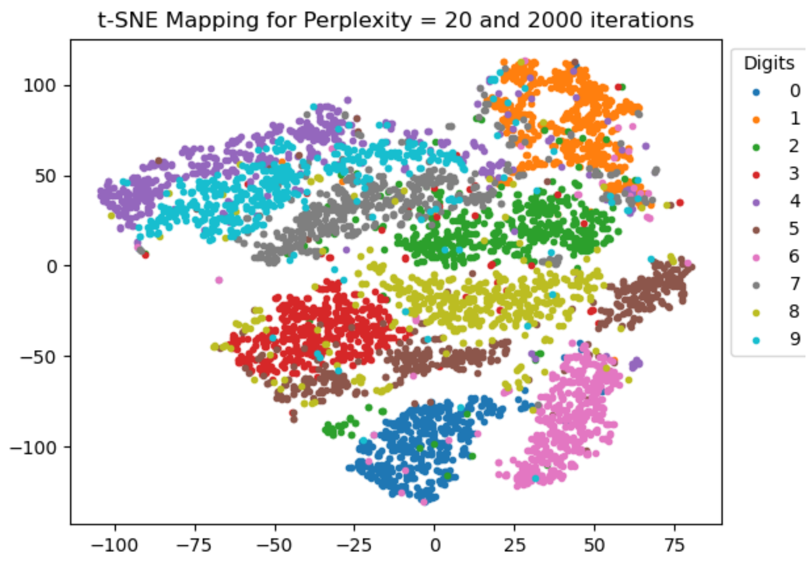
- **tSNE = TSNE(n_components=2, perplexity=30.0, early_exaggeration=12.0, learning_rate='warn', n_iter=1000, n_iter_without_progress=300, min_grad_norm=1e-07, metric='euclidean', init='warn', verbose=0, random_state=0, method='barnes_hut', angle=0.5, n_jobs=None, square_distances='legacy')**

In this function, I only changed perplexity and n_iter. n_iter is the iteration count and perplexity balance the attention t-SNE gives to local and global aspects of the data. I tried different different n_iter values such as 1000 and 2000 and different perplexity values such as 10 and 20. Also, I fixed the randomness by equaling random_state to 10 for each experiment.




t-SNE Mapping for Perplexity = 20 and 1000 iterations
Digits
0
1
2
3
4
5
6
7
8
9
75
50
25
0
−25
−50
−75
−100
−80
−60
−40
−20
0
20
40
60


t-SNE Mapping for Perplexity = 10 and 2000 iterations
Digits
0
1
2
3
4
5
6
7
8
9
150
100
50
0
−50
−100
−150
−100
−50
0
50
100


t-SNE Mapping for Perplexity = 20 and 2000 iterations
Digits
0
1
2
3
4
5
6
7
8
9
100
50
0
−50
−100
−100
−75
−50
−25
0
25
50
75

From the graphs, we can say that increasing perplexities and iteration count close the same digits. However, since our data set contains 5000 samples, which is not that big, we need larger data set to observe clustering performance in terms of iteration count and perplexities.

As a result, we can conclude that t-SNE performs very well when we compare with the Sammon’s Mapping since same digits are close and separate from different digits. Also, t-SNE code runs faster than Sammon’s Mapping since it requires less computational power. I tried modifying additional settings like learning rate, early exaggeration to better see the points and establish a better separation between the clusters, but the results did not improve significantly.

Used Tools

1- loadmat() function from scipy.io module to load the dataset (https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.loadmat.html)
2- import matplotlib.pyplot as plot : This module used to plot. (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html)
3- import numpy as np (https://numpy.org/)
4- from sklearn.decomposition import PCA : Applying PCA. (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)
5- from sklearn.discriminant_analysis import LinearDiscriminantAnalysis : Applying LDA. (https://scikit-learn.org/stable/modules/generated/ sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html)
6- from sklearn.model_selection import train_test_split : creating test and train data. (https://scikit-learn.org/stable/modules/generated/ sklearn.model_selection.train_test_split.html)
7- from sklearn.naive_bayes import GaussianNB : applying Gaussian Naive Bayes (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html)
8- from sklearn.manifold import TSNE : Applying TSNE. (https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html)
9- from sammon import sammon as SammonMapping : taken from Github repo for Applying SammonsMapping. (https://github.com/tompollard/sammon)

References

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825–2830. https://scikit-learn.org/stable/

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in Science &amp; Engineering, 9(3), 90–95. https://matplotlib.org/

Pollard, T. (2014). Sammon mapping in Python. https://github.com/tompollard/sammon