Hakan Gulcu
21702275
GE461-Project 2
10.04.2022

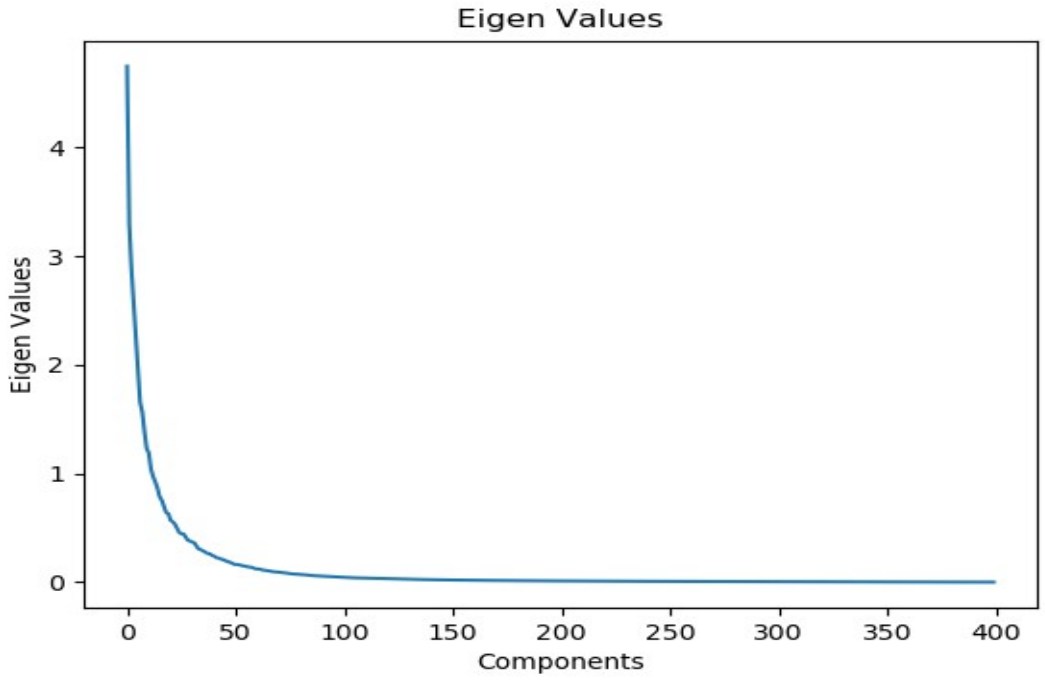
HANDWRITTEN DIGIT RECOGNITION SYSTEM

Question 1)

In this question, I have mostly used sklearn modules which is provided by Scikit-learn library. It has strong, simple and efficient data analysis tools. Also, matlabplot library is used to visualize data and scipy.io is used to load .mat dataset. I splitted data into half by using train_test_split function of sklearn.model_selection module. I used PCA class of sklearn.decomposition module to achieve PCA, GaussianNB class of sklearn.naive_bayes module to achieve Gaussian and metrics module to find accuracy scores. Lastly, I had to store result to make plots and I have used numpy module which is quite strong and widely-known array module.

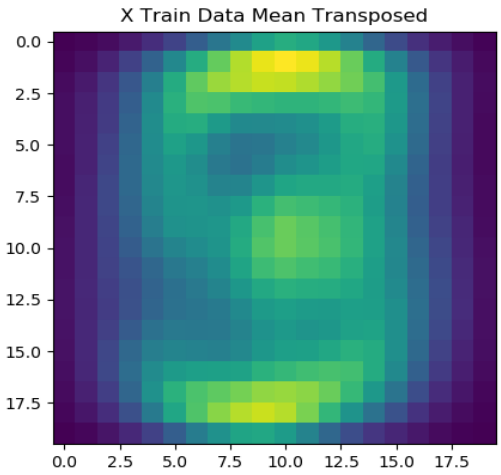
Question 1.1)

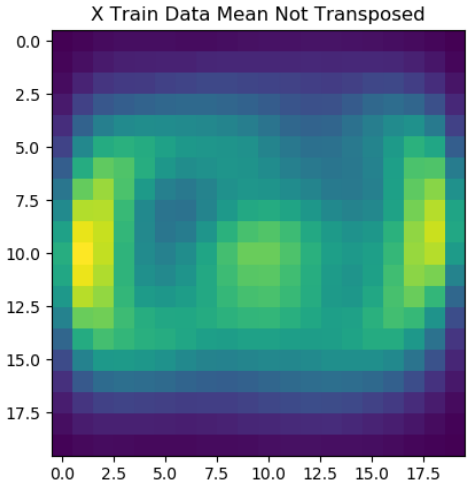
We have to find eigenvectors and plot them in descending order. I have 400 features in data set and after using PCA function, I can find out eigenvectors by using explained_variance_ attribute. It automatically sorts them in descending order. Finally, I plotted them by matlabplot. I would choose 60 components by just looking this plot. There are still some decrease until 60 but after that it becomes straight. After 60 components, it does not increase the variance. Therefore, approximately 60 components should be chosen.



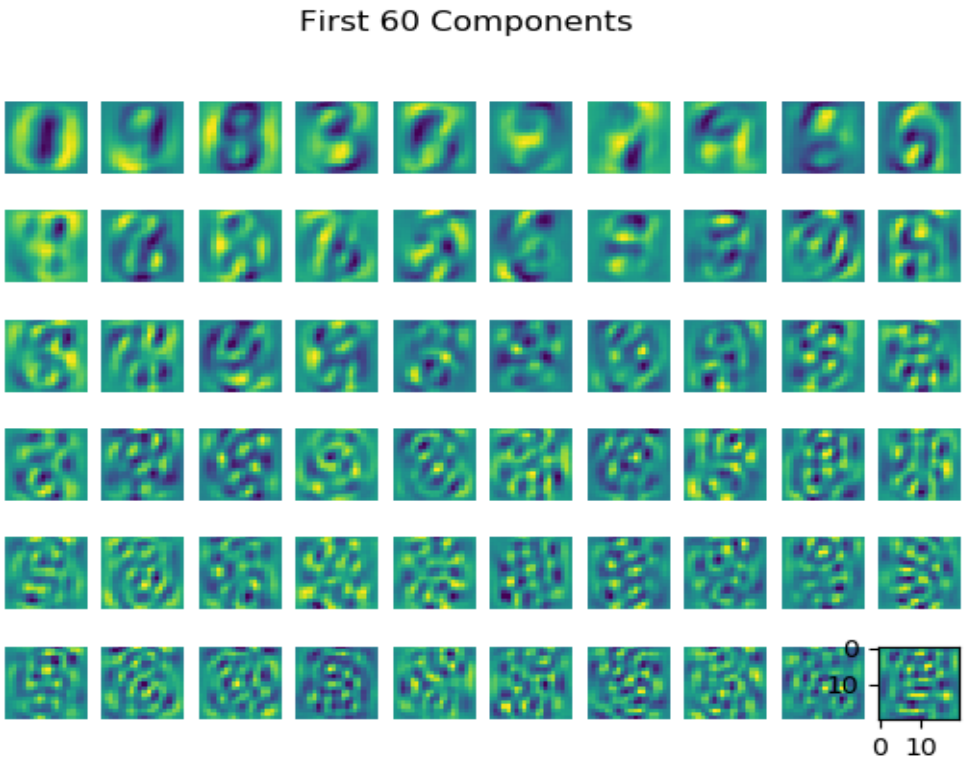
Question 1.2)

Firstly, I have used mean_ attribute of PCA class of sklearn module to find mean of data. After that, I transposed the data with .T and plotted the result. The most distinctive feature of the mean plot is that it has a circular shape. We can classify digits as either circular or sharp-shaped. For example, 1 and 7 are sharp, while 3 and 8 are circular. When you look at the mean, the circular shapes draw attention. Also, we can say that the parts containing yellow or light blue/green are common points between all digits. For example, we can say that the dark yellow at the top is very intense in the data. The digits we have are 0,1,2,3,4,5,6,7,8 and 9. I expected the mean to have the highest variance in circular shapes since all except 1,4 and 7 have circular shapes and it turned out to be correct.





First 60 components also displayed. I already mentioned why I chose 60 at previous part. All the components that are more suitable to guess what it is still have circular shapes in the same way. This also supports the things I mentioned above.

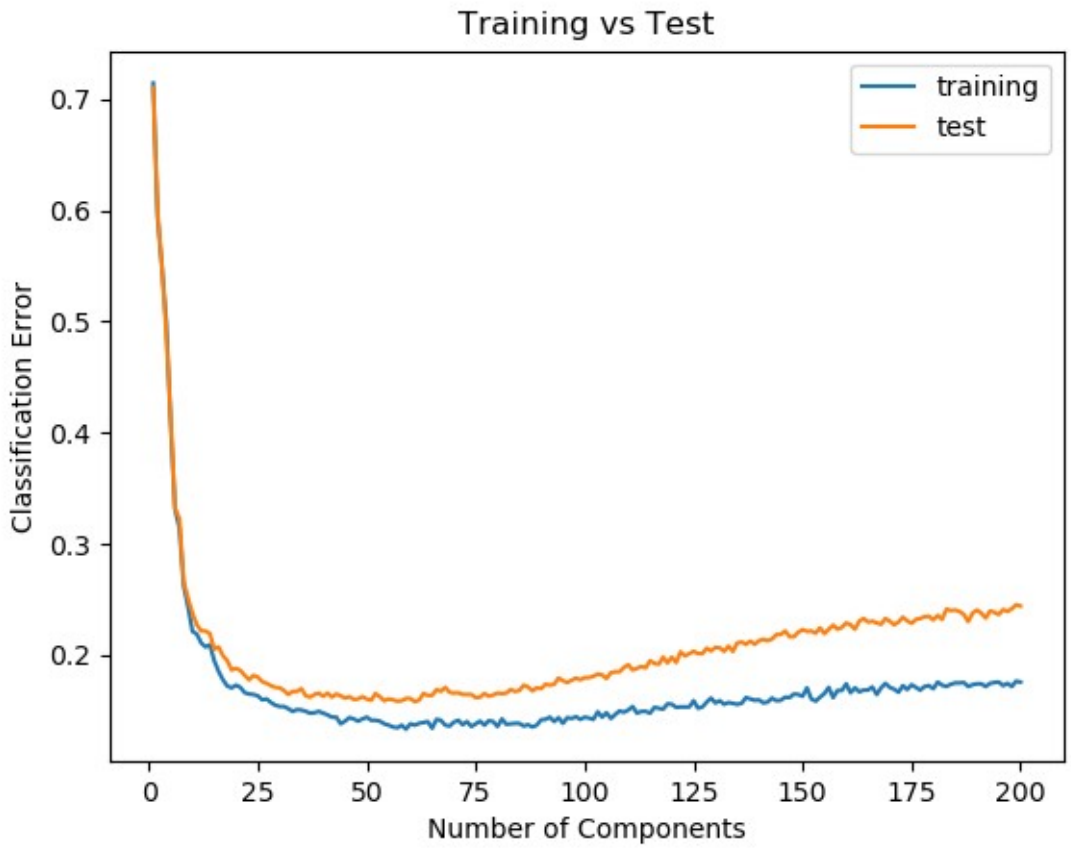


Question 1.3)

Firstly, I chose 200 subspace dimensions because it is stated that the more is better and the most is 200. For each iteration, new PCA with n components is created and fitted into train data with fit function of PCA class. After that, by using transform function which is a function of PCA class, train and test data are projected into new variables. To train a Gaussian classfier, sklearn.naive_bayes module has GaussianNB class and I used to implement my code. GaussianNB class has a function called predict() perform classification on test data. Finally, we need to find out classification errors and I could not find any function to calculate error directly. I find out sklearn has metrics module and it has accuracy_score function. By subtracting the accuracy from 1, I calculated classification errors and saved them into np arrays. I will present the plots created with those values.

Question 1.4)

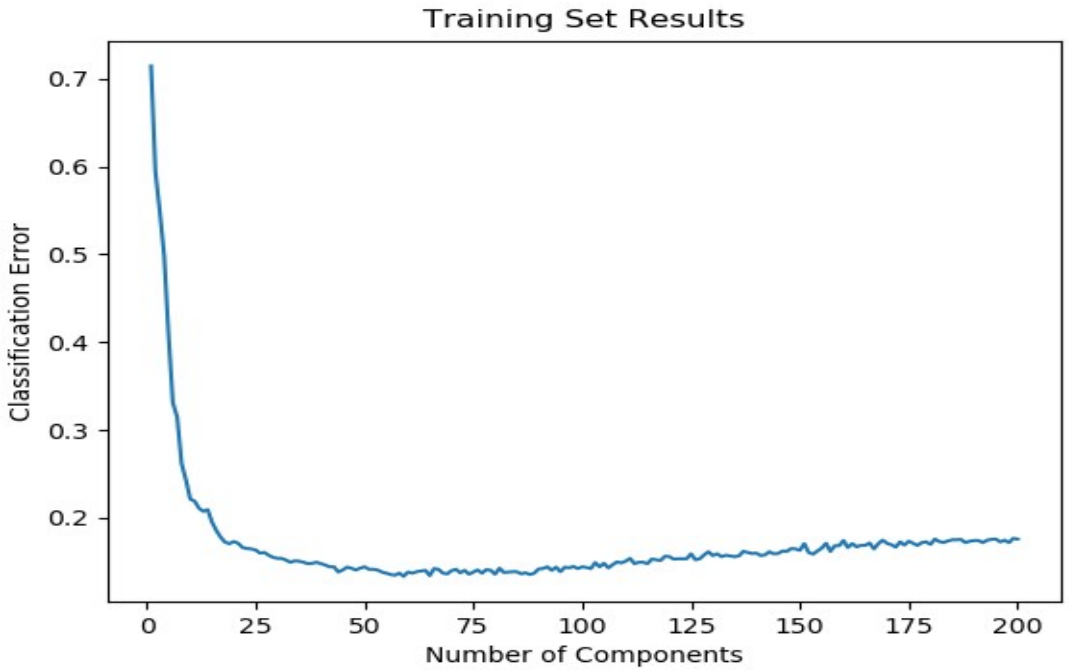
When we look at the comparison plot, it is clear that train data and test data shows almost equal classification errors where component counts between 2 and 10. After this point, test data has more classification error than training data at any number of

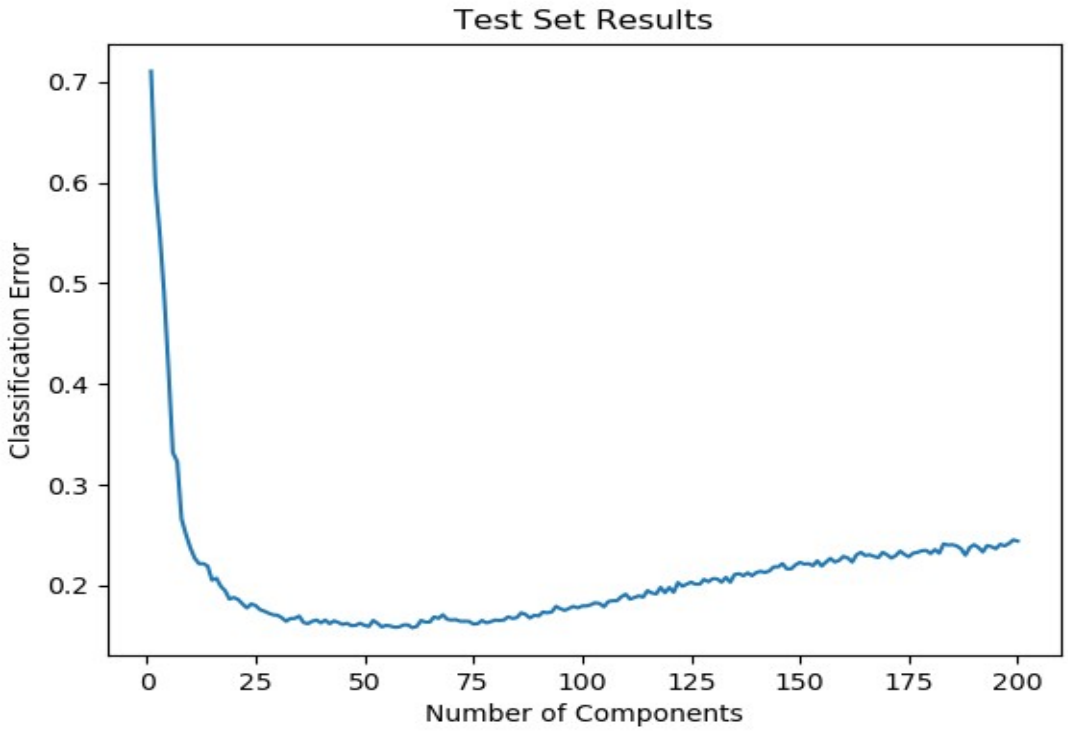


components. It is an expected result because model already used the train data and knows better it than test data. Even test data has worse results than train data, there are no significant differences.

If we look training and test data separately, there are significant decrease where number of components are between 2 and 10. This might be because of learning rate is decreased a lot at that points. After that, both are decreasing slightly till number of

components exceed 65 and increasing after number of components exceed 65. The increasing is understandable because when number of components increases, we may add unnecessary features and it will make accuracy scores lower and error rates higher.



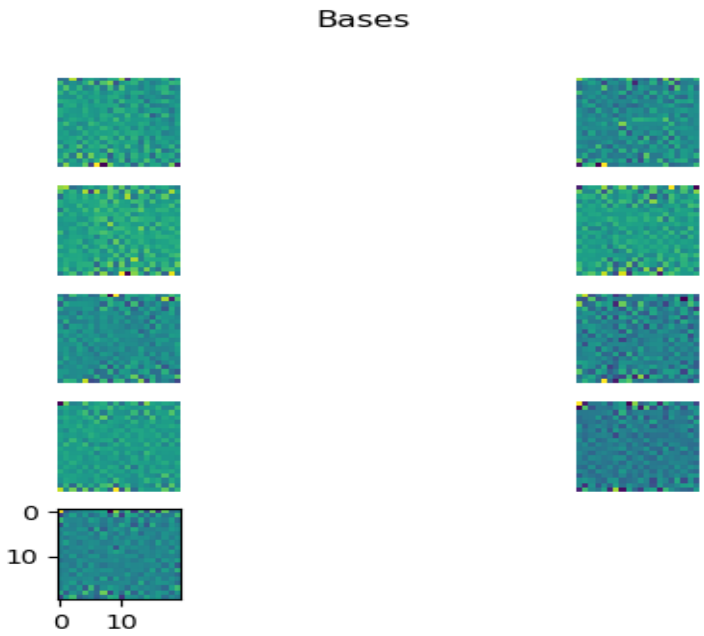


Question 2)

In this question, I have mostly used sklearn modules which is provided by Scikit-learn library. It has strong, simple and efficient data analysis tools. Also, matlabplot library is used to visualize data and scipy.io is used to load .mat data set. I splitted data into half by using train_test_split function of sklearn.model_selection module. I used LinearDiscriminantAnalysis class of sklearn.discriminant_analysis module to achieve LDA, GaussianNB class of sklearn.naive_bayes module to achieve Gaussian and metrics module to find accuracy scores. Lastly, I had to store result to make plots and I have used numpy module which is quite strong and widely-known array module.

Question 2.1)

In this question, I have 9 LDA bases because it is stated we should choose between 1 and 9. sklearn.discriminant_analysis has LinearDicriminantAnalysis class to achieve LDA. After using LinearDicriminantAnalysis, I used scalings_ attribute and plot them by reshaping them 20x20.



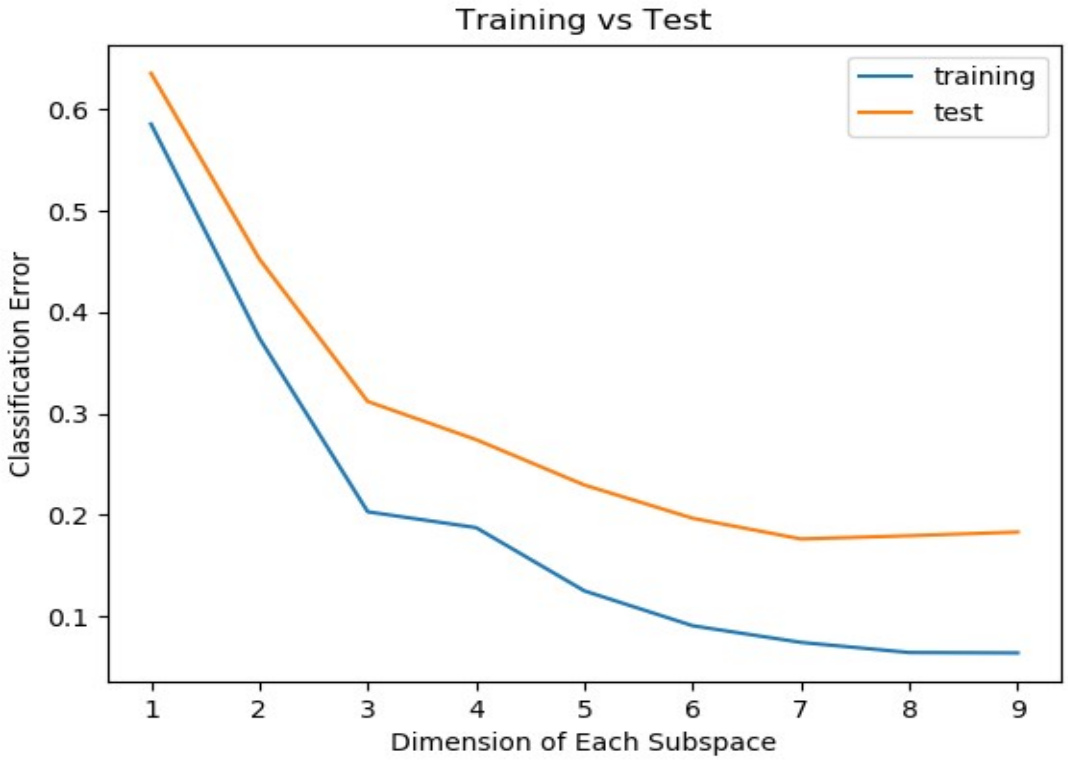
I was expecting that LDA result would be different than PCA results. The reason is that they have different techniques but they have similar purposes. PCA is an Unsupervised Dimensionality reduction technique and it ignores the class labels and aim to create maximum variation in data set. Against this, LDA is Supervised Dimensionality reduction technique and it cares about class labels and can make classification. LDA try to find a feature subspace which maximizes separability. Therefore, when we look at LDA bases, we cannot understand the digits where we can identify easily in PCA. The bases we see are basically some vectors where separability is maximum.

Question 2.2)

In this question the thing I do is exactly same as the Question 1.3. The only difference is using LinearDiscriminantAnalysis class of sklearn.discriminant_analysis module to achieve LDA and choosing dimension as 9. For each iteration, new LDA with n components is created and fitted into train data with fit function of LDA class. After that, by using transform function which is a function of LDA class, train and test data are projected into new variables. To train a Gaussian classifier, sklearn.naive_bayes module has GaussianNB class and I used to implement my code. GaussianNB class has a function called predict() perform classification on test data. Finally, we need to find out classification errors and I could not find any function to calculate error directly. I find out sklearn has metrics module and it has accuracy_score function. By subtracting the accuracy from 1, I calculated classification errors and saved them into np arrays. I will present the plots created with those values.

Question 2.3)

Before make comparison, we need to define what the number of components refer in PCA. We use PCA to calculate projection of data set and to make dimension reduction. To achieve this, we select number of dimensions or principal components of the projection data as input and they refer same thing. Basically, if there are N principal components, then we can say that we have N dimensional data or N eigenvectors. Also, we should not forget that the working principles of LDA and PCA are not exactly the same. Therefore, it is normal for us to get different values from each other in classification errors.

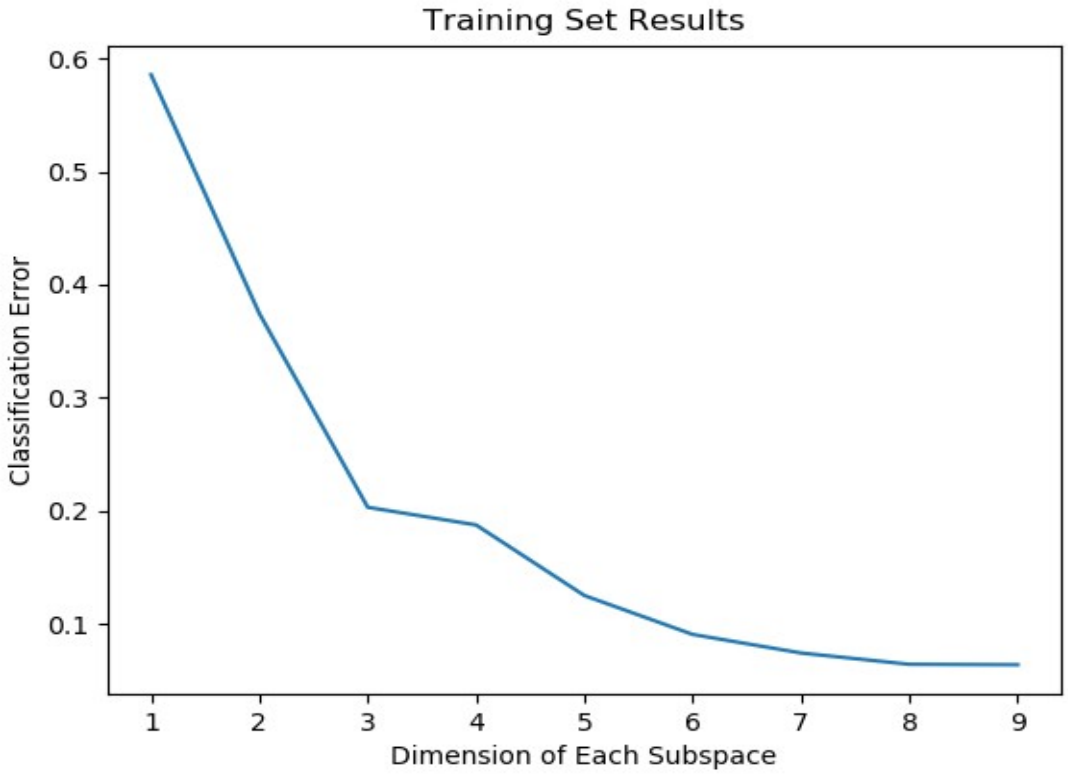


Firstly, when we look at the LDA results, error rate continuously decreasing while dimension of each subspace increasing and after some point it remains same. Also, in PCA, while dimension is increasing error rates are decreasing at some point and increasing after that, albeit a little.

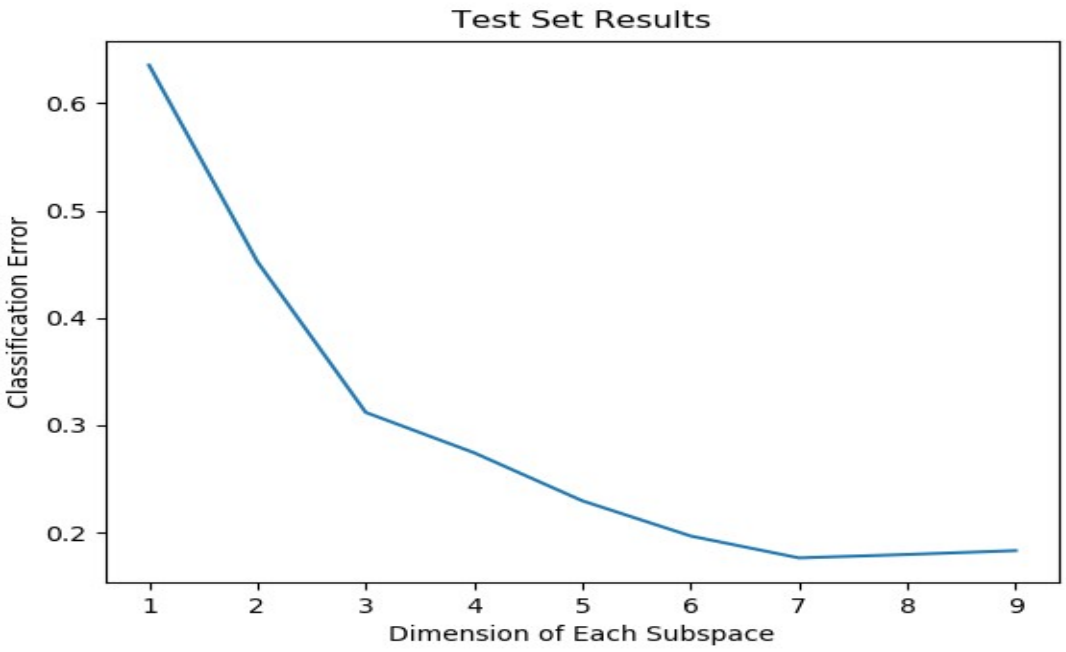
Secondly, at the beginning of PCA there is more classification error than beginning of LDA. After dimension of subspace for PCA reaches 20 and dimension of subspace for LDA reaches 5 in training data, PCA and LDA start to have close error rates. However, it is clear that LDA components have less classification errors on the training data. When, we look at test data, things are very similar as training data and LDA still better a bit. For both PCA and LDA, as expected, training data has less classification error

Thirdly, for PCA, training and test data has equal error rates at the beginning and after number of components or dimension pass 10, test data has more classification error rates than training data. However in LDA, at any point test data has more classification error than test data. It is already expected that training data has more accuracy in both cases.

Lastly, PCA has some increasing and decreasing times while number of components or dimension is increasing. Even those are very small differences, LDA has continuously decreasing error rates while dimension of each subspace is increasing. Therefore, LDA gives us more reliable and orderly result.



For training set, there is a sharp decreasing in classification error between 1 and 3 dimension of each subspace and it decreasing slightly until 8, then remains same after 8.



For test set, there is a sharp decreasing in classification error between 1 and 3 dimension of each subspace and it decreasing slightly until 7, then remains almost same after 7.

QUESTION 3)

For both Sammon's Mapping and t-SNE, I will introduce the techniques and describe how I used them.

1) Sammon's Mapping:

Sammon's mapping is used to map high dimensional space in lower dimensions for visualization purposes in general and it is well known algorithm. However, I could not find any library or module implemented in Python and I used a GitHub repository for its implementation. The main reason of Sammon's Mapping is to minimize error function. However, projection has to be recomputed for each data item and because of this, the algorithm is slow. Even for 450 iteration, I had to wait for 40 minutes.

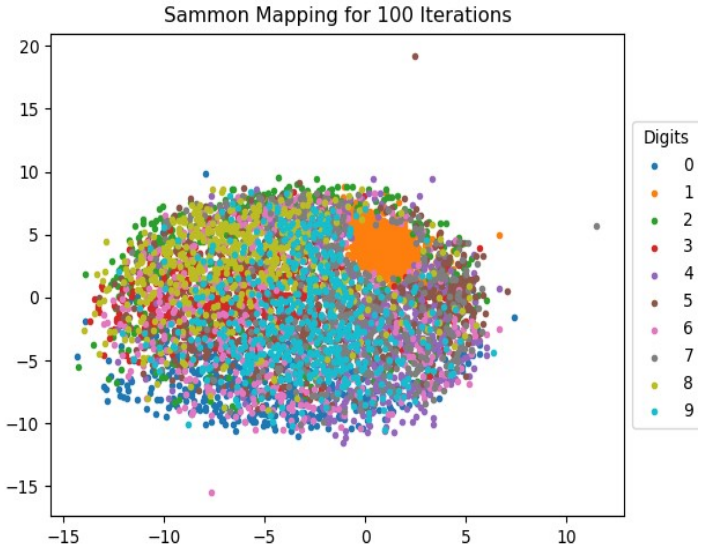
After getting result, I drew them with matplotlib.pyplot module. To make the experiment valid and see changes, I have used 100,150,250 and 400 iterations. I will first describe the function I used and how I used it.

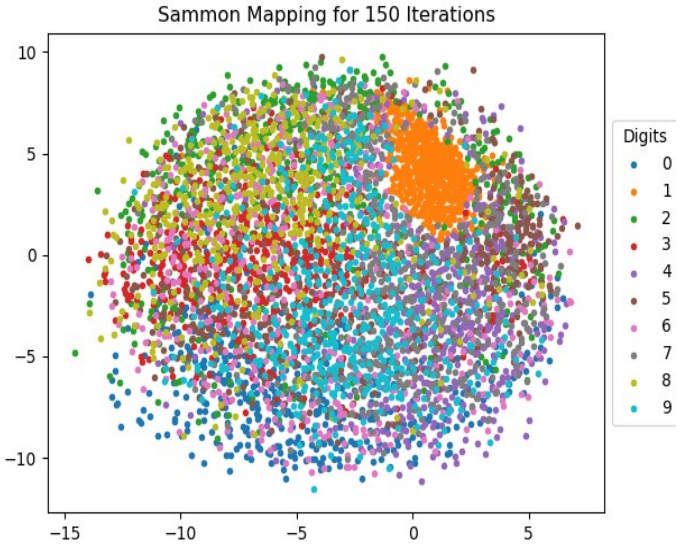
def sammon(x, n, display = 2, inputdist = 'raw', maxhalves = 20, maxiter = 500, tolfun = 1e-9, init = 'default')

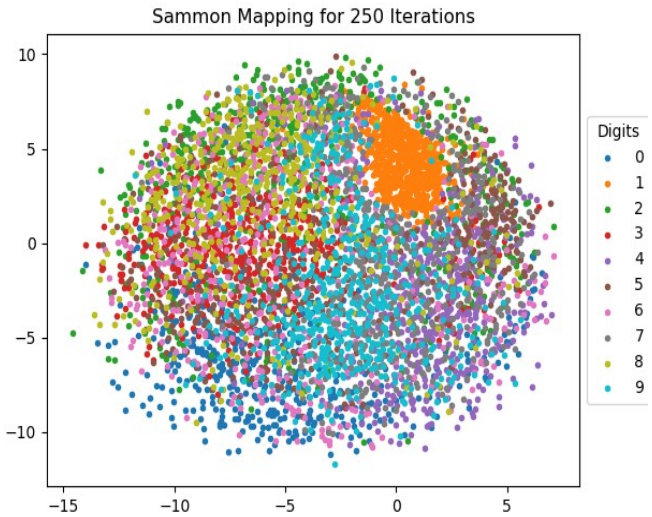
[x,E] = sammon(features, n = 2, maximeter = 100)

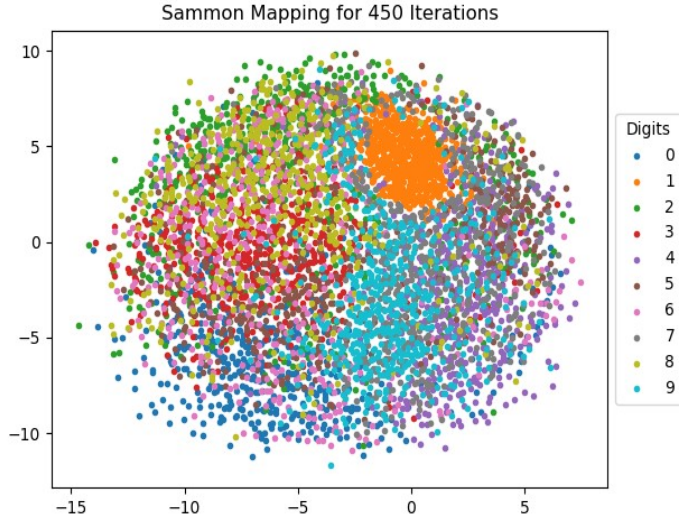
[x,E] also returns the value of the cost function in E.

Maximeter is the maximum number of iterations and the function does that much iteration and create results. I only change it and other arguments are default to make experiment valid. For example, by not changing init=’default’, I let it to use its own cmdscale.py code. The main reason I did not change default variables except maximeter is that the variables default are chosen to prevent algorithm crash.









As you can see from the plots I added, there is no significant change even I increased iteration count

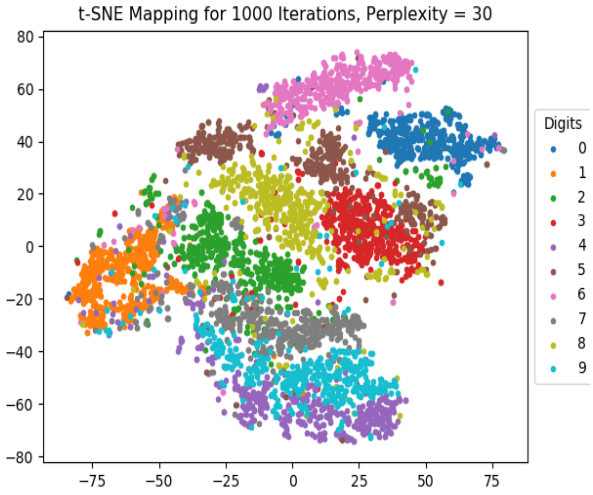
100 to 450 but there is point should be considered. Increasing the number of iterations brings the digits closer and closer to the same digits. The most obvious example of this may be that the digits that appear outside in 100 iterations get very close to the others when we increase the number of iterations. Also, let's look at the areas where the colors are intense. While the blue dots (digit 9) are not very close to each other in 100 iterations, they seem much closer when it reaches 450 iterations. Likewise, while we see that the red dots (digit 3) are not obvious at all at the beginning, we can say that they are more prominent in 450 iterations as they get closer to each other as the number of iterations increases.

In order to better understand these results, it may be useful to go up to higher iteration amounts. However, I was not able to do this in experience, as I did not have the necessary system requirements. Despite the low iteration numbers, I can say that the algorithm works just fine. I think the visualizations are also successful, despite the small iteration numbers, it can help us understand the algorithm just by looking at the plots.

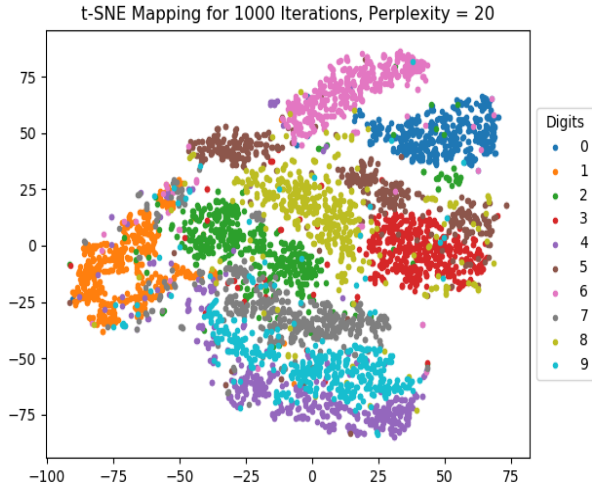
2) t-SNE (t-distributed stochastic neighbor embedding)

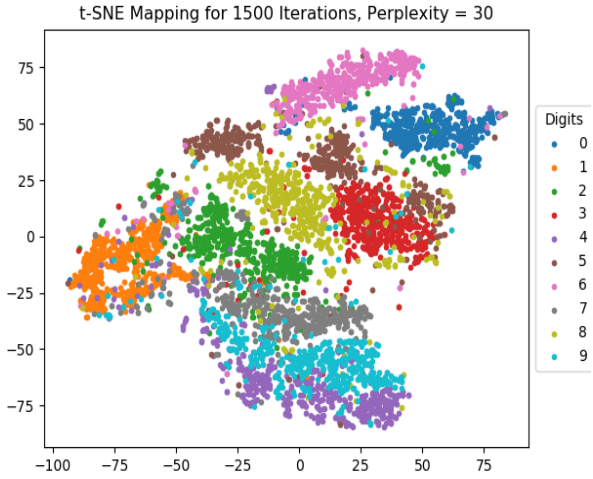
t-SNE is an unsupervised technique to visualize high dimensional data. It is an well-known algorithm and have some similarity with both Sammon's Mapping and PCA. t-SNE is also using cost function to optimize similarity measure. Like Sammon's Mapping, it approximates the same digits (same data), and tries to have equal amounts of neighbors around each data point. It is more likely to put different data points further away from each other.

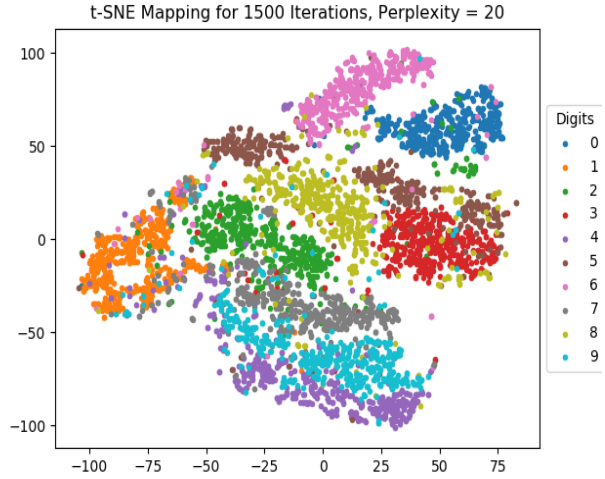
PERPLEXITY = 30

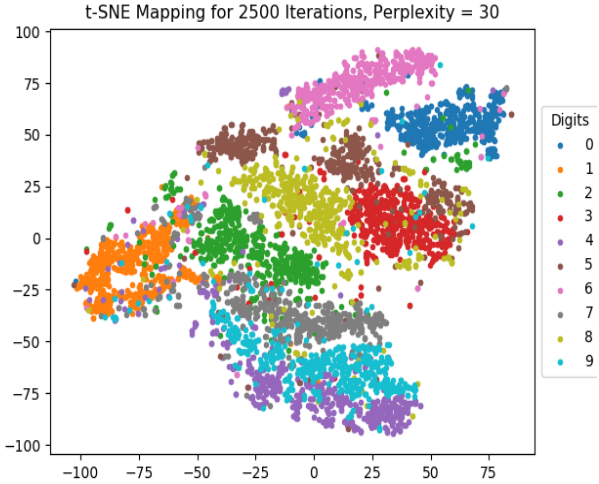


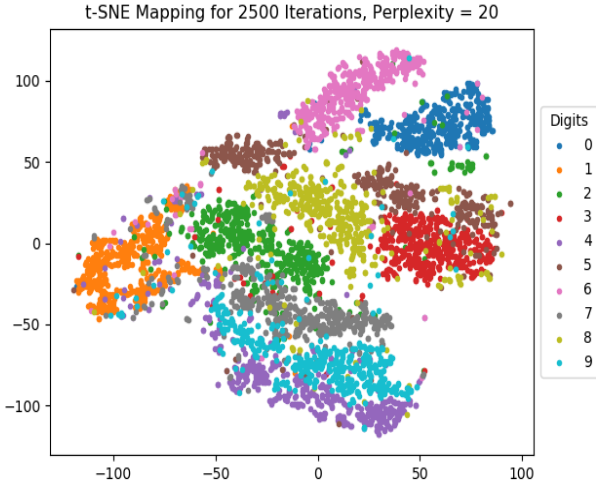
PERPLEXITY = 20

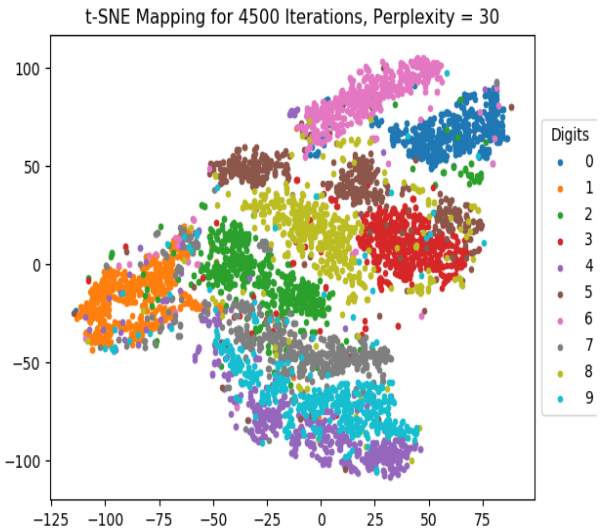


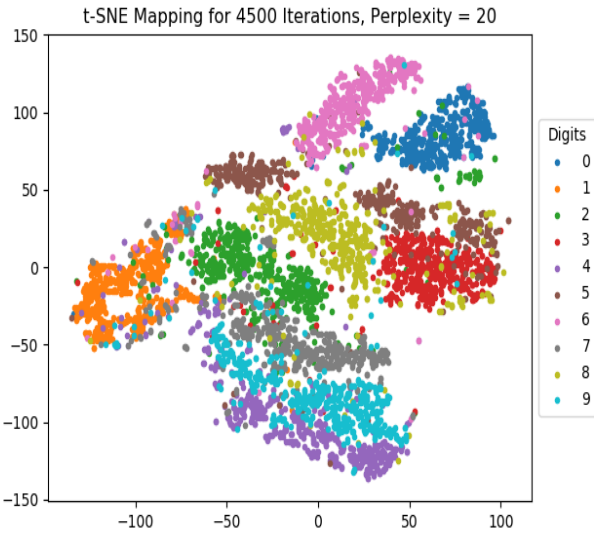

t-SNE Mapping for 1500 Iterations, Perplexity = 30
Digits
0
1
2
3
4
5
6
7
8
9


t-SNE Mapping for 1500 Iterations, Perplexity = 20
Digits
0
1
2
3
4
5
6
7
8
9


t-SNE Mapping for 2500 Iterations, Perplexity = 30
Digits
0
1
2
3
4
5
6
7
8
9


t-SNE Mapping for 2500 Iterations, Perplexity = 20
Digits
0
1
2
3
4
5
6
7
8
9


t-SNE Mapping for 4500 Iterations, Perplexity = 30
Digits
0
1
2
3
4
5
6
7
8
9


t-SNE Mapping for 4500 Iterations, Perplexity = 20
Digits
0
1
2
3
4
5
6
7
8
9

To compare plots I want to introduce the function I used. To achieve t-SNE sklearn library has TSNE class of sklearn.manifold module. By using this library t-SNE can be made easily.

class sklearn.manifold.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=None, method='barnes_hut', angle=0.5, n_jobs=None, square_distances='legacy')

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')

n_iter is the iteration count and I have changed it to 1000, 1500, 2500 and 4000 to see different results. It should at least be 250. However, the result was not like as I expected. There are almost no change even for 1000 and 4000 iteration. Therefore, I read the documentation to define how to make better experiment. After that, I found out changing lots of arguments will not help me. Therefore all variables except n_iter, perplexity and random_state are default.

random_state is an important argument to keep experiment valid. Even different random_state variables creates different plots, keeping it same helps me to see difference between different perplexities because it reproduces the t-SNE with same parameters if you do not change it. However, having different random_states may be useful when we have larger data sets because it directly affects the plot.

perplexities is the most important argument to make experiment. As can be seen from plots, having less perplexity make data points closer to its similar data points. Perplexity refers to number of nearest neighbors. If I have larger a larger data set I would choose it bigger. It is also affects the result of experiment.

TSNE algorithm does good job in total. When I compare it with Sammon’s Mapping it has some advantages. Firstly, when we look both plots of t-SNE and Sammon’s Mapping, t-SNE has more clear and understandable plots. Also, I had to wait a lot for Sammon’s Mapping because it is an iterative algorithm but t-SNE needs less computation power and does not take lots of time.

TOOLS USED :

1) import numpy as np :
    This module used to create arrays to store data.
    (https://numpy.org/)

2) import matplotlib.pyplot as plot :
    This module used to plotting.
    (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html)

3) import scipy.io as loader :
    This module used to load .mat data by loadmat()
    (https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.loadmat.html)

4) from sklearn.discriminant_analysis import LinearDiscriminantAnalysis :
    This module used to apply LDA.
(https://scikit-learn.org/stable/modules/generated/
sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html)

5) from sklearn.model_selection import train_test_split :
    This modules used to split data to half.
(https://scikit-learn.org/stable/modules/generated/
sklearn.model_selection.train_test_split.html)

6) from sklearn.naive_bayes import GaussianNB :
    This modules used to apply Gaussian Naive Bayes

(https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html)

7) from sklearn import metrics :
    This modules used to find accuracy score and find out classification error.
    (https://scikit-learn.org/stable/modules/model_evaluation.html)

8) from sklearn.decomposition import PCA :
    This module used to apply PCA.
    (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)

9) from sklearn.manifold import TSNE :
    This module used to apply TSNE.
    (https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html)

10) from sammon import sammon as SammonMapping :

    This is used to apply SammonsMapping and taken from Github repo, not a library.
    (https://github.com/tompollard/sammon)

Citiations

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

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., … Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585, 357–362. https://doi.org/10.1038/s41586-020-2649-2

Pedregosa, F., Varoquaux, Ga"el, Gramfort, A., Michel, V., Thirion, B., Grisel, O., … others. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825–2830.

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

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