Turgut Alp Edis

21702587

GE461-1

GE461 – Introduction to Data Science Project 2

Question 1)

In the question 1, I used principal component analysis (PCA) to calculate the lower dimensions of digit data. For PCA implementation part of this question, I used the manual implementation of PCA without using any external libraries except numpy from the formula for PCA. In order to load digit dataset, I used io module of scipy library which has the loadmat function to read the .mat file extensions. I used numpy to store the data in arrays. To split the data as training and testing, I used train_test_split function of model_selection module of sklearn library. For Gaussian classifier, I used naïve_bayes module of sklearn library. To calculate the classification errors, I used metrics class of sklearn library. Finally, I used pyplot module of matplotlib library.

Question 1.1)

As stated in the project document, the component size of PCA is 400. It means that there are 400 features left after the PCA is implemented. After the dimensionality reduction, I calculated the eigen values and eigen vectors with covariance matrix of the centered train data, which is the result of the difference between the data and its mean. After the eigen value and eigen vector calculations, I sorted eigen values in reverse order to obtain descending order eigen values. Then, I obtain the plot in figure 1.

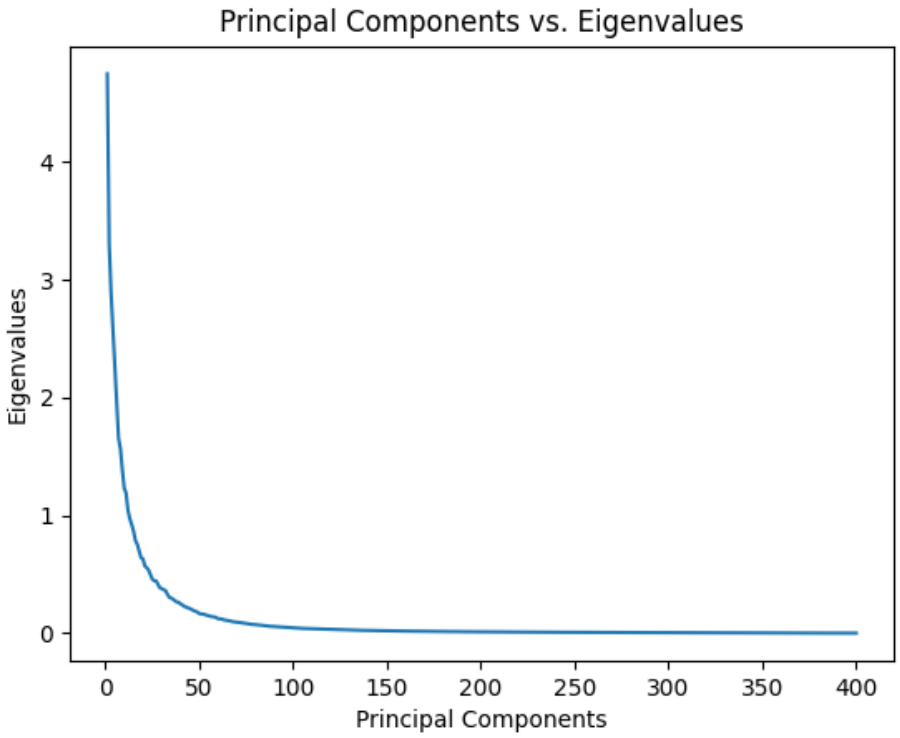

Figure 1. Principal Components vs Eigenvalues

From the plot in figure 1, I can assume that approximately 64 of the principal components can be chosen since approximately after this point, the plot becomes straighter due to decrease in the difference between principal components.

Question 1.2)

In this question, I calculated the sample mean of the training data set with using np.mean function. However, since the component size is 400, I reshaped the mean data to 20 by 20 array, then transposed the data since the mean of the data gives the result as if the image data turned the 90 degrees. Therefore, for better observation, I transposed the mean data. The mean of the data can be seen in figure 2 and can be seen as gray, for better observation, in figure 3.

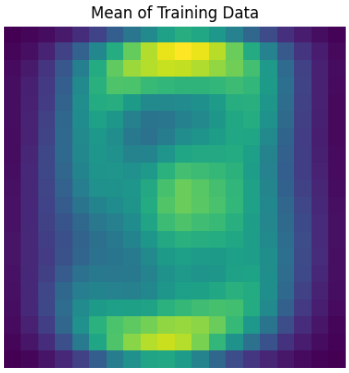

Figure 2. The Mean of Training Data

Figure 3. The Mean of Training Data in Gray

From the images in figure 2 and 3, I could say that the mean has more circular points as in digits 3,6,8,9,0 rather than sharp point as in digits 1, 7, 4. Therefore, it can be said that the digits have more circular points than sharp points. In addition, from figure 2, it can be said that the digits have the lines commonly in middle top and middle bottom since the yellow means that these points are more frequently used by the data.

I expected that the mean has more circular points rather than sharp points because only 1,4,5 and 7 have sharp points while 0,2,3,5,6,8,9 have circular points. Thus, my expectations are correct.

In the question 1.1, I explained the reason why 64 principal components are chosen. In figure 4, the 64 principal components are displayed.

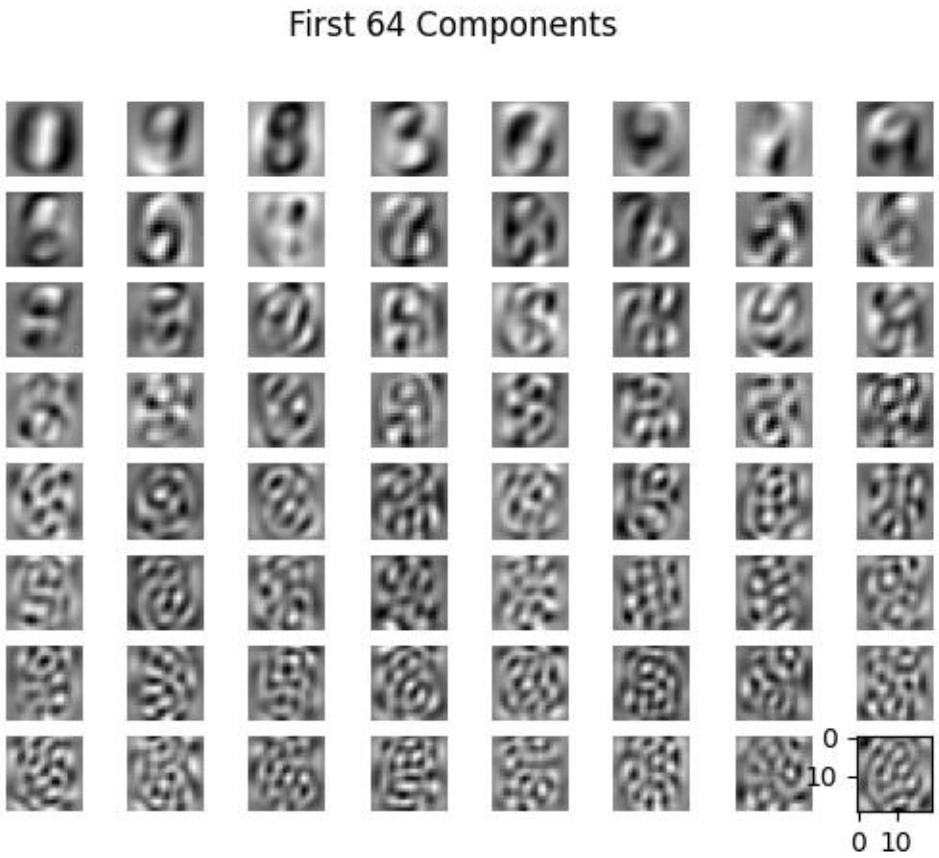

Figure 4. First 64 Principal Components

From the figure 4, I can state that my expectations, as I mentioned in earlier, are correct since each of these components has circular shape.

Question 1.3)

As stated in the project document, at least 20 different subdimensions can be chosen between 1 and 200. So, I chose first 150 subdimensions because it is stated that the more the better. Then, I calculated the PCA for each subdimensions for training data. Both training and test data sets are transformed with dot product of centered data sets and PCA. After these calculations, I trained the Gaussian classifier using training data with using GaussianNB class of naïve_bayes module of sklearn library, then I tried to predict both training data and test data using predict method of GaussianNB class, calculated the classification errors for each data set using metrics class of sklearn library and stored the errors in numpy arrays to plot the error graphs that is wanted in question 1.4.

Question 1.4)

I plotted the train and test classification errors using matplotlib library in the same plot space initially to observe the difference between them more easily as shown in figure 5.

Figure 5. Training and Test Errors for 150 components

From the figure 5, I can conclude that the difference between training and test errors increases after approximately 20 components. However, the difference between them is not quite large. As an expected result, the error for training is less than test error because training data is used for model the classifier.

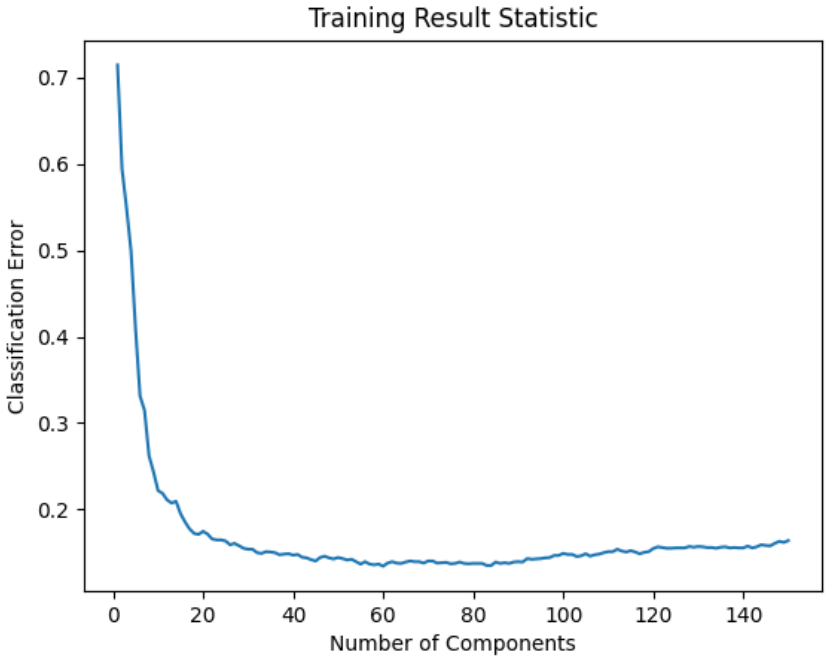

Figure 6. Training Error Graph

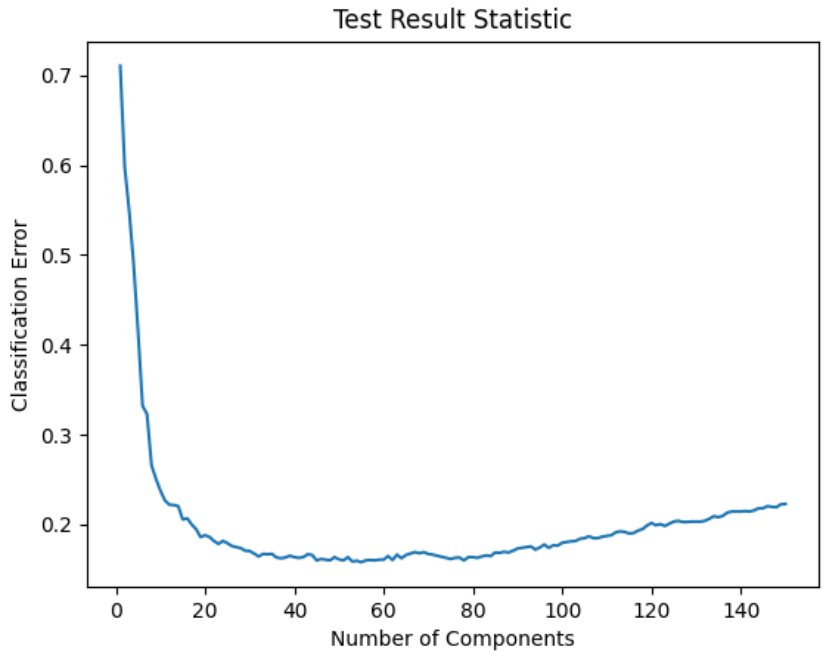

Figure 7. Test Error Graph

From the figures 6 and 7, I can say that there is a significant decrease between approximately 1 and 10. The reason of this significant decrease could be the decrease in learning rate between first 10 components. After some components, approximately after 75 components, the error increases in both graphs since after around 75 components, each component, in other words features, is not necessary for the PCA. Thus, each unnecessary components cause decrease in accuracy and increase in error rate.

Question 2)

To implement Fisher linear discriminant analysis (LDA), I used LinearDiscriminantAnalysis class of discriminant_analysis module of sklearn library. In order to load digit dataset, I used io module of scipy library which has the loadmat function to read the .mat file extensions. I used numpy to store the data in arrays. To split the data as training and testing, I used train_test_split function of model_selection module of sklearn library. For Gaussian classifier, I used naïve_bayes module of sklearn library. To calculate the classification errors, I used metrics class of sklearn library. Finally, I used pyplot module of matplotlib library.

Question 2.1)

In this question, I used LinearDiscriminantAnalysis (LDA) class of discriminant_analysis module of sklearn library to implement LDA. As it is stated in project document, I can only use maximum 10 components for LDA. So, I chose to 10 components because of this reason. Then, after I calculated the scaling of the LDA with using scalings_ method of LDA class, I plotted the bases with using pyplot module of matplotlib library as it can be seen in figure 8.

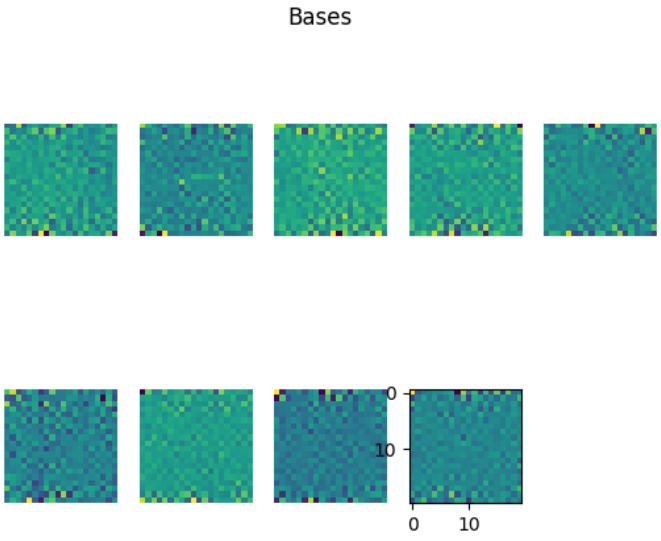

Figure 8. The bases calculated with using LDA class

Before this result, I searched about the differences between PCA and LDA. In PCA, the technique of the reduction is not supervised, so the images from the PCA looks familiar to us since its aim is to create maximum variate data for dimensionality reduction, while LDA uses supervised dimensionality reduction, so its images could not look familiar to us since its aim is to create feature subspace, so it can maximize the possibility of separation of data. Thus, I expected that the images of bases of LDA does not look familiar as in principal components of PCA.

Question 2.2)

In this question, I implemented the Gaussian classifier with using same algorithm in question 1 part 3. Initially, I calculated the LDA for each component between 1 and 9 with using LDA class of discriminant_analysis module of sklearn library, then, I transformed both the training and test data sets with using transform method of LDA class. After the transformation, I initialized the Gaussian classifier with using GaussianNB class of naïve_bayes module of sklearn library. After the initialization of Gaussian classifier, I fitted the transformed training data with using fit method of GaussianNB class to use transformed training data as learning model for classifier. After fitting, I tried to predict both training data and test data using predict method of GaussianNB class, calculated the classification errors for each data set using metrics class of sklearn library and stored the errors in numpy arrays to plot the error graphs as wanted in question 2 part 3.

Question 2.3)

In this question, I used the error data from the previous part of the question (question 2.2) to plot the classification error of each dimension as shown in figure 9 and figure 10.

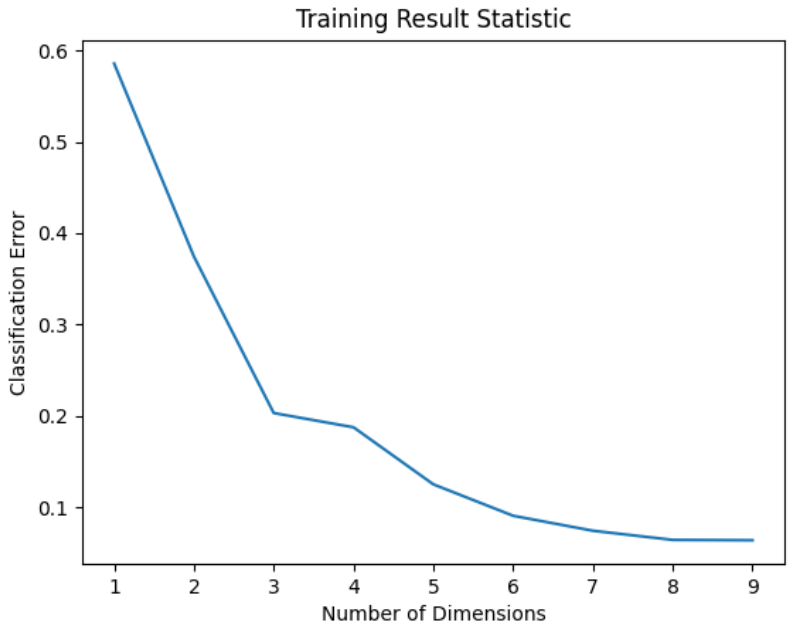

Figure 9. Classification Error for Training Data in LDA

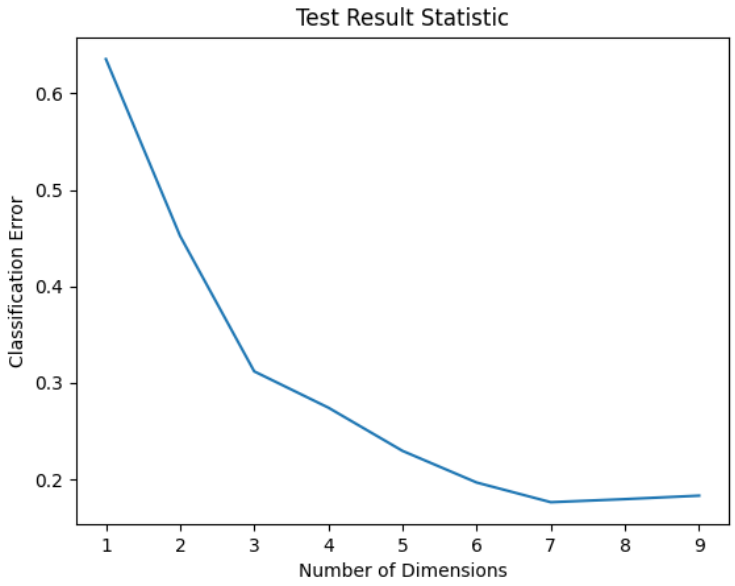

Figure 10. Classification Error for Test Data in LDA

From the figure 9 and figure 10, when the dimensions increase, the error decreases until it reaches 7 dimensions for test error graph (figure 10), but the error decreases in train error graph until it reaches 8 dimensions (figure 9).

In LDA, the increase in number of subdimensions reduces the classification error until 8 dimensions. After this point, it remains the same. The reason is that after some point, dimensionality is not necessary to predict the data just as in PCA where after some principal components, the error rate increases due to unnecessary components.

Before the comparison of LDA and PCA results, it needs to be understood that their methods are different from each other, so it is normal to get different graphs for classification error from these methods. In PCA, principal components are used to create specified size of eigen vectors of dimensional data to reduce dimensionality of data, while in LDA, maximum 10 dimensions are used to reduce the dimensionality of data.

When I compare the results, it is always true that train data has less error than test since train is used to model the classifier. Initially, the beginning of graphs is different. In PCA, the error rates are same for training and test data in the beginning but in LDA, it can be clearly inferred that the error rate of test data is more than train data. Secondly, the error rate is lower in LDA than it is in PCA. Finally, error of LDA is always decreasing while error of PCA is increasing at some point. Therefore, it makes LDA more reliable dimensionality reduction method than PCA.

Question 3)

For both Sammon's Mapping and t-SNE, I used external libraries to implement them. For Sammon's Mapping, since I could not find any libraries or packages in Python, I used Github repository to implement the Sammon's Mapping. For t-SNE, I used TSNE library of manifold module of sklearn library. I used io module of scipy library which has the loadmat function to read the .mat file extensions and used pyplot module of matplotlib library to plot the patterns as stated in project document.

Sammon's Mapping

Sammon's Mapping is used to map high-dimensional data sets into two dimensions for visualization. Sammon's Mapping aims to reduce the error in dimensionality reduction. However, it is significantly slow algorithm since even for 100 iterations, I had to wait 10 minutes and for 400 iterations I had to wait approximately 1 hour. The function I used to implement Sammon's Mapping is described below.

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

In this function, x is used to hold data, n is used to hold dimension count and maxiter is used to hold the maximum iteration number. The other parameters are not essentially important for this project, so they can stay in default.

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

E holds the cost function.

The only change in this function call is the change in maxiter count to observe the points for different iterations. Also, in order to make appropriate tests, I changed the maximum number of iterations and plot the graphs with using pyplot module of matplotlib library as it can be seen in figure 11,12,13 and 14.

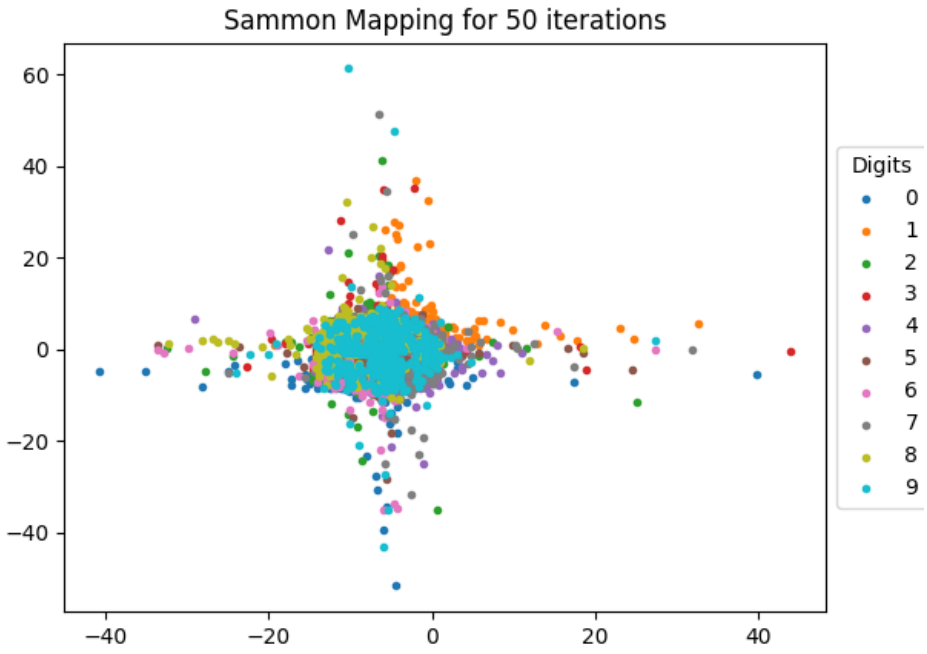

Figure 11. Sammon Mapping for 50 iterations

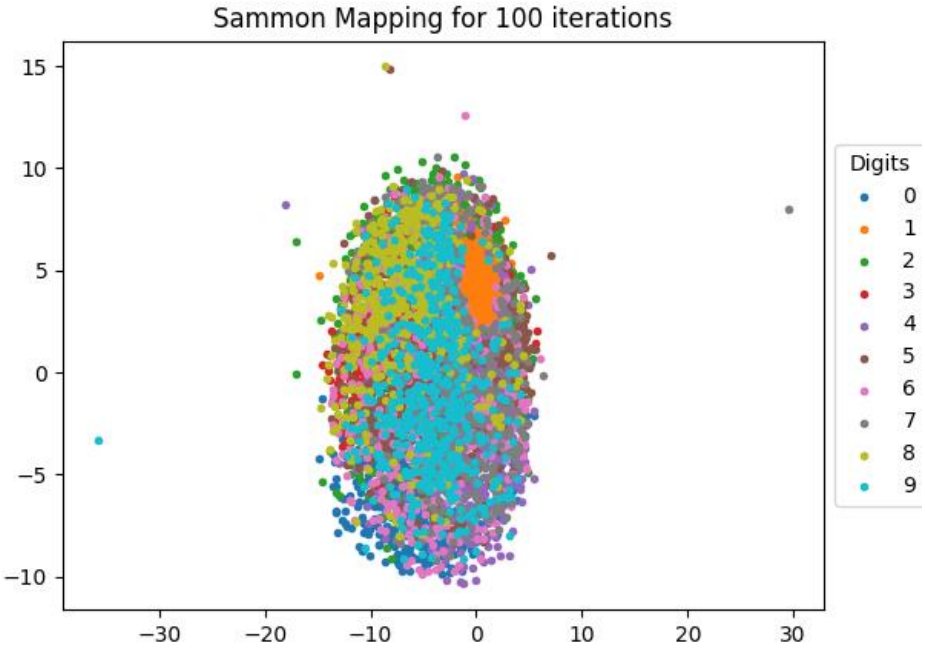

Figure 12. Sammon Mapping for 100 Iterations

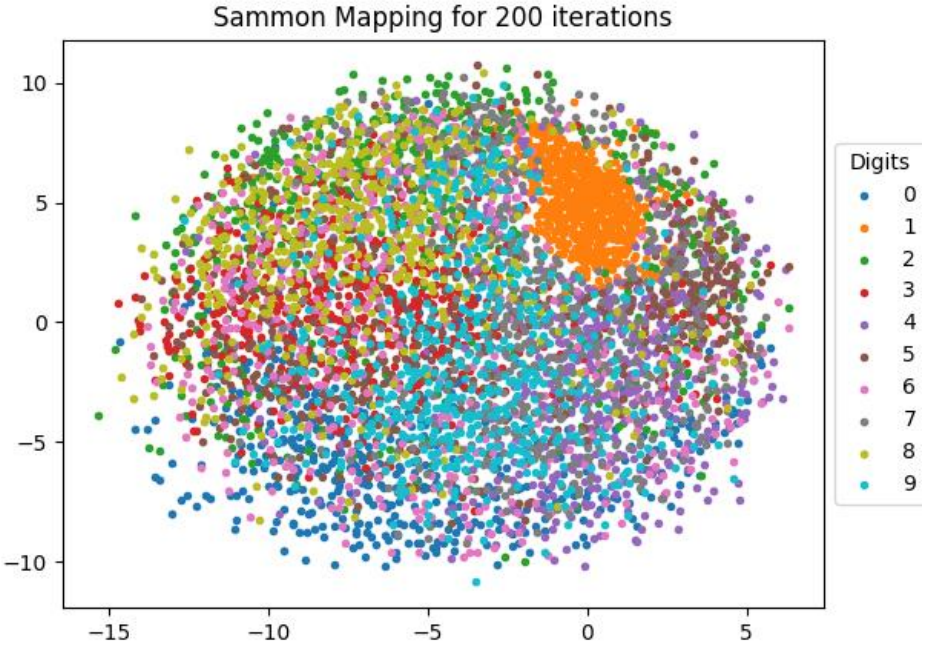

Figure 13. Sammon Mapping for 200 Iterations

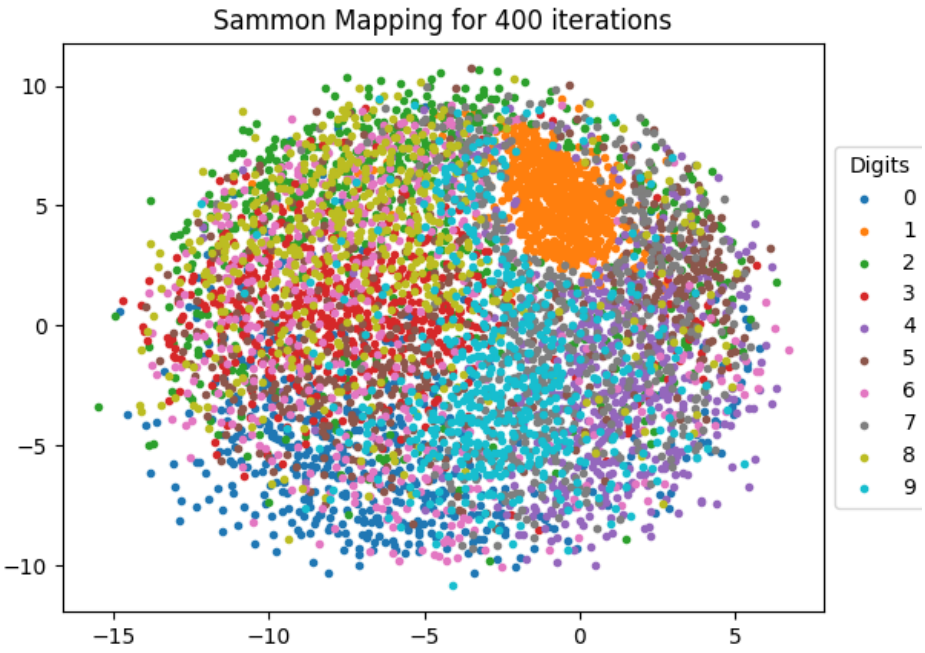

Figure 14. Sammon Mapping for 400 Iterations

From the figures, after 100 iterations, there are not much significant change of points in iterations. In 50 iterations, there are lots of extreme data in the plot. However, these points are closer to the center when the iteration increases. Also, it can be clearly seen that digit 9 (blue dots) are separated from each other when the iteration count increases. After 200 iterations, the points are not significantly changing.

In order to understand better, it is needed to do experiments with higher iteration numbers but its system requirements are high, so unfortunately, my pc did not allow me to higher experiments, though, it can be said that it works appropriately just by looking the figure 11,12,13 and 14.

t-SNE (t-distributed stochastic neighbor embedding)

t-SNE is an unsupervised mapping algorithm to visualize high dimensional data into lower dimensions. It is similar to Sammon's Mapping and PCA. It tries to optimize the neighbor count for each data point and tries to equalize the neighbor count with optimum cost like in Sammon's Mapping. In other words, it tries to group the data points. The function I used can be seen below.

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')

In this class, n_iter is the value of iteration for the mapping and I changed it to 1000,1500,2000 and 4000 to see various results and interpret the scatter plots. Initially, there are not any change between the mappings as can be seen in figure 15, 17, 19 and 21. The reason is that I also need to change perplexities to see the change because this value determines the distance between the neighbors. After the change of perplexities, I can see the change of scatter plots more clearly as it can be seen in figure 15,16; 17,18; 19,20 and 21,22. Also, cross observation meaning that to inspect different iteration count and different perplexity, helps me to see the effect of distance between the neighbors and iteration to the scatter plots.

Perplexity means to the number of nearest neighbors. When it becomes large, the data becomes more compress form. Therefore, the change of perplexity is important to observe significant changes in the data set. It can be observed in figure 15,16 or figure 17,18 or figure 19,20 or figure 21,22.

For this part, I used the TSNE function as follows.

tSNE = TSNE(n_components=2, perplexity=30.0, early_exaggeration=12.0, learning_rate='warn', n_iter=1500, 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')

For the experiment, I only changed the perplexity, random_state and n_iter. I changed the random state because if it stays in default which is None, it changed the data randomly. However, if it is set to 0, it does not change the data. Thus, I set that value as 0.

I changed the perplexity only to the values 20 and 30 for each iteration because it is enough to observe the change in data even with these 2 values.

The t-SNE Mapping for 1000, 1500, 2000, 4000 iterations for 20 and 30 perplexities can be seen in figure 15,16,17,18,19,20,21 and 22.

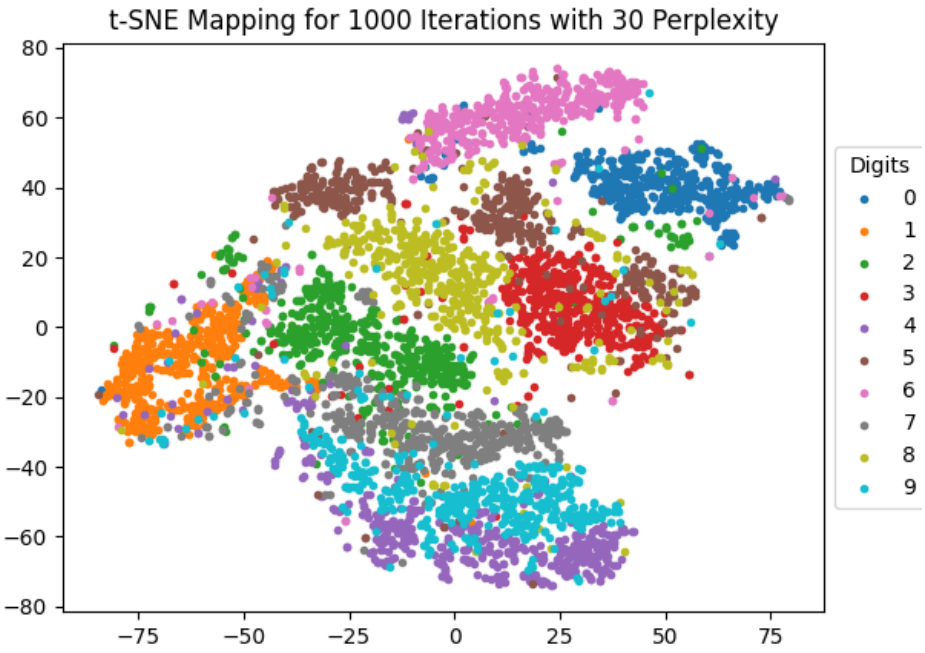

Figure 15. t-SNE Mapping for 1000 Iterations with 30 Perplexity

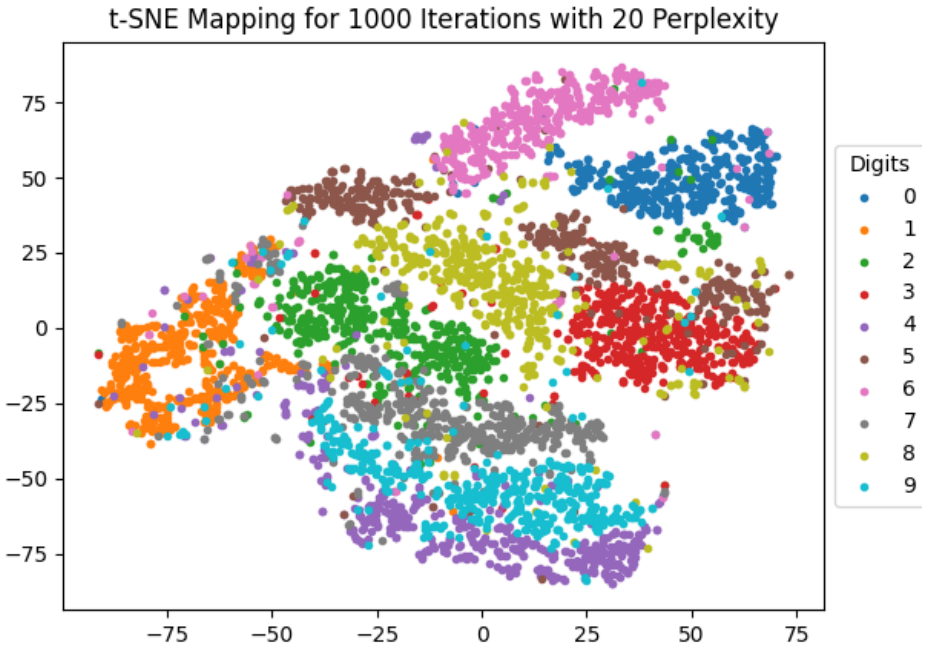

Figure 16. t-SNE Mapping for 1000 Iterations with 20 Perplexity

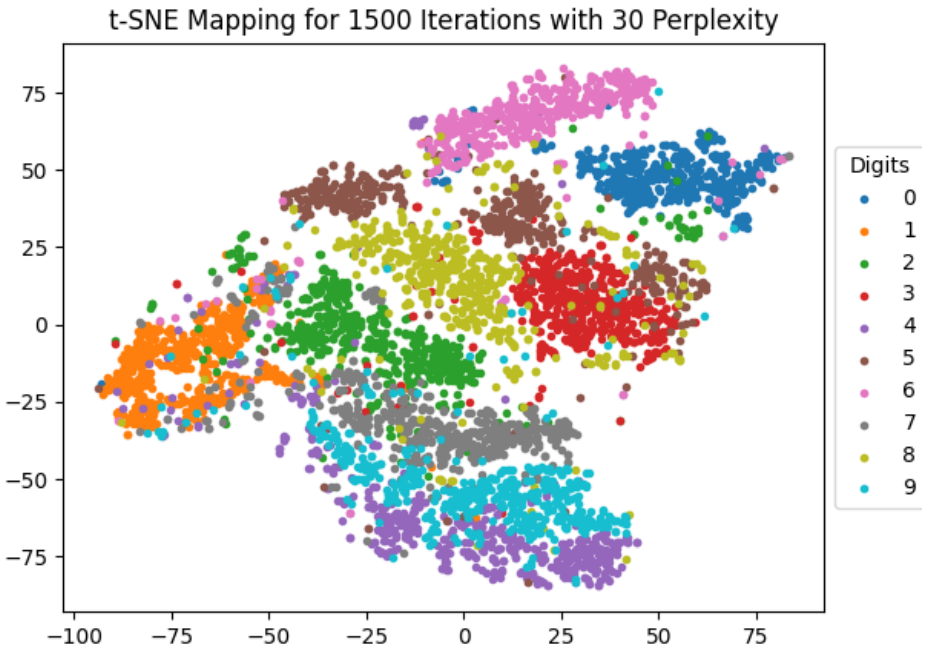

Figure 17. t-SNE Mapping for 1500 Iterations with 30 Perplexity

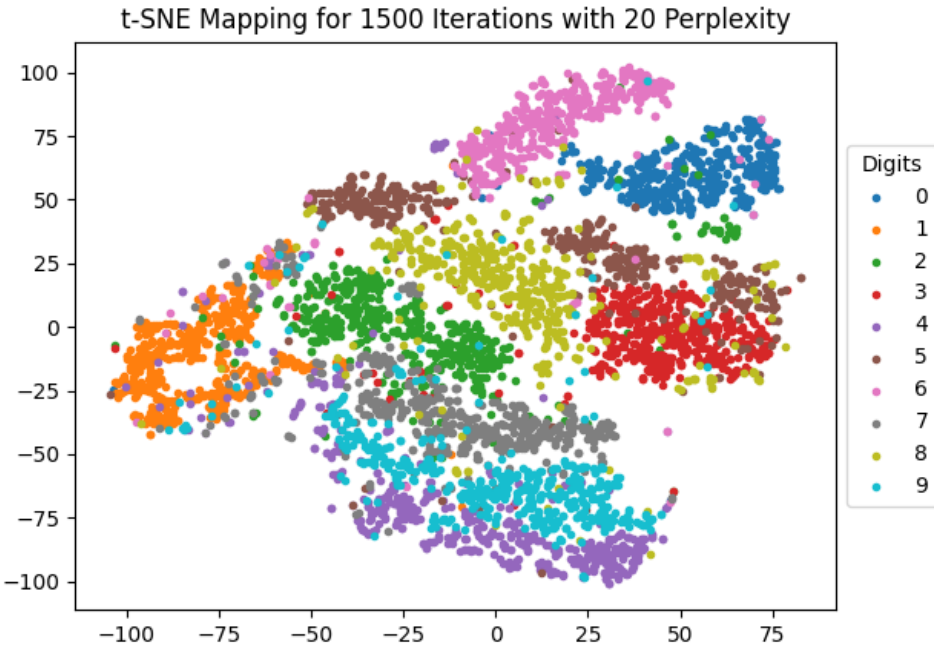

Figure 18. t-SNE Mapping for 1500 Iterations with 20 Perplexity

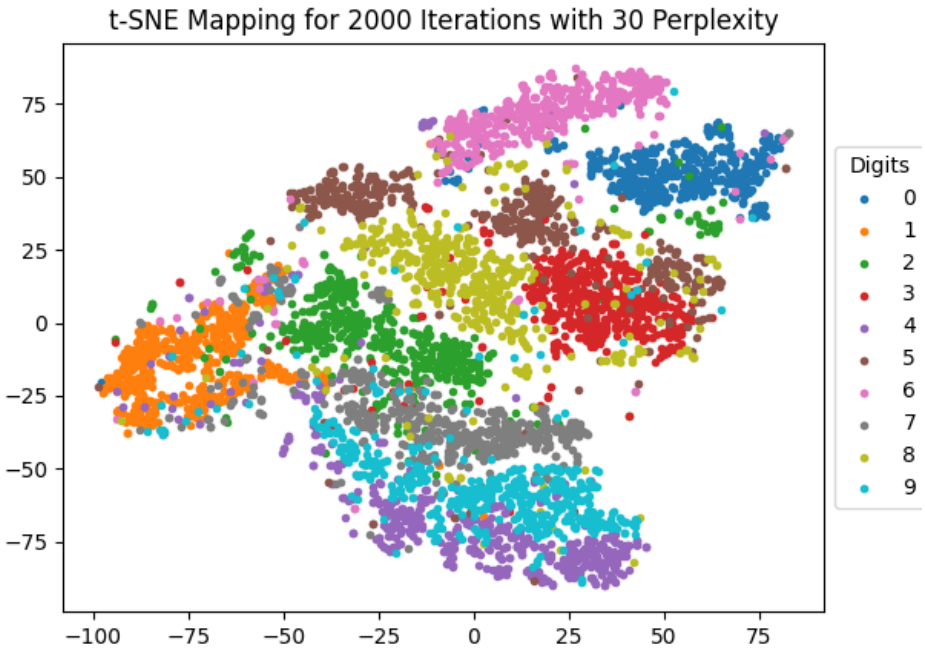

Figure 19. t-SNE Mapping for 2000 Iterations with 30 Perplexity

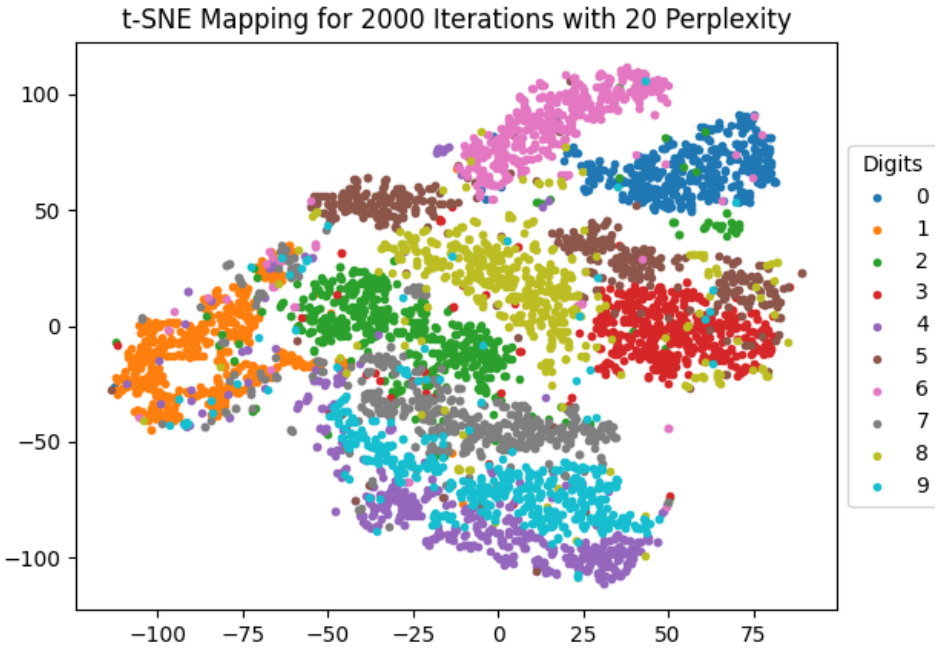

Figure 20. t-SNE Mapping for 2000 Iterations with 20 Perplexity

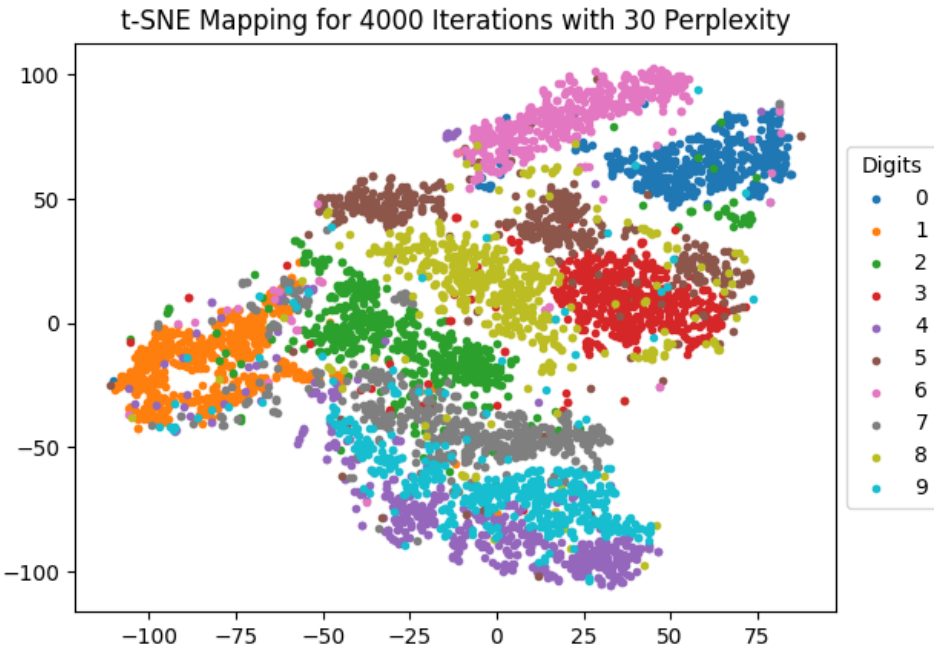

Figure 21. t-SNE Mapping for 4000 Iterations with 30 Perplexity

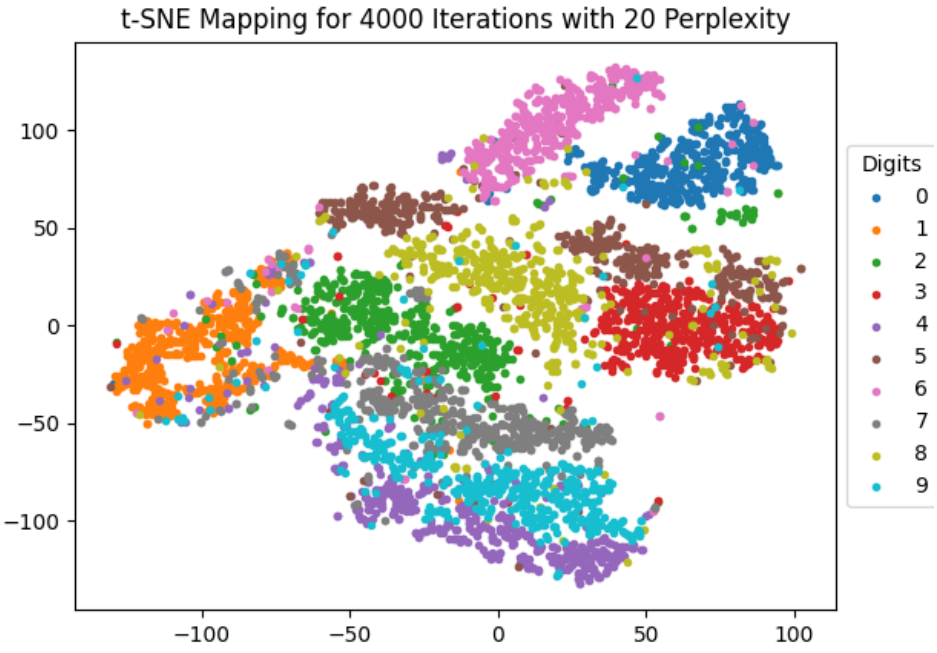

Figure 22. t-SNE Mapping for 4000 Iterations with 20 Perplexity

As a result, TSNE algorithm is fine to map the high dimensional data and it is better than Sammon's Mapping in various terms. Firstly, it is faster because TSNE does not use iterative calculation for mapping unlike Sammon, so it costs less than Sammon and it is really time saver algorithm compared to Sammon. Also, the plot is more understandable in TSNE, meaning that the data in the plot is not overlapping and I can see almost every data point clearly. However, in Sammon, the visualization is problematic because the data points are much more overlapping than TSNE. Therefore, I chose TSNE as mapping algorithm rather than Sammon.

TOOLS USED:

1) Implementation of PCA

PCA implementation is done by understanding the logic of PCA with using course slide and the website below.

(https://towardsdatascience.com/implementing-pca-from-scratch-ea3970714d2b#8006)

2) import matplotlib.pyplot as plt:

This module is used to plot the data.

(https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html)

3) from scipy.io as import loadmat:

This function is used to read .mat files

(https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.loadmat.html)

4) from sklearn.model_selection import train_test_split :

This function is used to split data into train and test.

(https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)

5) import numpy as np:

This module used to create arrays to store data.

(https://numpy.org/)

6) from sklearn.naive_bayes import GaussianNB :

This class is used to apply Gaussian Naive Bayes for Gaussian classifier

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

7) from sklearn import metrics:

This class is used to find accuracy score and calculate classification error.

(https://scikit-learn.org/stable/modules/model_evaluation.html)

8) from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA:

LDA is applied with using this class.

(https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html)

9) from sammon import sammon as SammonMapping :

This function is used to apply SammonsMapping and taken from Github repo, not a library.

(https://github.com/tompollard/sammon)

10) from sklearn.manifold import TSNE :

This class is used to apply TSNE.

(https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html)

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