MNIST Dataset Studies

Hannah Pavlovich

1 Clustering

The purpose of this section is to compare different classifiers and their performance for multi-class classifications on the complete MNIST dataset at `http://yann.lecun.com/exdb/mnist/`. The MNIST database of handwritten digits has a training set of 60, examples and a test set of 10, examples. Use the number of clusters $K = 10$.

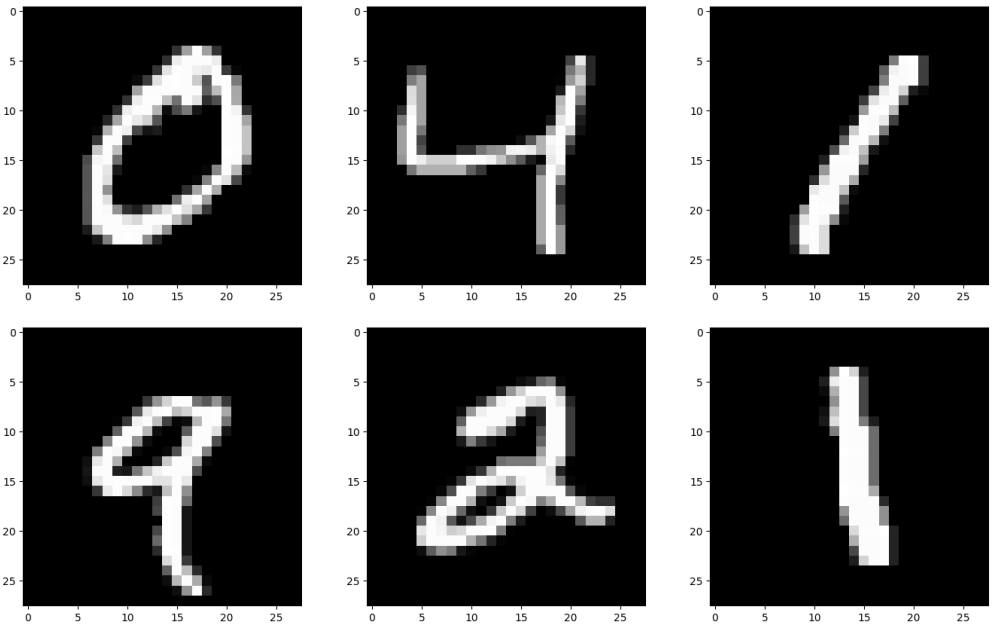
First, I "standardize" the features (pixels in this case) by dividing the values of the features by 255 (thus mapping the range of the features from [0, 255] to [0, 1]).

I use a *purity* score as a performance metric: each cluster is assigned to the class which is most frequent in the cluster, and then the accuracy of this assignment is measured by the number of correlated assigned samples and divided by the size of the cluster:

$$\text{purity}_i = \frac{\text{corrected assigned samples}_i}{\text{size of cluster}_i}$$

for the cluster i.

Below are a sample of images from the dataset:



1. Use the squared-ℓ_2 norm as a metric for clustering and report the *purity* score for each cluster.

group	**score**
0	0.726996
1	0.541382
2	0.697717
3	0.629098
4	0.546388
5	0.333333
6	0.639912
7	0.601915
8	0.600581
9	0.490503

Mean Purity Score for squared-ℓ_2 norm Distance: 0.581
Minimum Purity Score for squared-ℓ_2 onrm Distance: 0.333
Maximum Purity Score for squared-ℓ_2 norm Distance: 0.727

Here, I used unsupervised learning to create 10 clusters for the dataset with the intention of distinguishing numbers from one another.

Once clusters were found using the kmeans algorithm and squared-ℓ_2, this was placed against the real clusters, shown in the training data. Within the new y label clusters, the most frequent k label was found. Then, working under the assumption that the most frequent k label was the correct y label, the purity score is calculated.

In the table above, the purity scores are placed against each other groups. The average score is 0.581 and ranges from 0.333 to 0.727. If we were to randomly assign the labels, the chance of it being correct would be 1/10. Thus this algorithm does better than chance, and is useful.

2. I used k-means with the Manhattan distance (or ℓ_1 distance) and repeat the same steps in Part (1). Please note that the assignment of data points is based on the Manhattan distance, and the cluster centroid (by minimizing the sum of deviance – as a result of using the Manhattan distance) is taken as the "median" of each cluster. Report the *purity* score for each cluster.

group	score
0	0.423
1	0.997
2	0.440
3	0.426
4	0.428
5	0.352
6	0.608
7	0.676
8	0.402
9	0.506

Mean Purity Score for Manhattan Distance: 0.526
Minimum Purity Score for Manhattan Distance: 0.353
Maximum Purity Score for Manhattan Distance: 0.997

The approach to this is similar to the squared-ℓ_2 norm, with clusters found using the Manhattan distance. Once clusters were found, they was placed against the real clusters, shown in the training data. Within the new y label clusters, the median k label was found. Then, working under the assumption that the media k label was the correct y label, the purity score is calculated.

In the table above, the purity scores are placed against each other groups. The average score is 0.526 and ranges from 0.353 to 0.997. This algorithm is similar to using the squared-ℓ_2 norm. The mean score slightly lower here, but the other metrics are similar. From this data, there is not a large difference between the two tables. However, because this is a large dataset, the Manhattan Distance would be a better choice, as it is more robust to large sets.

2 Implementing EM for MNIST dataset

In this section I implement the EM algorithm for fitting a Gaussian mixture model for the MNIST handwritten digits dataset. For the exercise, I reduce the dataset to be only two cases, of digits "2" and "6" only. Thus, I will fit GMM with $C = 2$. The data file also includes the true labels of the digits.

The matrix images is of size 784-by-1990, i.e., there are 1990 images in total, and each column of the matrix corresponds to one image of size 28-by-28 pixels (the image is vectorized; the original image can be recovered by mapping the vector into a matrix).

First, I use PCA to reduce the dimensionality of the data before applying to EM. We will put all "6" and "2" digits together, to project the original data into 4-dimensional vectors.

Now implement EM algorithm for the projected data (with 4-dimensions).

1. To implement EM algorithm, I use the following initialization

   - initialization for mean: random Gaussian vector with zero mean
   - initialization for covariance: generate two Gaussian random matrix of size n-by-n: S_1 and S_2, and initialize the covariance matrix for the two components are $\Sigma_1 = S_1 S_1^T + I_n$, and $\Sigma_2 = S_2 S_2^T + I_n$, where I_n is an identity matrix of size n-by-n.

   To plot the log-likelihood function versus the number of iterations I took the following steps:

   (a) Centered the data about it's mean

   (b) Extract values and Vector using SVD from numpy

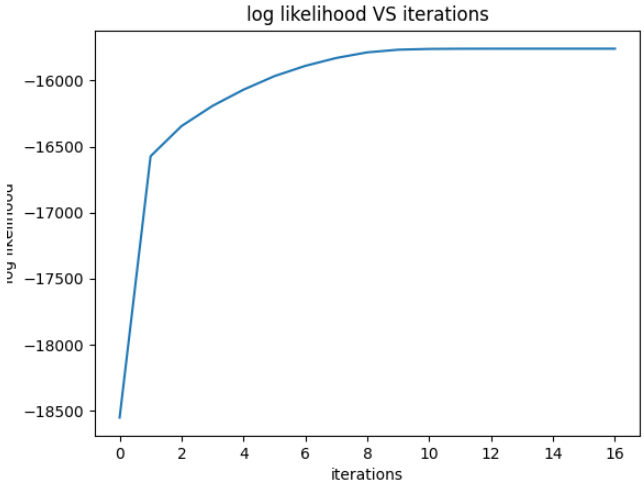
   (c) Project the first 4 principal components onto the data

   (d) establish the covariance matrix, using the equations above

   (e) Create loop, first calculating the expectation over the posterior distribution and storing the log-likelihood and tau values

   (f) In loop, enter Maximization stage, checking if the differences between the new and old means are within tolerance

   (g) If converged, loop ends



   The above plot shows the log-likelihood against the number of iterations. The total number of iterations is 16, and we can see the first 4 iterations did most of the work, after which the convergence was much slower.

2. I report the fitted GMM model when EM has terminated in in my algorithm as follows. I report the weights for each component, and the mean of each component, by mapping them back to the original space and reformating the vector to make them into 28-by-28 matrices and show images. I report the two 4-by-4 covariance matrices by visualizing their intensities (e.g., using a gray scaled image or heat map).

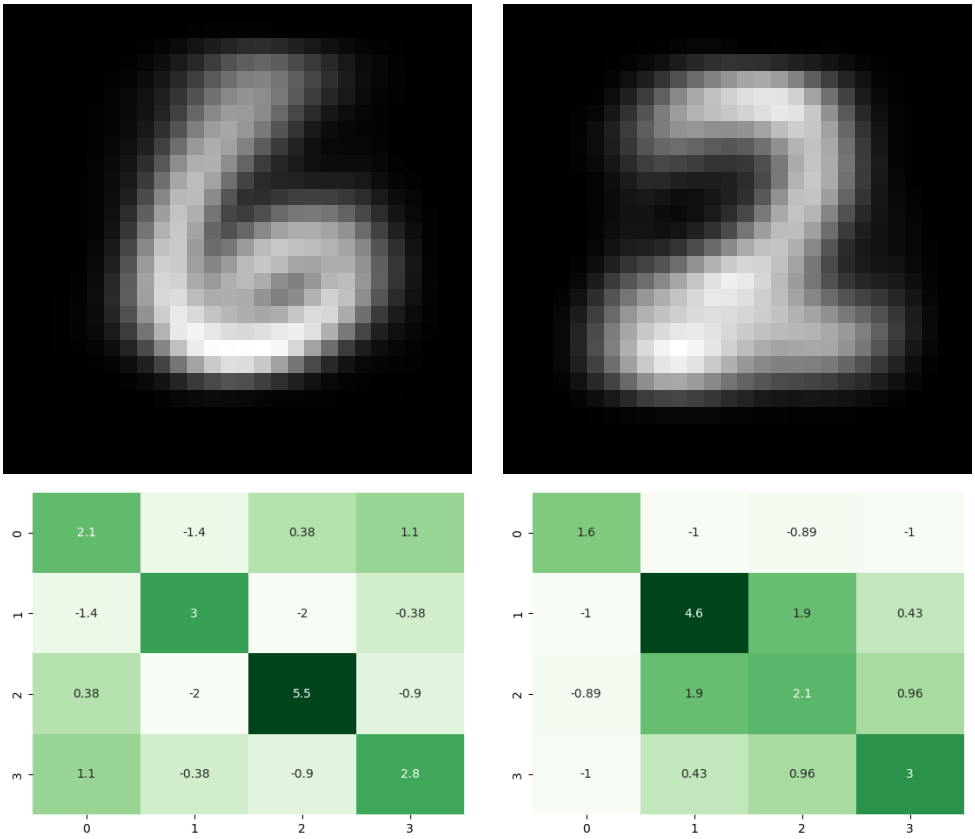
   To compute the means and covariances, I did the following steps for each component:

   (a) Means
      i. Use the output μ from the previous step of finding the log-likelihood, which is in the maximization step.
      ii. Multiply the means through the eigenvectors of each component and add back the previous mean from which we first centered data
      iii. Reconstruct the image

   (b) For the covariance, I used the heatmap package from seaborn.

	PC1	PC2
weight, π	0.513	0.487
means, μ	1.994, 0.574, 0.113, -0.0237	-2.101, -0.605, -0.119, 0.025

   The weights are 0.513 and 0.487, which show a strong and balanced model, suggesting this is a good model. The covariance



The above images clearly display the 6 and 2, with only slight noise. Below each figure are the corresponding covariance matrices.

3. I use the τ_k^i to infer the labels of the images, and compare with the true labels. I report the mis-classification rate for digits "2" and "6" respectively by performing K-means clustering with $K = 2$. Below are the mis-classification rate for digits "2" and "6" respectively, compared with GMM.

   To find the missclassification rate I did the following steps::

   (a) GMM

   i. Extract the labels from the GMM by finding the maximum τ
   ii. Map the 0 and 1's from the above to 6 or 2, the true labels
   iii. Create a subset of the original label model of just figure 6 (or 2)
   iv. With the model subset from above, subset the new labels from GMM.
   v. Missclassification rate $= \frac{\text{number of missclassified 6's (or 2's)}}{\text{total number of 6's (or 2's)}}$

   (b) K-means

   i. Using KMeans package, determine the kmeans model of the data set

ii. repeat items ii. - v. above

	GMM	K-Means
2	6.49%	6.59%
6	0.94%	6.15%

Above are the final results. GMM's missclassification rate is less than that for K-means, which means that GMM is the better model for this dataset.

3 Comparing multi-class classifiers for handwritten digits classification

This exercise is to compare different classifiers and their performance for multi-class classifications on the complete MNIST dataset at `http://yann.lecun.com/exdb/mnist/`. The MNIST database of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. I compare **KNN, logistic regression, SVM, kernel SVM, and neural networks**.

To implement:

- "Standardize" the features before training the classifiers by dividing the values of the features by 255 (thus mapping the range of the features from [0, 255] to [0, 1]).
- Adjust the number of neighbors K used in KNN to have a reasonable result using cross-validation.
- Use a neural networks function sklearn.neural_network with hidden_layer_sizes = (20, 10).
- For kernel SVM, use radial basis function kernel and choose the proper kernel.
- For KNN and SVM, randomly downsample the training data to size $m = 5000$, to improve the computation efficiency.

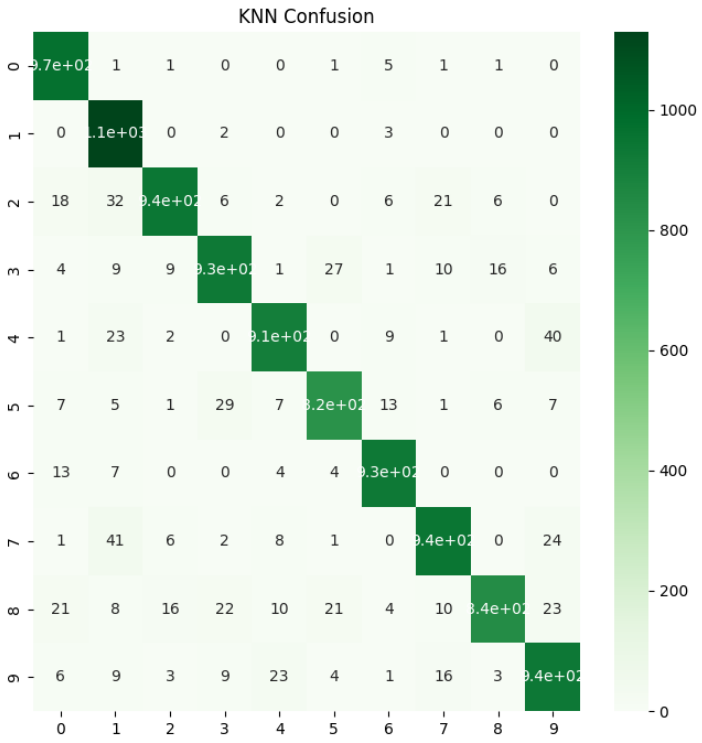
Train the classifiers on the training dataset and evaluate them on the test dataset.

1. I report confusion matrix, precision, recall, and F-1 score for each of the classifiers. Create a table of the precision, recall, and F-1 score of each classifier for each of the digits.

   (a) **KNN**
   To tune KNN, I used GridSearchCV using cross-validation with 5 splits and 28 parameters. From this, I exracted the optimal number of neighbors = 3. The test accuracy from the best model is 93.12%.

Precision, recall, and f1-score all perform similarly well using KNN.
In the confusion matrix, we see that the algorithm mistakes 7 for 1 41 times and 2 for 1 32 times. These are the most significant errors in the matrix.
The precision for class 1 is low for KNN, which means that the classifier incorrectly identified other numbers as 1, which we see in the confusion matrix, also.

Confusion Matrix:

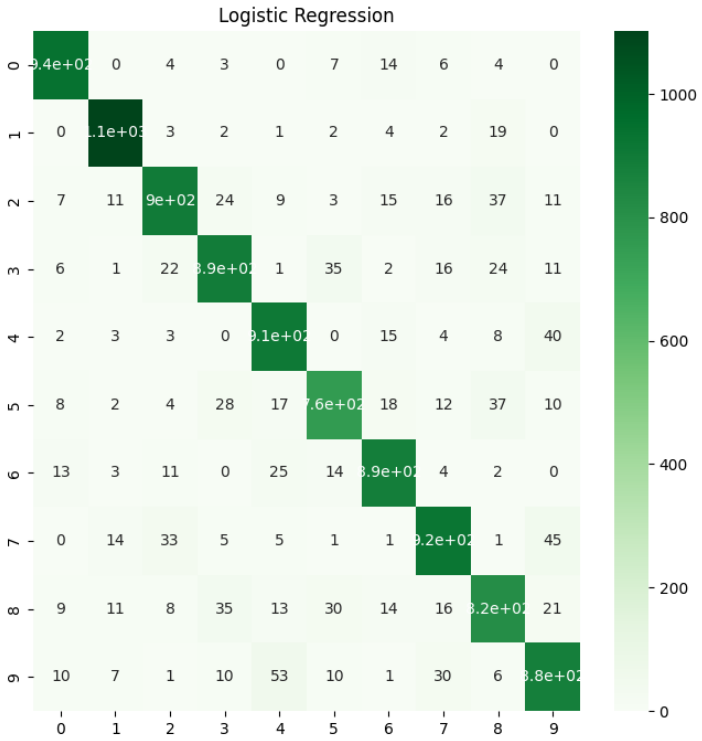


KNN Confusion Matrix

	precision	recall	f1-score
0	0.922857	0.988776	0.954680
1	0.880062	0.995595	0.934270
2	0.968783	0.902132	0.934270
3	0.926326	0.933663	0.929980
4	0.935583	0.931772	0.933673
5	0.931193	0.910314	0.920635
6	0.958333	0.960334	0.959333
7	0.927875	0.926070	0.926972
8	0.968495	0.852156	0.906608
9	0.915493	0.901883	0.908637

(b) **Logistic Regression**
To tune logistic regression, I again used GridSearchCV from scikitlearn. From here, the best parameters were C=0.2 with penalty l2. The best model gave accuracy = 92.58%

Overall, the logistic model does worse than the KNN model, which could be due to the linear hyperplane boundary. The confusion matrix is similar, with errors differentiating 1,7 and 1,2. Class 8 has the lowest precision and recall, which means that the classifier incorrectly identified other numbers as 8's and did not identify the true 8. KNN did much better for precision in class 8, so we may infer that the soft boundary is an asset in this case.



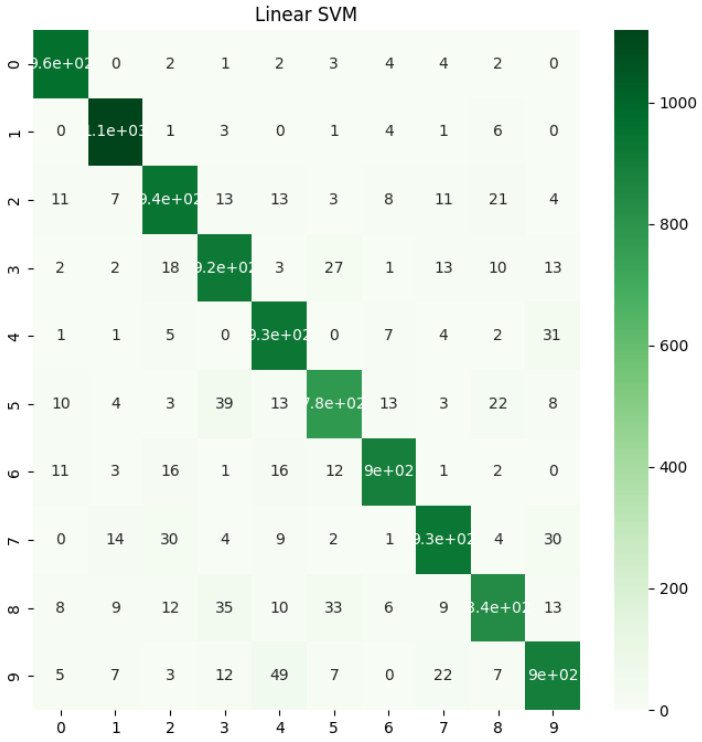
Logistic Regression Confusion Matrix

	precision	recall	f1-score
0	0.951389	0.978571	0.964789
1	0.962738	0.978855	0.970730
2	0.927856	0.897287	0.912315
3	0.902559	0.907921	0.905232
4	0.935517	0.930754	0.933129
5	0.906250	0.877803	0.891800
6	0.937307	0.951983	0.944588
7	0.939664	0.924125	0.931829
8	0.872340	0.883984	0.878123
9	0.914032	0.916749	0.915388

(c) **Linear SVM**

To tune Linear SVM I used GridearchCV using C-values 0.1- 1. The best model had C=0.1. THe best model had accuracy = 92.17Like KNN, the lowest recall is with class 8. However, the precisions for 3,4,5, and 9 are also low, which means this

classifier more often classified other numbers to those in that set. Interestingly, in the confusion matrix we do not see the same high error in distinguishing between 1 and 7. From this I would infer that Linear SVM can most accurately distinguish between class 1 and class 7.



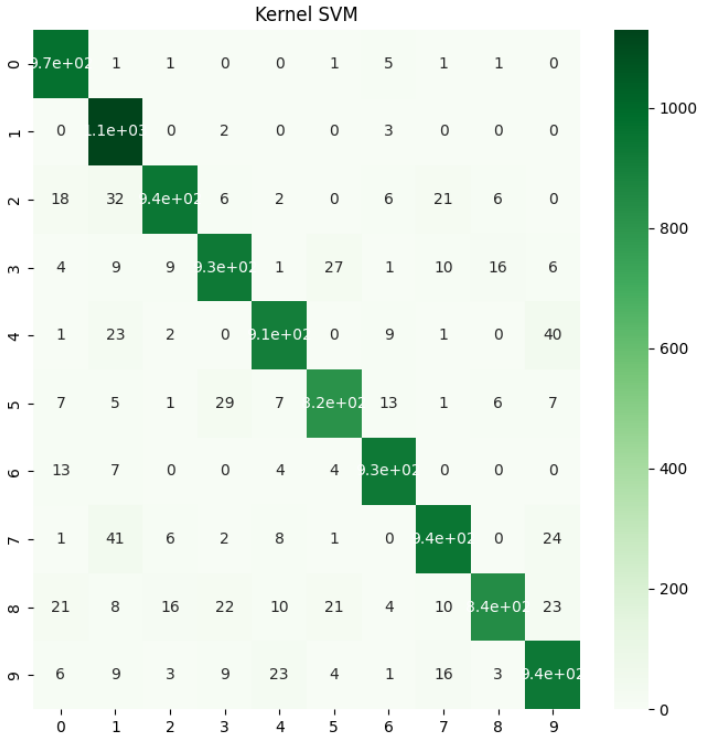
Linear SVM Confusion Matrix

	precision	recall	f1-score
0	0.952475	0.981633	0.966834
1	0.959691	0.985903	0.972621
2	0.912706	0.911822	0.912264
3	0.895044	0.911881	0.903384
4	0.890057	0.948065	0.918146
5	0.898266	0.871076	0.884462
6	0.953191	0.935282	0.944152
7	0.932136	0.908560	0.920197
8	0.916940	0.861396	0.888301
9	0.900602	0.888999	0.894763

(d) **Kernel SVM**

After tuning the Kernel SVM, the optimal C=4.0, and the best model has accuracy = 93.39%. This is better than the Linear SVM accuracy, which could mean that the data is better suited for a non-linear classifier.

The precision for class 8 is much higher than the other models, but the recall is still low, meaning class 8 is incorrectly classified more often than the other classes. In the confusion matrix, we see similar difficulties in distinguishing between 1 and 7 as we did with the previous classifiers

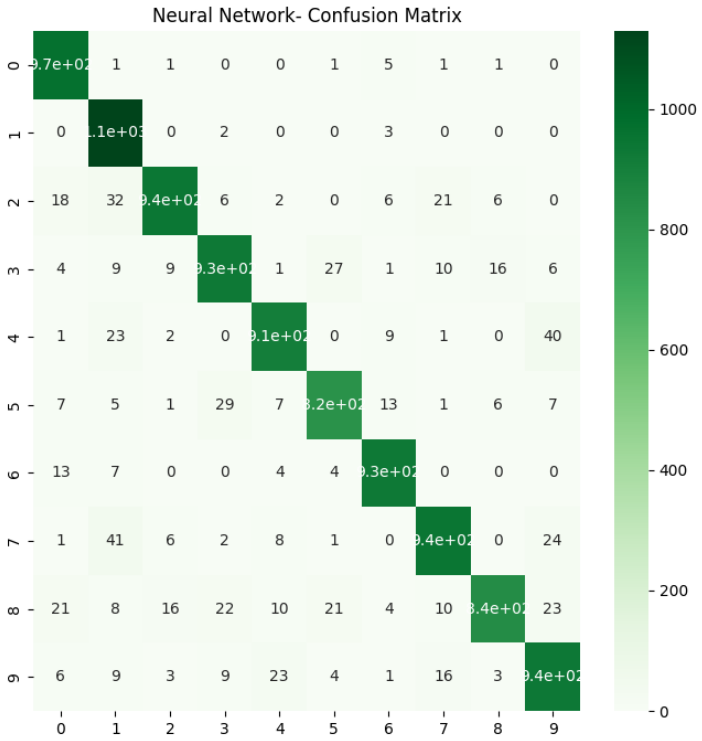


Kernel SVM Confusion Matrix

	precision	recall	f1-score
0	0.931796	0.989796	0.959921
1	0.893281	0.995595	0.941667
2	0.961185	0.911822	0.935853
3	0.929789	0.917822	0.923767
4	0.942768	0.922607	0.932578
5	0.933638	0.914798	0.924122
6	0.956790	0.970772	0.963731
7	0.940299	0.919261	0.929661
8	0.963261	0.861396	0.909485
9	0.903382	0.926660	0.914873

(e) **Neural Networks**

The best Neural Network model had accuracy = 94.79%, which is higher than the previous classifiers. It also has the best precisions and recalls overall, with none under .91. The recall for class 8 is also must higher than previously seen. Even so, the most difficulty came in distinguishing between 1 and 7.



Neural Network Confusion Matrix

	precision	recall	f1-score
0	0.967413	0.969388	0.968400
1	0.978966	0.984141	0.981547
2	0.947420	0.942829	0.945119
3	0.910318	0.934653	0.922325
4	0.957688	0.945010	0.951307
5	0.937071	0.918161	0.927520
6	0.957983	0.951983	0.954974
7	0.958783	0.950389	0.954568
8	0.922919	0.921971	0.922445
9	0.936647	0.952428	0.944472

2. (5 points) Comment on the performance of the classifier and give your explanation why some of them perform better than others.

KNN	Logistic Regression	Linear SVM	Kernel SVM	Neural Networks
0.9312	0.9258	0.9217	0.9339	0.9479

Neural Networks performs the best on the data. Neural Networks perform well on high data sets, and I believe with more hidden layers, the performance can be even greater.

The non-linear classifiers KNN and Kernel SVM did markedly better than Logistic Regression and Linear SVM, probably due to the linear hyperplanes of the latter two. This would suggest that the data is linearly separable, and thus they are not appropriate for this data.