ECE 2720

Final Project Report

Mueed Ur Rehman 11/30/18 mr2265

Table of Contents

The Task:

We were required to train a classifier on the MNIST dataset. The dataset is divided into 60000 images with corresponding labels for the digits (0 to 9) for the training set and 10000 images for the test data. Each image is single channel (grayscale) with dimensions 28 x 28.

Loading the Data:

The data was provided in the binary format that Yann Le Cun had made available on his website. More accessible versions of the data did exist (a few .csvs can easily be found online) but as the project required us to first process and get the data in, I moved to use the struct module that allows us to interpret strings as packed binary data.

This was a fairly simple task. The file was opened and the struct.unpack method was used to unpack the values into integers and create a numpy array out of it. A helper function was written for this purpose. The function was then called in order to create 4 different numpy arrays: the training data, the training labels, the test data and the test labels.

Searching for an appropriate Classifier:

I began preliminary research into the most optimum classifier to use for this task. There were a few basic classifiers for which I ran preliminary tests. These were just to get an idea for the classifiers and the accuracies involved.

I had a looked at quite a few classifiers. These included logistic regression classifiers, multilayer perceptrons, random forests, K Nearest Neighbor classifiers and SVMs. I quickly built models and got ballpark figures for the performance of each of the classifiers.

There were a few reasons that compelled me to not opt for these.

I had run preliminary tests using Scikit Learn on most of these classifiers. Logistic Regression was giving classification accuracies around 85% for the test set. SVMs with linear kernels were around 91%. Random Forest offered classification accuracies of 93%. Stochastic Gradient Descent was relatively quick to train and yielded classification accuracies of approximately 89%. The best so far was the K Nearest Neighbor Classifier, which yielded a classification accuracy of 96% roughly. However, I decided to not opt for the K Nearest Neighbor Classifier due to the very large model sizes (north of 500 MB per model) and that the best K Nearest Neighbor models did not seem to be breaking 97.5% in terms of classification accuracy. Similarly, Random Forests also struggled to get into the highest echelons of classification accuracy.

Logistic Regression was rather simple and was also more vulnerable to outliers as it employed the logistic loss function versus the hinge loss function used by SVMs. Optimized SVMs were the second-best option as research showed they could break 98% in terms of classification accuracy, and Neural Networks were clearly the best, as indicated by multiple research papers demonstrating their effectiveness on the MNIST dataset. The current state of the art on MNIST is a convolutional neural network which achieves a stunning 99.8% classification accuracy.

A very useful resource in this regard was the page of Rodrigo Benenson on Github, where he maintains a list of the state-of-the-art classifiers for different datasets, including MNIST. An analysis of this led me to conclude that the best possible option would be to develop a Convolutional Neural Network. Support Vector Machines were a close second. However, I wanted to challenge myself to go beyond the scope of the course curriculum, so I went ahead with the design of a CNN.

ECELinux Cluster Woes:

Having developed both the multilayer perceptron model and the first convolutional neural network, I decided to login to the ECELinux cluster and see how to get it to run a task in the background for me so that I could offload the training work. By chance, I decided to check the availability of several neural network libraries and found that Keras, Tensorflow, Theanos and others were all missing from the ECELinux cluster Python installation. This compelled me to pivot and explore other options. However, this change has altered the structure of this report. I intended to dedicate the last 5 days to report writing and drafting up why my CNNs worked so well. As I changed classifiers, I did not get around to training the second CNN I had been designing and relegated the first draft of my report to **Appendix A**. It would be of interest, I am certain, as I have gone into the design choices for my CNN such as why I opted for a custom TensorFlow build and the significance of having convolutional, pooling and dropout layers in my CNN. In any case, SVMs it is. 5 days to go till the deadline. No pressure.

Support Vector Machines:

As a convolutional neural network was not an option anymore, I decided to go ahead with a Support Vector Machine instead.

The issue of computational efficiency and training times remained. I was already aware that the best performing support vector machines were either based on the RBF kernel or on the Polynomial kernel.

My issue with these was that the algorithms used for the computation of the RBF kernel has $O(n^2)$ complexity and polynomial kernel functions with a higher degree are also computationally expensive. Due to the large size of the dataset and subsequent optimization required to get the correct hyperparameters for the SVM in general, I decided to split up my work. I had tried training an RBF SVM on my laptop on my own. Perhaps it was the poor parameters and lack of scaling at the time, but one hour of slowing my laptop to a crawl yielded no results.

Tuning the Hyperparameters:

When I arrived at Cornell, I had, out of curiosity, gone and done an assessment of the computers available for student use at Cornell. The ECELinux cluster was one option. However, the computers at Carpenter had better specifications and Carpenter was open 24/7. I decided to camp in at Carpenter during the nights in order to get the work done. Fun times.

I knew I had to run a slew of tests fairly quickly. I had to determine the most appropriate kernel for my SVM and then optimize the hyperparameters for that kernel. I was also interested in preprocessing to improve the classification accuracy once I had found the most optimum hyperparameters.

Here the issue is more complex as we have two parameters C and gamma to optimize. Now, the best method would be to iterate through all possible values of C and gamma and see which ones yields the optimal accuracy for the training set. However, this presents an issue as this would dramatically increase the training time.

Here, I decided to refer to already established SVM parameters and further fine tune those in order to generate better results and save time. I found a set of SVM parameters and corresponding Cross Validation Accuracy online. These had been created by an individual using an 85:15 split of the 70000 MNIST samples. This was very close to the split that I would have to use myself.

I have provided a figure using their data that condenses their results. This figured indicates that the optimum hyperparameters were around a C of 5 and a Gamma of 0.05. The figure below gives the average accuracy on the Cross Validation Set for different C and Gamma values for the RBF Kernel.

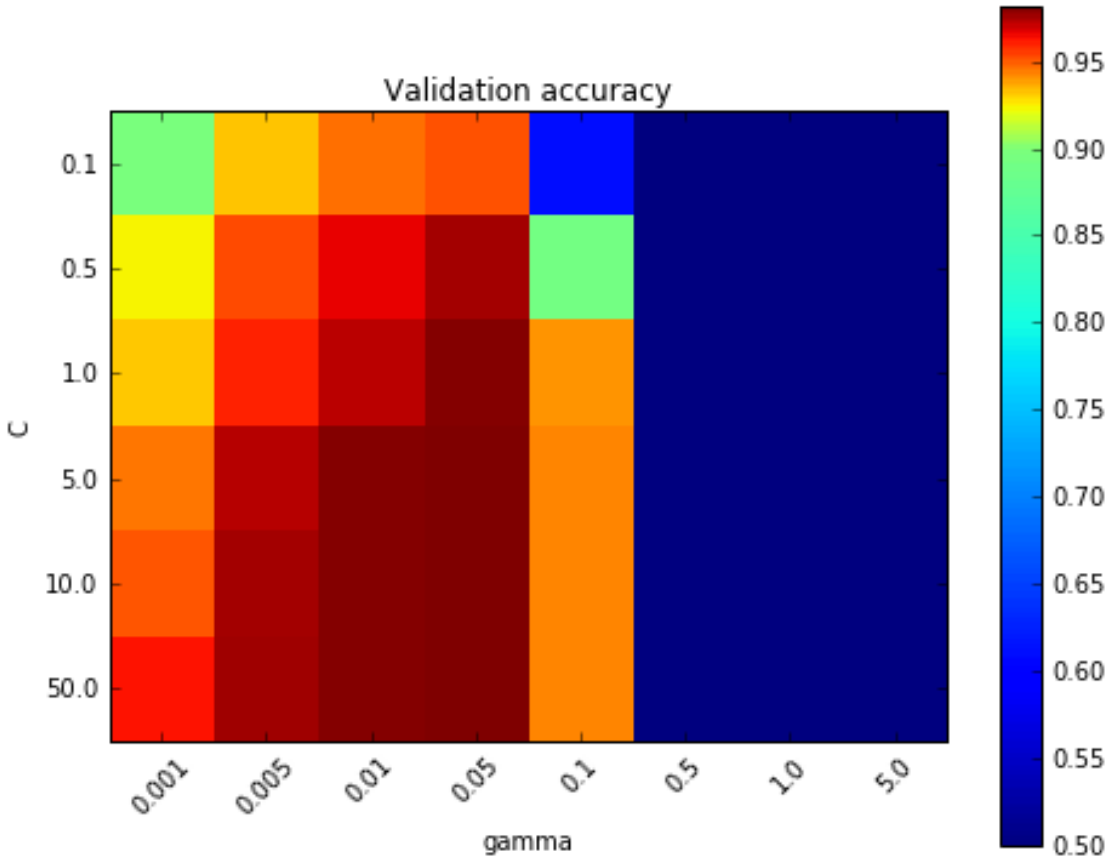

Figure 1 Adapted Cross Validation Accuracies for C versus Gamma values on an RBF kernel SVM trained on a subset of the MNIST dataset. Source is in Acknowledgements.

It should be noted that the validation here is the average of the accuracies on the cross-Validation sets (all subsequent figures have the same principle, validation accuracy refers to average accuracy after k-fold cross validation). This person had used a 3-fold (k is 3) cross validation strategy. This meant that they had dropped one third of their training data (which was initially 85% of 70000 samples), trained their classifier on the remaining two thirds and then tested it on the one third that they had dropped. This was repeated three times (each time a different third of the training data was dropped). The average cross Validation accuracy was reported by them, and I have generated the above Heat Map from it.

To further tune the parameters, rather than run a Python code file multiple times, I decided to opt for a better solution. Here, I used GridSearchCV from skLearn to select the kernel for me. Essentially, it iterates through a matrix of C and Gamma values provided by us and reports the average cross validation accuracies for each combination. The model with the highest average cross validation accuracy is the one singled out as the best and may be used for predictions on the test set.

I was aware that the different kernels may require radically different hyperparameters. Rather than randomly selecting one and failing to capture a kernel's classification prowess accurately, I decided to run a GridSearch for each individual kernel. For the C and Gamma values, I decided to go with a wide range using the np.logspace command so that I could find a regime of interest and subsequently hone in on it later if I saw potential within that classifier.

I set up a PC at Carpenter to do the work for me. I selected the same C and Gamma values as in the Figure given above, opted for the Linear kernel as it was supposed to be the fastest and set up the task to use all physical cores on the PC. It was 11pm, and I was hungry. I went for pizza.

I returned and found that the PC was switched off with someone else working where I had been. Mouth already burnt from piping hot pizza, I was considerably annoyed. I doubted they had taken over my PC as there were others around that were empty. A closer analysis of the CIT Lab policy led me to understand that sessions would be automatically terminated after an hour of inactivity. Of course, the inactivity criteria for Windows generally boils down to a lack of user input, something I could easily work around.

I did not have sufficient access to the PC so I could not schedule a task to work around this or run a script. I configured a simple utility (MoveMouse) to move the cursor slightly every 30 seconds and to run in the background every time I left the PC. I again set up the GridSearch for the Linear kernel and got to work.

About an hour later, I realized that things were going to get difficult. I had initially created a 6x8 matrix of C and gamma values to explore. I had also incorporated 3-fold cross validation like the individual had. This meant that within the training set, 40000 samples would be used for training while 20000 samples were used for cross validation. This meant that GridSearchCV would produce a total of 6x8x3 = 48x3 = 144 fits. In other words, 144 different SVM classifiers would be created.

Even though I had parallelized the task to use all 4 physical cores on the PC, I noticed that the cross-validation jobs were running asymmetrically in terms of the training time required. The first job for a particular C and gamma combination would take around 5 minutes say and subsequent jobs would take 25-30 minutes. It appeared that the two subsequent cross validation tasks were taking up to 5 to 6 times longer.

As I was surrounded by empty PCs, I decided to start parallelizing this task. My initial motivation was to have all of the PCs work in conjunction to perform the GridSearch for me (I had assumed these would be connected via Ethernet). My intention was to use spark-sklearn (a Scikit-learn integration package of the Apache Spark Computing Framework) to parallelize the GridSearch task across multiple machines. However, I decided to forgo this plan due to considerations of scaling efficiency. Some quick Google Searches made me realize that while I could get spark-sklearn to have a distributed GridSearch, it would not scale perfectly. Time was of the essence. This is why I decided to branch out the task to further machines.

The initial GridSearch I had started was without any scaling whatsoever. I then launched another PC and had it run the same GridSearch, but only after dividing each of the Model Parameters by 255 as this was one of the scaling strategies suggested by numerous resources online.

Here, I was momentarily stumped by integer Division.

I had stored the training data in a variable trainImage and scaled it by trainImage = trainImage / 255. This resulted in integer division and turned practically every single data point per image into zero. I chanced upon this because I had run a fit for a linearSVC() on my machine (as I waited on the GridSearch) using the scaled data and found a classification accuracy of 33% with around 25% of the training time for the non-scaled SVC. This was surprising as there was near unanimous consensus online that scaling generally increases classification accuracy rather than reducing it. I printed out the trainImage data, realized that I had rows upon rows of zeros, amended the code and relaunched the linear SVC kernel on the second PC after ensuring that the division resulted in floating point values. What is the difference between 255 and 255.0? Around 60% of your classification accuracy apparently.

By observing the console output (I had increased the verbose parameter for the GridSearchCV so that I would get more frequent print statements and have a better idea of how the model was progressing), I deduced that scaling was helping. Not only was the training time down, but now I was consistently completing a job every 4-5 minutes.

In parallel to this, I decided to start a RBF kernel with the same C and gamma parameters. Here, I did have some idea of the C and gamma parameters that would work from Figure 1.

As I was aware that RBF training would take significantly longer (my initial tests suggested each individual fit could take up to 30 minutes), I decided to use a subset of the data instead. Thus, I now proceeded to run GridSearch with the RBF kernel and a subset of the original training data (5000 samples for the wider searches increasing up to 20000 samples as I honed in on the parameter values). The difference in training time was rather remarkable.

This search led me to conclude that the optimum parameters were around C=3 and gamma = 0.03. Of course, I would then return to validate this data.

In order to do so, I decided to launch up 4 additional PCs. Each of these would look at the full MNIST training data and explore the C and gamma values in the vicinity of the C and Gamma values I had identified by running GridSearch on a subset of the train data. It should be noted that the subset I chose had been obtained by randomly selecting rows from the original 60000 rows. However, the numpy seed value had been fixed to 7 to ensure reproducibility of my results if required.

To reduce processing times further, I carried out 2-fold cross validation rather than 3-fold cross validation. I found that the overall test set accuracy remained unchanged when using 2-fold cross validation and decided to proceed with this for all subsequent tests.

Below are some of the results I obtained for the RBF kernel SVM that I trained on a subset of the training data. I have specified the number of samples included, the C and Gamma values, the training time along with the accuracy on the Test Set.

Samples	C	Gamma	Training Time (seconds)	Accuracy on Test Set
5000	2	0.04	13.2	96.17%
10000	3.5	0.03	31.6	97.19%
20000	3.5	0.03	216	97.81%

I have enclosed 3 different figures to show the cross-validation accuracies that I achieved for these runs. The darkening reddish maroon shade shows how convergence towards optimum hyperparameters was quickly achieved.

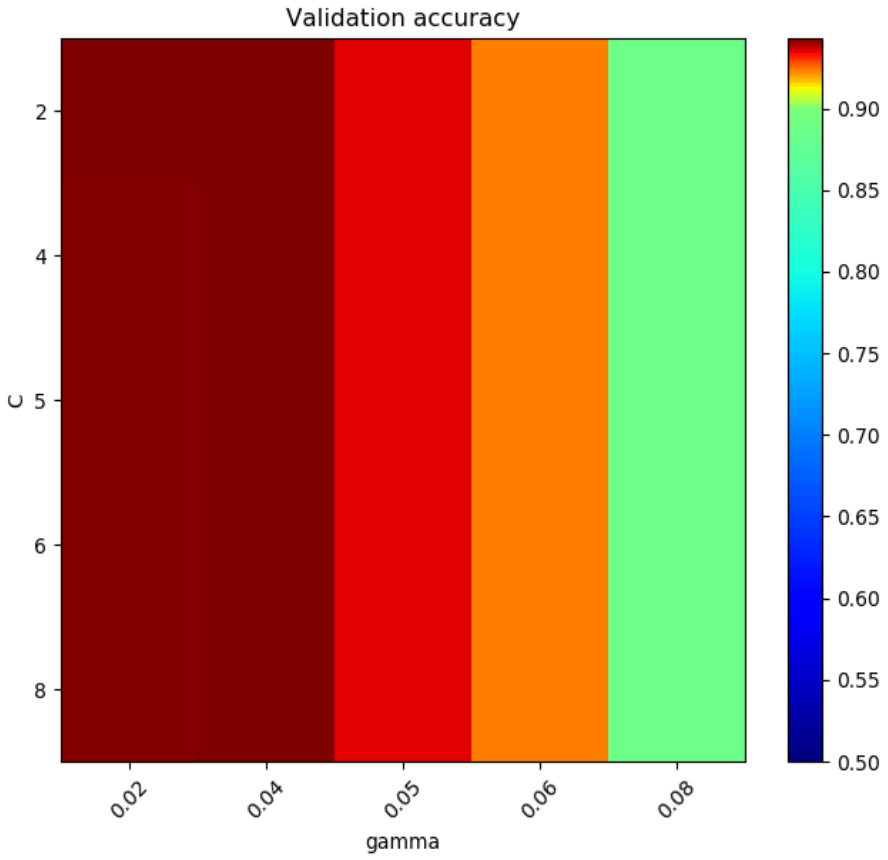

Figure 2 5000 Samples. Cross Validation Accuracy for C Versus Gamma using GridSearchCV

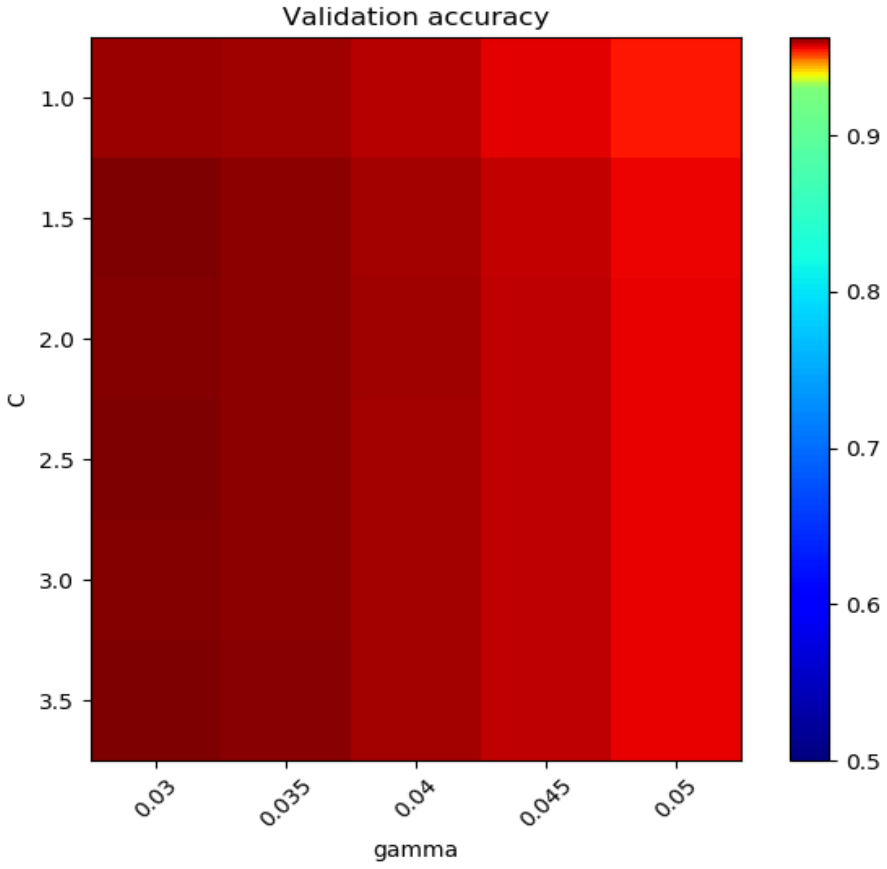

Figure 3 10000 Samples. Cross Validation Accuracy for C Versus Gamma using GridSearchCV

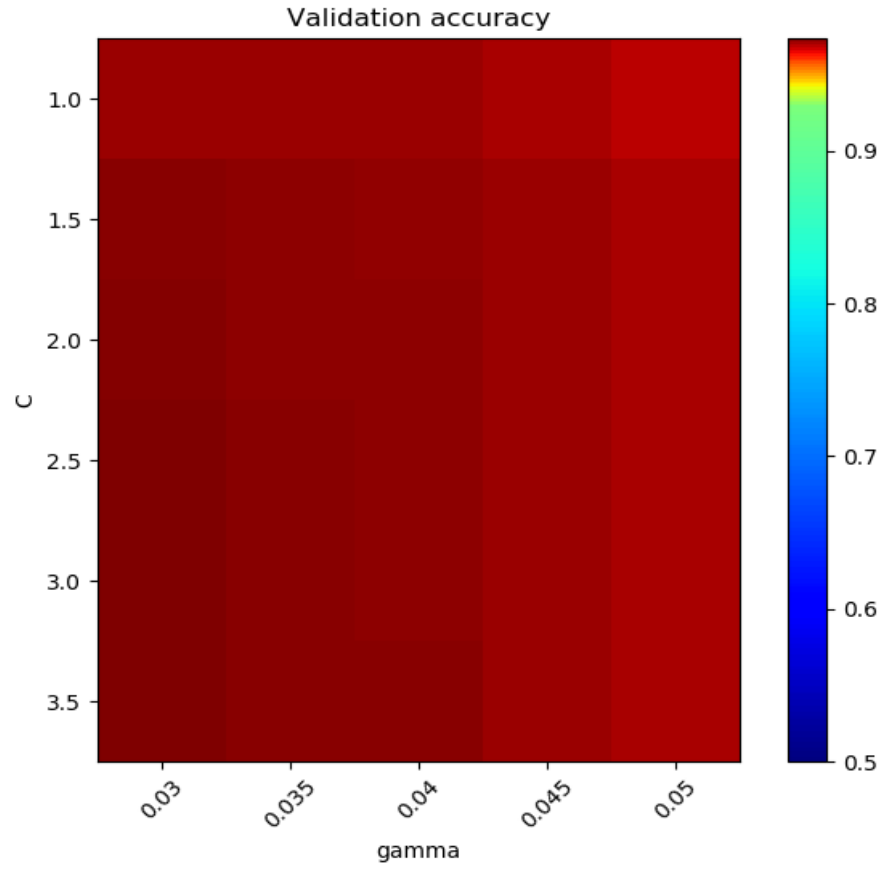

Figure 4 20000 Samples. Cross Validation Accuracy for C Versus Gamma using GridSearchCV

It should be noted that the first SVM I began with (a linear kernel without any form of scaling for the data) was still chugging along trying to get 144 fits done (the fit time was increasing with increasing C). Furthermore, after 15 hours of waiting for the linear SVM models to produce a result, I closed the job. Clearly, this was not working.

Meanwhile, I noticed that a PC that I had reconfigured to test scaling between -1 and 1 was not faring well either. This actually produced lower accuracies across the entire C and Gamma range and was discarded as a scaling option.

Consequently, all results indicated within tables this report have been obtained by scaling the test results between 0 and 1. This preprocessing has been applied as a default for every classifier from here onwards.

The results of the 4 PCs that had been working with the RBF Kernel on smaller C and gamma matrices but the full MNIST training data led me to conclude that a C of 4.0 and a gamma of 0.03 were optimum.

I then proceeded to run a final search around these values with a C = [3.9,4,4.1] and a Gamma of [0.025,0.03, 0.035]. My overall parameters did not change.

The model accuracy was 99.56% at this point. The highest accuracy had been achieved with an RBF kernel as expected based on prior research as the RBF kernel can, with sufficient data, generalize better as it defined a much larger function space than the other kernels.

Other SVM Kernels:

In parallel to this, I began looking at the polynomial kernel as well out of curiosity. My interest in this was motivated by the fact that the best SVM to date on the MNIST dataset had actually used a polynomial kernel function of degree 9 along with a rather involved preprocessing scheme to achieve their accuracy results. What I discovered was that the polynomial kernel did yield good accuracy, generally upwards of 94% on the test set. However, it was heavily preferential towards larger C values. Training times are slower for higher C values, which made it problematic to do a more extensive probe of these, even across multiple machines.

In contrast, the sigmoid kernel required extremely long training times, even exceeding 2 hours to do a simpler 2-fold cross validation fit for a particular C and gamma values.

The following table condenses my results of running the other SVM kernels. The data I have presented here is only a subset of the 30+ simulations that I ran. I did not run predictions on all of my data sets as for any given GridSearch, I would look at the most optimum parameters found by GridSearch and only run predictions on those.

All training times have been reported after testing on a Core i7 6700 machine with 16 GB of RAM. These are the PCs available for student use in Carpenter Hall. This applies to all other training times given in this report.

Kernel	C	Gamma	Accuracy on Test Set	Degree (for Polynomial Kernel)	Training Time (minutes)
Sigmoid	0.01	0.01	11.35%	NA	65.4
Sigmoid	0.1	0.01	89.88%	NA	15.1
Sigmoid	5	1	11.06%	NA	47.3
Polynomial	55	0.012	96.6%	4	7.3
Polynomial	3.5	0.025	95.7%	5	52.0
Linear	0.1	NA	94.72%	NA	5.0

As noted previously, these were some of the more optimum Sigmoid C and Gamma parameters that I had discovered. For the Sigmoid Kernel, it may appear that increasing the C value yielded significantly higher accuracy. However, this C of 0.1 is actually the optimum C value. Increasing the C to 0.5 and beyond caused the prediction

accuracy to again drop below 85%. Similarly, increasing the gamma also resulted in a similar loss in classification accuracy as the we deviated from the optimum value of 0.01.

It should be noted that 48 different combinations had been tried for the linear kernel without scaling. However, each iteration of GridSearch could take up to 100 minutes for the non-scaled training process. Prediction results took too long and were not recorded.

All polynomial kernels gravitated towards ever increasing C values. Higher degrees resulted in higher accuracy for the same C and gamma in general but training times increased 8-9 times per fit.

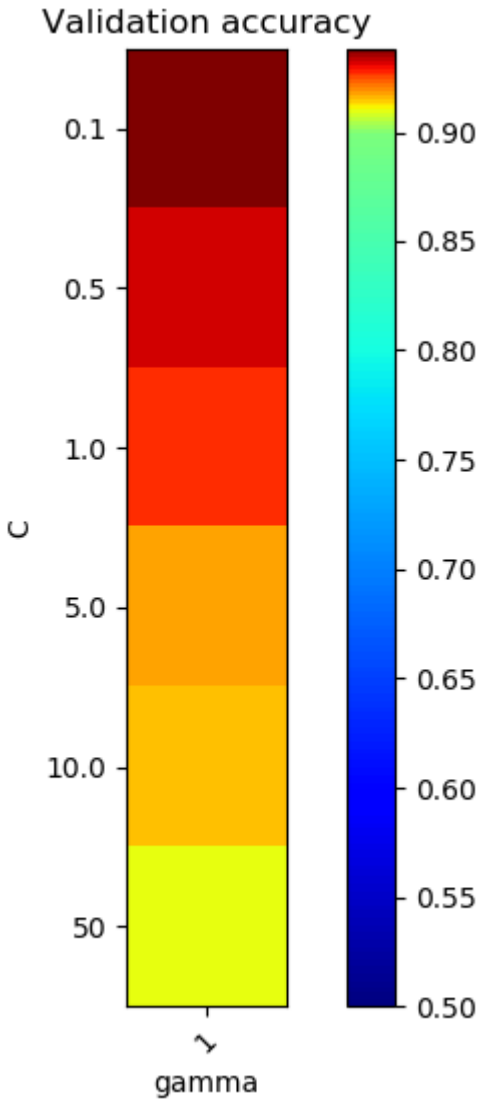

Figure 5 Linear Kernel. Gamma Values do not matter so only 1 has been plotted. It is clear that classification accuracy on the cross-validation set is increasing with decreasing C value

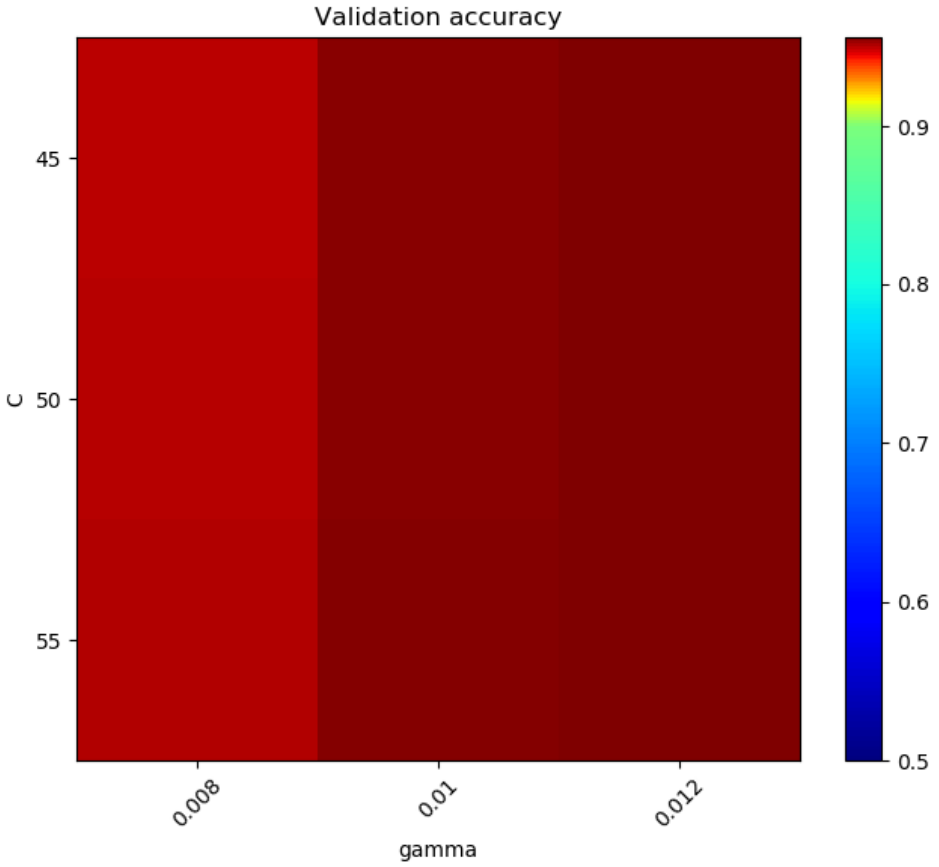

Figure 6 Polynomial Kernel with Degree 5. There is an obvious trend with darker red (higher classification accuracy) corresponding with higher C and gamma values. This makes optimizing the polynomial kernel rather expensive.

Additional Results for the RBF Kernel:

Here are some other results of the RBF Kernel. It should be noted that these were run without any form of preprocessing apart from scaling from the pixel intensities to be between 0 and 1. These were run on the Full MNIST dataset in order to arrive at a final value for the most optimum hyperparameters.

C	Gamma	Training Time (minutes)	Accuracy on Test Set
3	0.05	23.2	98.34%
3	0.03	8.8	98.50%
3.5	0.05	28.3	98.19%
4.5	0.05	22.5	98.36%
4	**0.03**	**9.8**	**98.56%**

The last rows gives the final model parameters that I had obtained before I went to meet the professor.

Preprocessing:

After a discussion with the Professor, I began to look into preprocessing. Here, I discovered a few things of interest.

I decided that I needed more information on the data I had. Pandas was more convenient for looking at the data than Excel, which would take ages to open the data, even on a laptop with a high end SSD (it might be CPU bound). I took the numpy arrays that I had created, added in Column labels, appended the arrays and labels together and put all of the training data in Pandas. I used the .describe() method of a Pandas dataframe to look at the data and discovered that there seemed to be certain pixels whose value was a constant from each image to the next. I found this by noticing that the minimum value for certain pixels was 0, which would imply that the

pixel remained white across all the different images. Below, you can find a Jupyter Notebook print of the .describe() results that led me to conclude this.

	Pixel 1	Pixel 2	Pixel 3	Pixel 4	Pixel 5	Pixel 6	Pixel 7	Pixel 8	Pixel 9
count	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 7 Importing the MNIST data into a Pandas DataFrame then running .describe() to get quickly compute key statistics on each column

Another bar chart visualizing the data has been given in the appendix.

I then wrote methods to delete the repetitive pixels (in total 65 black pixels with an intensity value of 0 were deleted). I reasoned that these pixels would not impact classification accuracy but ran another GridSearch at Carpenter to be safe. While my GridSearch successfully completed (I noticed a reduction in training times on the RBF kernel), I could not predict the classes for the output values as the test data still had 784 pixels. To get around this, I loaded all 70000 samples into a Pandas dataframe and then deleted the white pixels that remained constant. This time, 65 pixels were deleted. As the features were now consistent, I reran GridSearch and found that apart from reduced training times, the classification accuracy remains unchanged in all cases on the model. Results for these are tabulated further down.

Next, I began to look into deskewing, which is the process of straightening images in order to align it vertically. This helps correct for the slant that letters tend to have when they have either been scanned in incorrectly or written at an angle.

Of course, correcting the slant exactly is a rather involved process. The big question is one of accurately figuring out the slant angle for each image and then rectifying it. I had to conduct significant research into how to do this.

The method I ended up considers deskewing as an affine transformation. Affine transformations are linear mappings that preserve points, lines and planes. Parallel lines remain parallel under such a transformation. Normally, affine transformations have been used to account for geometric distortions and deformations with images that arise from suboptimal camera angles.

So, the skew on the image can now be considered as an affine transformation of the form:

$$\text{Skewed Image} = \text{A}(\text{Original Image}) + \text{B}$$

We do not know what the exact transformation is. However, an estimate can be made. First, we align the center of mass with the center of the image. Furthermore, the skew angle is approximated using the covariance matrix of the pixel intensities for each image. The transformation matrix that allows us to reverse the affine skew transform is given as follows.

$$\begin{bmatrix} 1 & 0 \\ \alpha & 1 \end{bmatrix}$$

$$\alpha = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(X)}$$

We then use the affine_transform module inside scipy.ndimage to return the straightened image by applying this transformation.

This technique was largely adapted from the implementation described by Fsix on their GitHub page. I have a given a link in the acknowledgements.

It is important to note that this technique does alter the scaling of the images and causes values to become negative. I modified the above version so that the deskewed images are rescaled in order to ensure shorter training times and more accurate classification.

Below, I have plotted the images to give an idea of how the deskewing process worked.

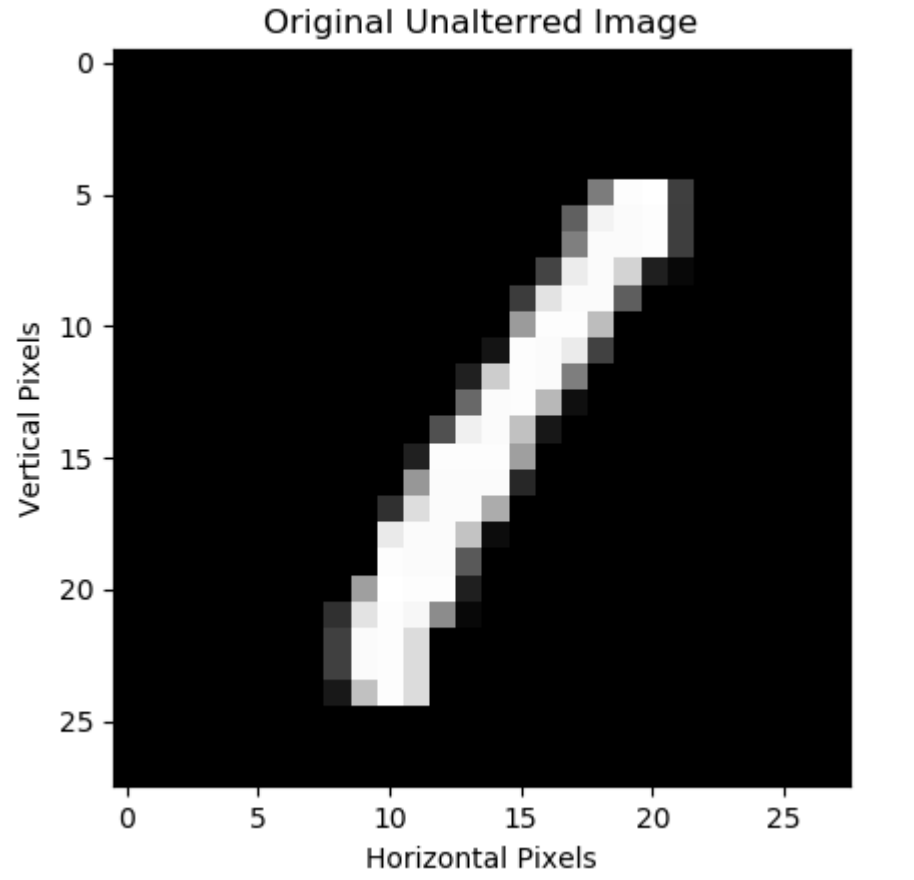

Figure 8 The digit 1. ColorMap Gray. Represented as given in MNIST. Sample number 4.

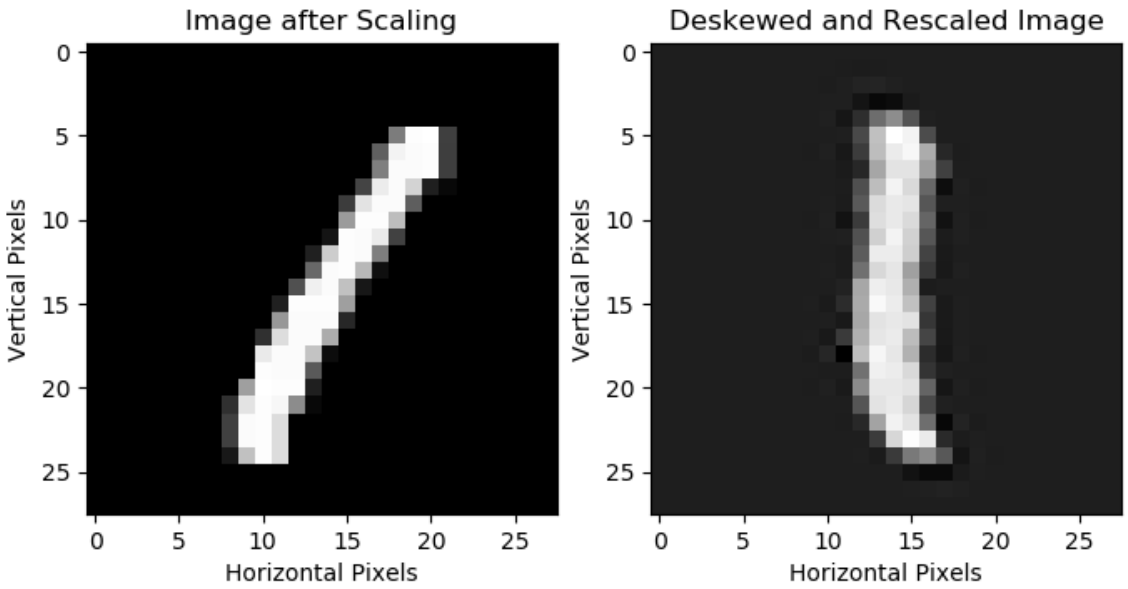

Figure 9 The digit 1. Sample number 4. ColorMap Gray. To the left after scaling pixel intensities between 0 and 1. To the right after deskewing and then rescaling the original image.

Here is another example for a different number (row 41495 in the training set).

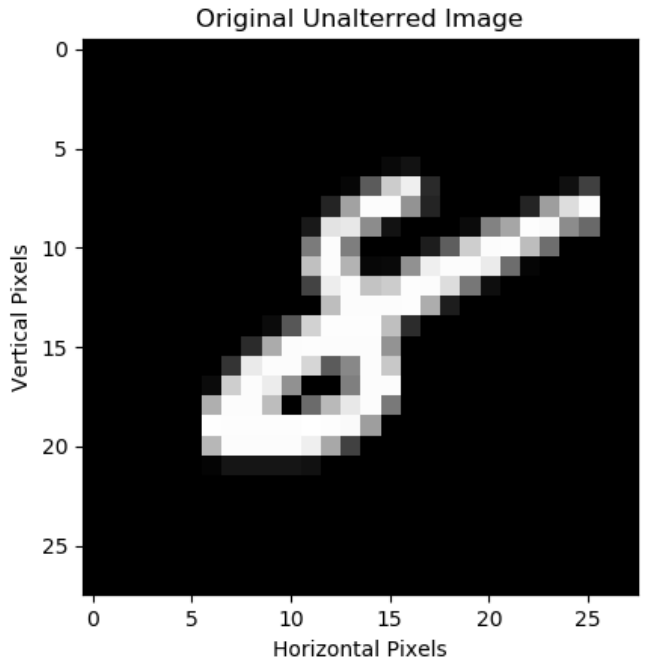

Figure 10 The digit 8. Sample 41493. ColorMap Gray. As Presented in MNIST.

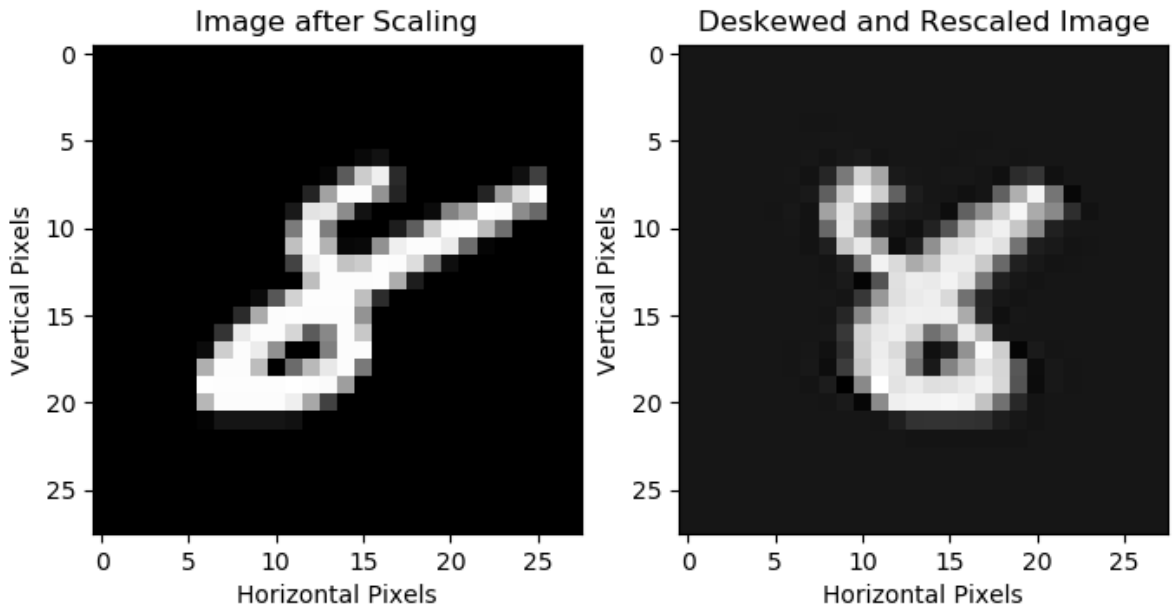

Figure 11 The digit 8. Sample number 41493. ColorMap Gray. To the left after scaling pixel intensities between 0 and 1. To the right after deskewing and then rescaling the original image.

Here, we can clearly see that the digit 8 is a lot more recognizable when it is vertically aligned rather than when it is written at a slant.

The table below gives the results of using these preprocessing techniques when performing GridSearchCV with 2 fold cross validation with C values (3.5,4,4.5) and Gamma (0.035,0.04,0.045). Thus 18 fits in total.

Preprocessing Technique	Original Time (minutes)	Final Time (minutes)	% Change
Pixel Removal	71.4	65.1	-8.8
Deskewing	71.3	39.2	-45.0
Pixel Removal and Deskewing	71.4	44.2	-38.1

Below, I have condensed the results of preprocessing on the processing times and the Test Result Accuracy into a table for the optimum model.

Model Parameters (kernel, C, gamma)	**Preprocessing**	**Time without Preprocessing** (Minutes)	**Time after Preprocessing** (minutes)	% **Change**	Accuracy on Test Set
RBF, 4, 0.04	Pixel Removal	7.1	8.5	19.71831	98.56%
RBF, 4, 0.04	Deskewing	7.1	5.8	-18.30986	99.01%
RBF 4, 0.04	Pixel Removal and Deskewing	7.1	8.1	14.08451	99.01%

Pixel Removal did lower the average training time during GridSearch when considering multiple C and Gamma parameters but for the most optimum hyperparameters, it increased the total training time.

From this again, I am led to conclude that deskewing on its own is the most effective preprocessing technique.

The preprocessing was very effective as we can see. Not only did training times go down by 28% on the most optimum model, but we also noted an improvement in accuracy from the 98.56% to 99.01%. It is of interest that

deskewing did alter the model's optimum hyperparameters. The C value of 4 remained the same but the gamma value shifted to 0.04.

This was the final model that I decided to proceed with. I also tried to look at the combination of pixel removal and image deskewing but pixel removal combined with image deskewing actually resulted in a slightly greater training time than just deskewing alone. Thus, deskewing alone is the most optimum form of preprocessing for the RBF kernel on the MNIST training datasets.

One Hot Encoding:

One Hot Encoding was a technique that I had come across during my research on convolutional neural networks. Essentially, what one hot encoding does is that it converts categorical variables to a one hot vector so that machine learning algorithms do a better job on it. What this means is that if we are using an algorithm like a neural network that has true multiclass classification abilities, it may deem the higher categorical value (9) to be better than the smaller categorical values such as zeros.

To get around this, we use one hot encoding where we convert each of the labels into a one hot vector. The vector has entries corresponding to each class. All of these entries are zero except for the entry corresponding to the class of the label, which becomes 1. There is near uniform consensus that the one hot encoding method improves classification accuracy on a wide variety of classifiers. Thus, the label 2 on the MNIST dataset would become [0 0 1 0 0 0 0 0 0 0].

This technique helped me improve the performance of the neural networks I had designed. The scikit-learn documentation seemed to suggest (I believe incorrectly) that one hot encoding could help my SVM as well. However, not only could multidimensional arguments for the labels not be passed as arguments for the .predict() method of the SVM classifiers but I also reasoned (as did the Professor) that this should not make intuitive sense. The ScikitLearn SVM achieved multiclass classification abilities by training a number of One Versus Rest classifiers (each classifier would say if a digit is 9 or not 9 for example) and thus one hot encoding should not help for this binary classification process. It would have been interesting to have compared the effects of one hot encoding for other classifiers as time permitted. I have seen online results that claim a 6-7% improvement using this process on multiclass classifiers but this is not always the case.

Final Classifier Metrics:

In the following table, I have provided the key metrics for the final classifier that I trained. All parameters that have not been specified are the default ones used by the SciKit Learn SVC() classifier.

Parameter	Value
Type	SVM
C	4
Gamma	0.04
Kernel	RBF
Preprocessing	Scale Images to between 0 and 1. Deskew Images as described in Preprocessing section
Training Time	4.3 minutes (!)

The astute reader will notice that this training time is different from that given above for these same parameters. The reason is that this training time was achieved without using the GridSearchCV object. All other training times had been reported by GridSearch CV, which was running tasks in parallel, so each individual task was slower.

The final model.dat file submitted was created on the ECELinux Cluster in order to ensure compatibility as some other students had reported compatibility issues.

Classification Report:

This is the classification report for the model that I generated using Scikit-Learn. It should help give a more holistic overview of the model performance.

Classes	Precision	Recall	F1-Score	Support
0	0.99	0.99	0.99	980
1	1.00	1.00	1.00	1135
2	0.99	0.99	0.99	1032
3	0.99	0.99	0.99	1010
4	0.99	0.99	0.99	982
5	0.99	0.99	0.99	892
6	0.99	0.99	0.99	958
7	0.99	0.99	0.99	1028
8	0.99	0.99	0.99	974
9	0.99	0.98	0.99	1009
Averages	0.99	0.99	0.99	
Total Support				10000

The metrics are computed as follows.

$$\text{Precision} = \frac{\text{tp}}{\text{tp} + \text{fp}}$$

$$\text{Recall} = \frac{\text{tp}}{\text{tp} + \text{fn}}$$

$$\text{F1} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

In the above equations, tp is the number of true positives for that class, fp is the number of false positives for that class, fn is the number of false negatives for that class.

The averages are weighted averages of precision, recall and f1-score where the weights are the supports.

The support is the number of samples of the true response that lie in that class.

It was interesting that the precision for Class 1 was 1.00. This would imply that were no false positives for class 1. To investigate this further, I decided to plot a confusion matrix for the test set results.

Confusion Matrix for the Final Classifier:

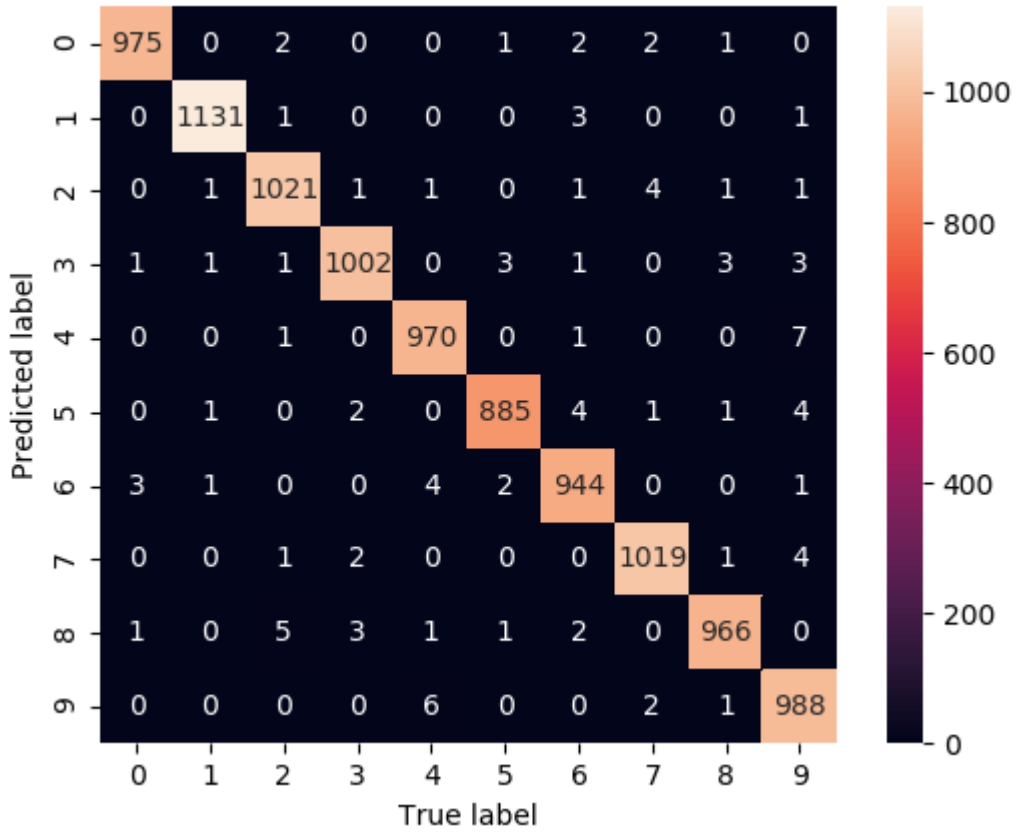

Figure 12 The confusion matrix for the final classifier. Generated using the Seaborn library's heatmap option.

The confusion matrix above summarizes the results of the classification report in a more intuitive manner. The Expected Labels for each sample are given on the horizontal axis. Interestingly, the vertical axis has the Predicted labels but these were plotted from top to bottom rather than bottom to top, as we would normally expect them. This is because confusion matrices are plotted so that the main diagonal (the non-black one shown above) contains all of the supports for that classifier. Where we have an intersection of a Predicted Label and a True Label for that same class, we have the total number of labels for that class that were correctly classified. The matrix has another advantage in that it also gives the number of samples that were misclassified for any given True Label and also tells us what those samples were misclassified as. For example, we know that for True Label 9 and Predicted Label 7, we get a value of 4. Thus 4 of the samples for digit 9 were misclassified as 4. Similarly, the sum of the column values gives you the total samples for that specific class.

Other statistics can be easily computed form the classification matrix. These are numerous and would take considerable space to enumerate here. I have enclosed calculated values for a few of these metrics for digits 0 and 9 within Appendix B.

Final Classifier Classification Breakdown by Class.

Class	Total Samples	Correctly Classified Samples	Misclassified Samples	Accuracy
0	980	975	5	99.490
1	1135	1131	4	99.648
2	1032	1021	11	98.934
3	1010	1002	8	99.208
4	982	970	12	98.778
5	892	885	7	99.215
6	958	944	14	98.539
7	1028	1019	9	99.125
8	974	966	8	99.179
9	1009	988	21	97.919
Overall	10000	9901	99	99.010

This shows that the precision of 1.00 for class 1 was due to rounding up as we can see that the classification accuracy is 99.648 for this class.

Below is an image that shows how the actual MNIST images and what the classifier predicted on them. These were randomly sampled from the MNIST test data.

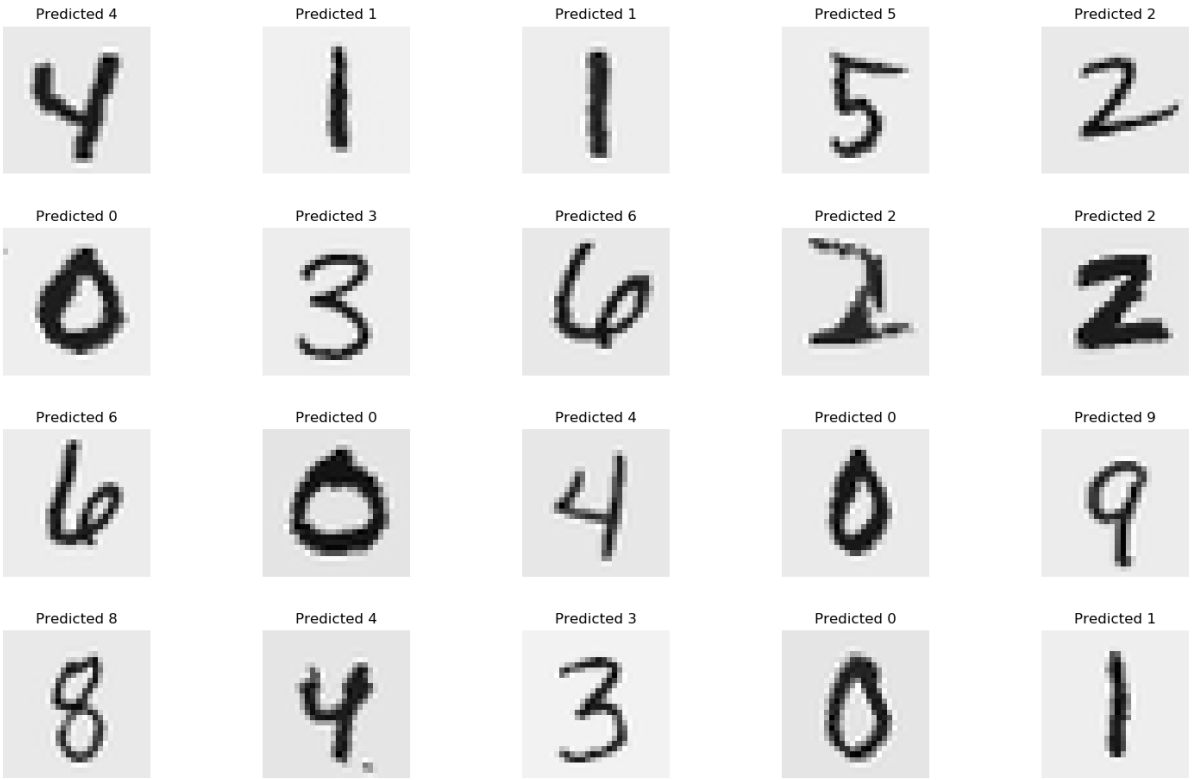

Figure 13 The predictions of the Final Model. Note Images have been deskewed and a different MatplotLib colormap was used here to display the digits.

Improvements:

While this is a good classifier (it beats any of the "tutorial" based machine learning classifiers and even the first convolutional neural network that I designed), it is not the best. I did have a few ideas for improving it.

Firstly, I am interested in ensemble learning where several machine learning algorithms can be combined together. There are different techniques involved such as bagging to decrease variance, boosting to decrease bias and stacking to improve classification accuracy. Here bagging refers to training multiple classifiers on subsets of the data and then computing the ensemble as follows:

$$f(x) = \frac{1}{N}\sum_{n=1}^{N} f_m(x)$$

The bagging technique would combine the outputs by voting for the classification results.

Boosting involves using an entire family of individually weak learners who look at weighted versions of the data in sequence. Higher weights are attached to data that was misclassified by earlier learners. A common algorithm for this is AdaBoost (Adaptive Boosting).

Finally, stacking involves combining different classification models using a meta-classifier, which takes as input the outputs of the base classification models.

Ensemble learning isn't necessarily better all the time. The training times for ensemble classifiers are also significantly longer, hindering network parameter tuning. Incorporating techniques for Feature Selection such as Principle Component Analysis can reduce training times but these tend to lower classification accuracy. As my preprocessing tests showed, even removing redundant pixel values does not speed up training times consistently. Testing would be involved. The figures I could find online for ensemble classifiers generally had accuracies around 97-98%. As the state-of-the-art classifiers I looked at did not seem to use ensemble learning, I decided to forgo this technique but it should be an interesting point of exploration, given what I have learned to date.

A far more interesting regime than this is adversarial training. The MNIST dataset is fairly clean but within the real world, if this classifier was ever deployed, it would have to deal with all sorts of noise in each image. One of the techniques to get around this would be perturbing the training images. This means that we deliberately introduce random noise, rotations, skews and distortions into the dataset so that it performs well on test data that also contains similar noise and distortions. This is a more specialized technique for improving real world classification performance by generating "jittered' images and is also what was leveraged for creating the state-of-the-art SVM classifiers for MNIST. This would be a worthy area of exploration.

As an aside, perhaps future iterations of this course could have the Professor request Google Cloud credits (it is free, you have an application to fill out) so that they can get to develop more intensive classifiers and also learn gain experience with offloading computations to the cloud.

Conclusion:

My Drive folder for this project has more than 20 GB of data due to the number of tests that I ran on different SVMs, neural networks and other classifiers on the machines in Carpenter Hall.

To give an idea of how I managed to achieve that, see below for an image of Carpenter Hall in the day.

Figure 14 Carpenter Hall in the day. Regular students doing regular work. Except for one who is watching a basketball match.

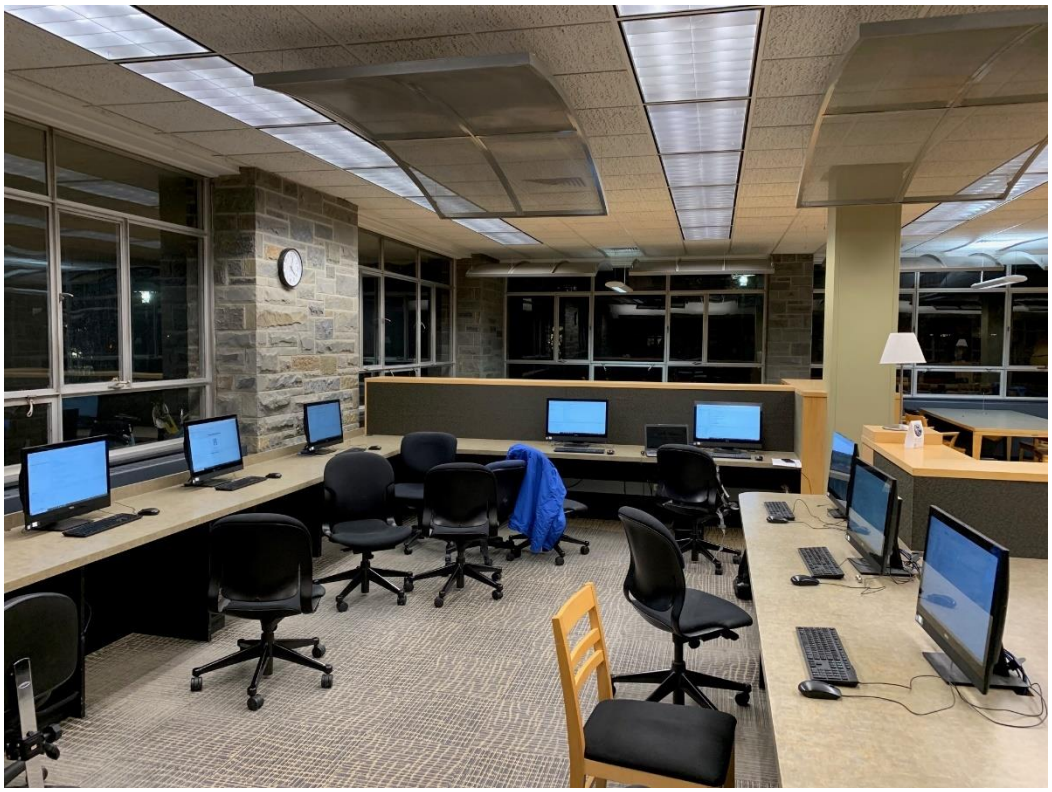

Figure 15 All 8 PCs are running GridSearchCV or testing Preprocessing Algorithms.

Throughout this project, I approached a large number of classifiers. I liked that it was open ended and, ultimately, the ECELinux cluster's limitations compelled me to learn more about hyperparameter tuning and preprocessing rather than relying on hitherto-not-fully-understood *Deep Learning Magic*^{TM} to do the work for me.

The final classifier I made beats all SVMs in classification accuracy that I could find with the exception of one that was leveraging adversarial training techniques and used the 9 degree polynomial kernel (which would require an insane amount of training time). In any case, the classifier is decent and a long shot away from the 91% classification accuracy that I achieved by running 5 lines of code for 3 minutes using the LinearSVC() algorithm.

It was a good project.

Appendix A, Partial Content on Convolutional Neural Networks:

This is the portion of the report that I had originally written for the Convolutional Neural Networks that I intended to leverage for model classification at the start. This is not conclusive like the section above. It has been included only to give an idea of the different considerations that went into the Neural Network design.

Library Selection:

When it came to library selection, I opted for Keras. The reason is simple. Tensorflow and Theanos (even though it is no longer in active development) are both difficult to work with. Keras is cleaner and provided a High level Abstraction of both that could be used to rapidly build and deploy neural networks. Another advantage of Keras was that its backend could be changed to switch between TensorFlow, Theanos and CNTK (a Microsoft toolkit). Each of these provides different speed and accuracy advantages for different neural network types, which provided an additional point of exploration.

Backend:

For my Backend, I disregarded Theanos as it was not in active development. When it came to choosing between CNTK and TensorFlow, I looked at different benchmarks online. I found that TensorFlow was, in general, both faster and more accurate than CTNK for convolutional neural networks. For the multilayer perceptron models, CNTK was faster but had slightly lower accuracy. As accuracy was weighed more heavily by me and CNNs would be my final design, I went with TensorFlow. Some of the benchmarks are viewable here.

https://minimaxir.com/2017/06/keras-cntk/

Initial Network:

To get started, I began looking at perceptrons. These are similar to neural networks as the perceptron is a single layer neural network at essence. However, as it is only a binary classifier, I would require something more advanced. This led me to the multilayer perceptron model.

I designed a basic multilayer perceptron model.

Multilayer Perceptron Model had an error of 1.74%.

Speed Considerations:

I had already selected a good library in terms of speed (and the best in terms of accuracy when it came to deciding the Keras Backend). To further speed up training time, I decided to opt for a relatively small number of Epochs (10) in order to get my network trained faster. Epochs are how many times the algorithm gets to see the entire training data (a complete forward and backward pass through the neural network). Higher Epochs generally correlate well with better classification accuracy at the expense of longer training times.

I had installed the default tensor flow build that was available by using pip. However, I was getting the following error.

tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2

This led me to conduct research into AVX2. I came to discover that AVX 2 stood for Advanced Vector Instructions. Basically, these are extensions to the x86 Instruction Set Architecture (used by Intel and AMD for their processors). Essentially, AVX 2 allows you to carry out most operations on 256 bit wide integers and also allows for FMA (fused multiply-accumulate operations). It is an operation in which we compute the product of 2 numbers and add the product to a register that tracks the count with a single rounding applied to the result.

Benchmarks suggested that AVX2 could speed up training times for my neural network by 40%.

Compiling this would have taken a few hours away from me. Luckily, I found a precompiled binary on Github.

Downloading and installing this version proved to be a significant hassle. This version was less up to date than the one I had installed previously, which lacked support for AVX instructions. As a result. For one, all of this had to be done in the terminal. For another, Anaconda had a bug that prevented it from installing properly. As a result, I was forced to reinstall Python, then create a new virtual environment. Within this, I first installed the tensorflow.whl file for Python 3.7 and then proceeded to test my multilayer perceptron model again.

https://medium.com/@sometimescasey/building-tensorflow-from-source-for-sse-avx-fma-instructions-worth-the-effort-fbda4e30eec3

This suggested that if I was using the CPU alone, I could achieve a 40% faster training speed. Note, that while my laptop did have an Nvidia GPU, the 820m is weak and does not have support for Nvidia Compute Capability 3.0, which is required for GPU-powered training on TensorFlow.

https://developer.nvidia.com/cuda-gpus

It is mildly irking that a single step up (from the 820m to the 830m) would have allowed for GPU training of the CNN, which would have dramatically cut down on the training times. CPUs it is for now.

Activation Function:

Essentially, the activation function helps the neural network model non-linear relationships. The reason is that for each node of neural network, we are taking the product of the Inputs and the Weights and feeding it to the next layer of neurons. This is a linear operation and the linear function is a one degree polynomial. To model non-linear relationships, the activation function takes the product of weights and Inputs as an input itself, processes it and then this processed value is passed on to the next layer.

The ReLU (Rectified Linear Units) activation function was selected. There are a few other popular options. These include the sigmoid function, the tanh function and the Relu Function. I intended to explore the effects of each of these and the differences in training times and classification accuracy.

https://towardsdatascience.com/exploring-activation-functions-for-neural-networks-73498da59b02

Choosing the optimizer:

The optimizer I decided to proceed with was Adam. This is not an abbreviation, the word itself comes from adaptive moment estimation. In essence, stochastic gradient descent maintains a constant learning rate for each individual parameter or weight within the network. Adam actually combines the unique advantages of two other extensions to stochastic gradient descent. These are the Adaptive Gradient Algorithm (AdaGrad) that maints a per-parameter learning rate and results in improved performance on problems with sparse gradients (this is of interest to us as sparse gradients are often encountered in machine learning problems involving natural language and computer vision aka MNIST). The second algorithm is Root Mean Square Propagation (RMSProp) that adapts the individual learning rate for a parameter by averaging the recent magnitudes of the gradient for that parameter. This helps it perform well on problems with significant noise. Adams further extends on this by taking the average of the second moments of the gradients (the uncentered variances).

https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Of more concrete interest is that Adam, which was introduced back in 2015, is that ADAM is fast. Sebastian Ruder tested it for a multilayer convolutional neural network that accompanied dropout (more on this later) and found that ADAM was generally the best optimization algorithm for a wide variety of use cases, edging out RMSProp in terms of raw speed.

http://ruder.io/optimizing-gradient-descent/index.html

Salient CNN Features:

- Convolutional layer: Improves classification Accuracy
- Pooling: Reduces training times
- Dropout: Improves classification accuracy

Second Neural Network Results:

```
Epoch 8/10
 - 163s - loss: 0.0190 - acc: 0.9940 - val_loss: 0.0327 - val_acc: 0.9890
Epoch 9/10
 - 165s - loss: 0.0156 - acc: 0.9950 - val_loss: 0.0324 - val_acc: 0.9888
Epoch 10/10
 - 157s - loss: 0.0143 - acc: 0.9958 - val_loss: 0.0328 - val_acc: 0.9891
CNN Error: 1.09%
```

It is an improvement over the 1.74% error of the multilayer perceptron classifier.

Appendix B, Additional Statistics for the Final Classifier Using pandas_ml:

Classes	0	...	9
Population	10000	...	10000
P: Condition positive	980	...	1009
N: Condition negative	9020	...	8991
Test outcome positive	983	...	997
Test outcome negative	9017	...	9003
TP: True Positive	975	...	988
TN: True Negative	9012	...	8982
FP: False Positive	8	...	9
FN: False Negative	5	...	21
TPR: (Sensitivity, hit rate, recall)	0.994898	...	0.979187
TNR=SPC: (Specificity)	0.999113	...	0.998999
PPV: Pos Pred Value (Precision)	0.991862	...	0.990973
NPV: Neg Pred Value	0.999445	...	0.997667
FPR: False-out	0.000886918	...	0.001001
FDR: False Discovery Rate	0.00813835	...	0.00902708
FNR: Miss Rate	0.00510204	...	0.0208127
ACC: Accuracy	0.9987	...	0.997
F1 score	0.993377	...	0.985045
MCC: Matthews correlation coefficient	0.992658	...	0.983399
Informedness	0.994011	...	0.978186
Markedness	0.991307	...	0.98864
Prevalence	0.098	...	0.1009
LR+: Positive likelihood ratio	1121.75	...	978.208
LR-: Negative likelihood ratio	0.00510657	...	0.0208335
DOR: Diagnostic odds ratio	219668	...	46953.5
FOR: False omission rate	0.000554508	...	0.00233256

Figure 16 A few of the statistics that can be generated from the confusion matrix of the final classifier. The pandas_ml library was used to quickly compute these.

Appendix C, Bar Chart showing how the dataset is balance.

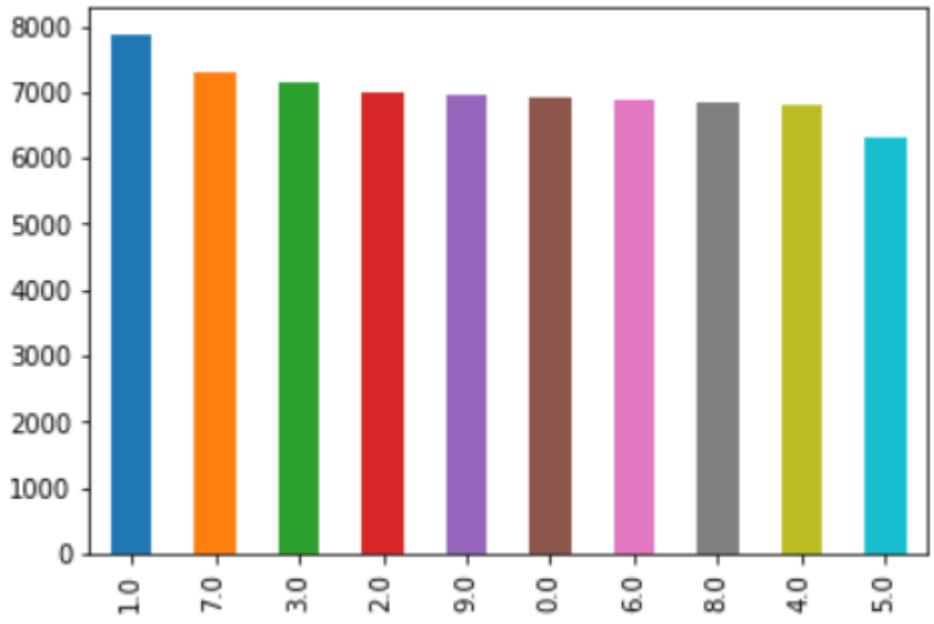

Figure X Showing the distribution of digits within MNIST. The dataset is well balanced. The Y Axis has sample count. The X axis has the classes. The title of this bar chart should be "Sample Count versus Classes for the full MNIST training dataset. I understand that axis labels and the title were missing so I moved it here. Why not plot these? I was using a Jupiter notebook to get nicer formatting and it was not

cooperating with me as my Anaconda Navigator decided to die. A reinstall was not possible as my Project depends on my Anaconda installation.

Acknowledgements:

There were 5 resources that were instrumental in this work, apart from the Scikit learn and Keras documentation available online.

MNIST Dataset Top Results:
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354

Useful GridSearch Tutorial along with SVM parameters:
https://github.com/ksopyla/svm_mnist_digit_classification

Saving and Loading Classifiers:

https://machinelearningmastery.com/save-load-machine-learning-models-python-scikit-learn/

Removing Duplicate Pixels:

https://www.kaggle.com/damienbeneschi/mnist-eda-preprocessing-classifiers
Deskewing:

https://fsix.github.io/mnist/Deskewing.html

Creating HeatMaps:

https://jakevdp.github.io/PythonDataScienceHandbook/05.07-support-vector-machines.html

Talk about data shuffling:

http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

Ensemble Techniques:

https://blog.statsbot.co/ensemble-learning-d1dcd548e936

One hot Encoding Process.

https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f

Other references:

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-159.pdf

https://www.youtube.com/watch?v=71iXeuKFcQM

https://stats.stackexchange.com/questions/70801/how-to-normalize-data-to-0-1-range

https://scikit-learn.org/stable/modules/svm.html#multi-class-classification

https://scikit-learn.org/stable/modules/svm.html#multi-class-classification
http://www.cs.nthu.edu.tw/~shwu/courses/ml/labs/07_SVM_Pipeline/07_SVM_Pipeline.html
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelBinarizer.html
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html
https://medium.com/@kasiarachuta/basic-statistics-in-pandas-dataframe-594208074f85
http://yann.lecun.com/exdb/publis/pdf/lecun-95b.pdf

https://www.r-bloggers.com/exploring-handwritten-digit-classification-a-tidy-analysis-of-the-mnist-dataset/
https://stackoverflow.com/questions/43577665/deskew-mnist-images
https://stackoverflow.com/questions/20763012/creating-a-pandas-dataframe-from-a-numpy-array-how-do-i-specify-the-index-colum
https://martin-thoma.com/svm-with-sklearn/
https://www.quora.com/Why-does-RBF-kernel-generally-outperforms-linear-or-polynomial-kernels