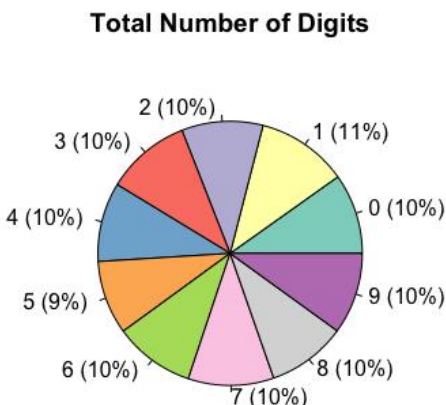


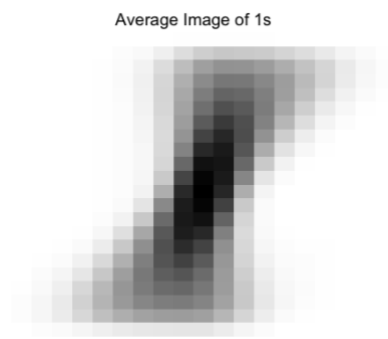
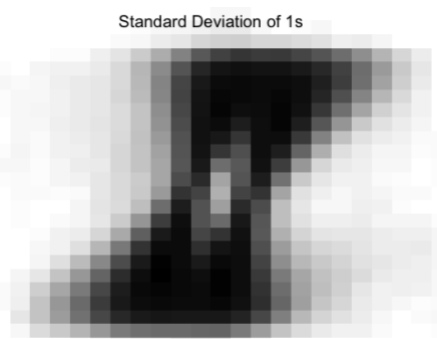
Digit Recognizer

The goal of this analysis is to use machine learning techniques to identify handwritten digits. I explored which algorithms would be the most efficient considering execution time and accuracy. The training data set used in this analysis came from a Kaggle competition and can be found in the data folder of this project.

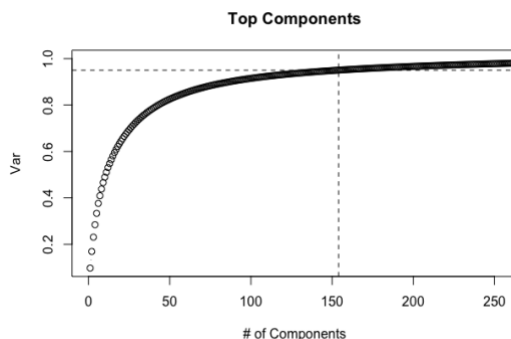
First, the source.R script was opened and the working directory was set. The functions and packages needed for the analysis were then initialized. The data was then loaded and explored using the explore.R script. The data has 42,000 observations and 785 columns. Here are the percentages of how often each digit appears in the data:



The columns provide data regarding the shading of each pixel with the darker ones having a higher value. Here are some sample images from the observations:



The next step was to run the PCA.R script which performs Principal Component Analysis (PCA) on the data. The data was standardized by dividing all pixel value columns by the max value of all pixels. Pixels ranged from 0-255 and by standardizing it, they range from 0-1. The second step is to compute the covariance matrix, then the eigenvectors and eigenvalues of the covariance matrix was computed to identify the principle components. I looked at the scree plot and the cumulative variance plot.

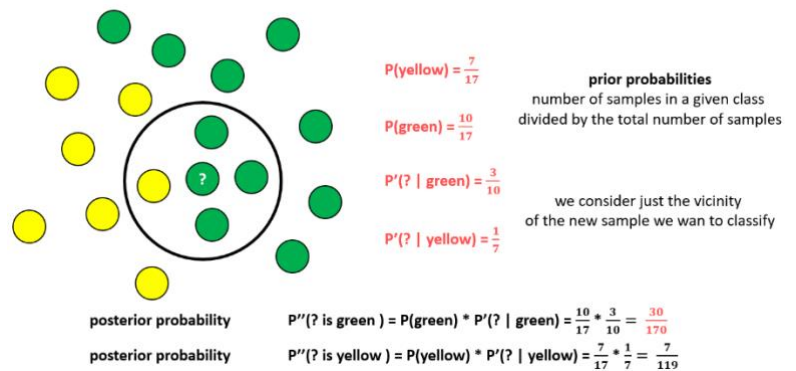
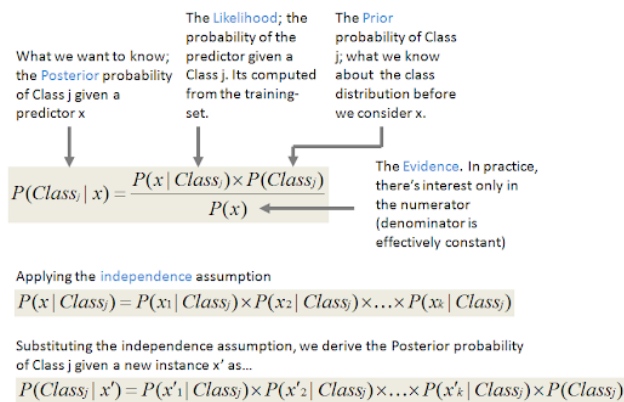


When the Cumulative Variance Accounted For (CVAF) percentage is 95%, the number of components is 154. I matrix multiplied the training data set (not including labels) by the first 154 eigenvectors and created a new variable for the labels of the training data. The data was split into two data frames, a training set (75% of the total data set) and a validating data set (the remaining 25%). The training data set was used to build a classifying algorithm that can accurately predict the validating data set and then the predicted results were compared against the actual results of the validating set. Next, I looked at the sample sizes of the two data sets to make sure there were enough in each label.



I ran the machine learning algorithms, Naïve Bayes (NB), Neural Networks (NN), Support-Vector Machines (SVM), K-Nearest Neighbors (K-NN), Random Forests (RF), and Quadratic Discriminant Analysis (QDA). There are many others that can be explored as well.

Naïve Bayes:



These 2 diagrams give really simple explanations about how the Naïve Bayes classifier works.

$$\text{Bayes Rule: } P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

x is the pixel values of the image and c is the digit label (0-9). $P(c|x)$, the posterior, is the probability that the label is c given the data x . $P(x|c)$, the likelihood, is the probability that the data x belongs to the label c . $p(c)$ = number of times a label is present in the data / number of images. I use the naiveBayes function in R to calculate the predictions. Naïve Bayes assumes all pixels are independent. If I look at a dark pixel, are the pixels around it going to be dark? Probably. Therefore, since the naïve assumption is not correct, this method might not perform so well.

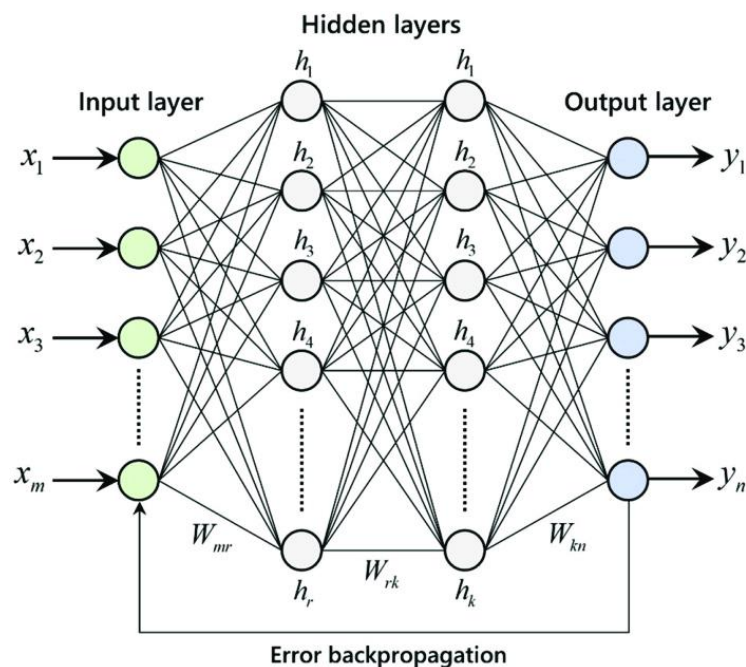
Results: the algorithm ran in approximately 71s with about an 85.56% accuracy rate in predicting the validating data set. This is not great but gives a good baseline to compare other algorithms against. Here are some of the misclassified digits: (green-correct, red-incorrect prediction)

Naive Bayes Misclassifications

3 3	6 4 4	2 9 9	2
9 9	7 9 9	4 7 7	8
7 7	9 6 6	2 5 5	8

Neural Networks:

Here is a simple diagram and explanation of how neural networks work:



I started with the training dataset. Look at one observation. It is composed of $28 \times 28 = 784$ pixels. After PCA I had 154 important components. The input layer is made up of 154 neurons. Each pixel is fed as input to each neuron of this layer. Neurons of one layer are connected to the next layer through channels. Each channel is assigned a numerical weight. Weights are multiplied by the input and the sum(bias) is sent to the neurons in the hidden layer.

$$h_1 = (x_1 * W_{11}) + (x_2 * W_{21}) + (x_3 * W_{31}) + (x_m * W_{m1})$$










It takes the weights and multiplies by the input, add this bias to it and passes it through a threshold/activation function. This determines if the particular neuron will get activated. Activated neurons transmit data to the next layer over the channels. This forward flow of data transmission is called forward propagation.

During the training phase, the output layer gives probabilities of what observation should be classified as. The one with the largest probability is what the observation ultimately gets classified as. The predicted output is then compared against the actual to realize the error. This information is then fed backwards through the network called back propagation. Based on this information the weights are adjusted. The cycle of forward and back propagation is iteratively performed with multiple inputs till the max iterations is reached or the network can accurately predict the digit. This ends the training process and the network is run against the test/validating set to see how well it performed.

In general, neural networks are complex and slow to train. I decided to use the nnet function in R to calculate the neural network, because it was simple and easy to use. I could use the neuralnet function in the MASS package if I needed more flexibility in defining parameters. Neuralnet allows you to use the hyperbolic tangent function and the back-propagation algorithm for training which is more common and better. Nnet uses the BFGS algorithm to find internal weight constants and the logistic sigmoid function to introduce non-linearity after a layer computation.

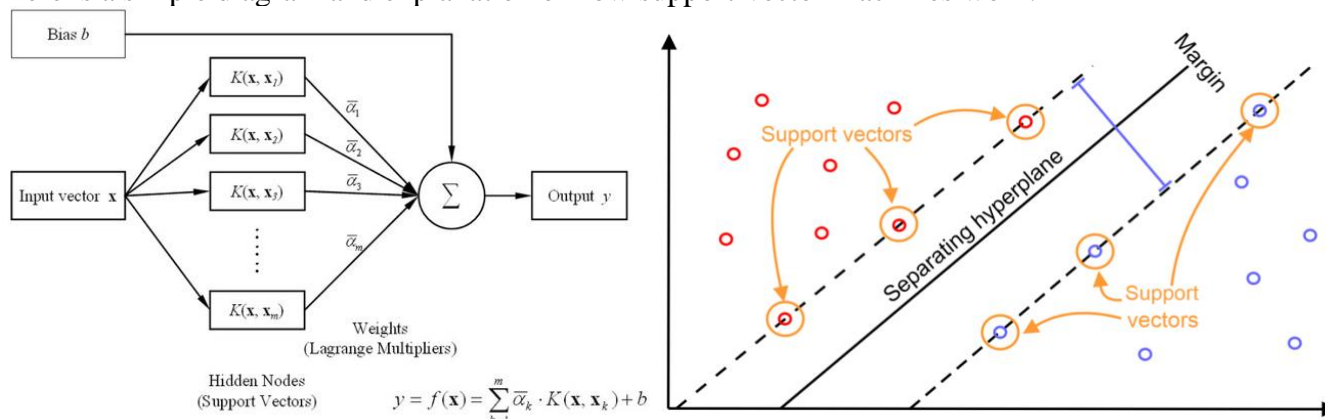
Results: the nnet algorithm ran in approximately 2,308s with about a 96.98% accuracy rate in predicting the validating data set. This is very accurate but runs very slowly, depending on how important the tradeoff is between time and accuracy maybe I should not use this method. Here are some of the misclassified digits: (green-correct, red-incorrect prediction)

NN Misclassifications

8		6 9		0 7		1
3		8 9		7 7		8
5		0 7		9 2		4

SUPPORT-VECTOR MACHINES:







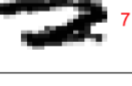


Here is a simple diagram and explanation of how support-vector machines work:



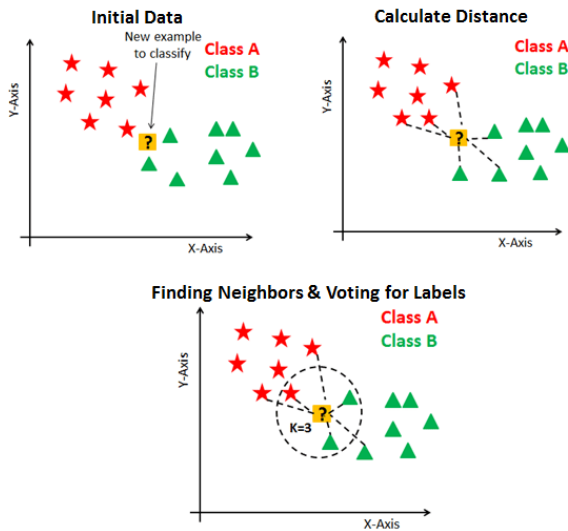
SVMs work somewhat similarly to Neural Networks demonstrated in the left image above. It takes a sample of input features, and outputs weights for each feature whose combination predicts the value of y , however, SVMs have a stronger mathematical foundation than neural networks. SVMs try to construct a linear or non-linear separating hyperplane in n dimensional space to separate the classes with the largest margin possible. This optimization is used to reduce the number of weights to just the features that are important for deciding the hyperplane. These weights are the support vectors because they support the separating hyperplane.

Results: the svm algorithm ran in approximately 527s with about a 97.06% accuracy rate in predicting the validating data set. This is almost as accurate as the neural networks but runs about 4 times faster. Here are some of the misclassified digits: (green-correct, red-incorrect prediction)

SVM Misclassifications

5		3 1		4 2		8
5		3 8		3 3		8
2		7 7		9 3		9

K-Nearest Neighbors:



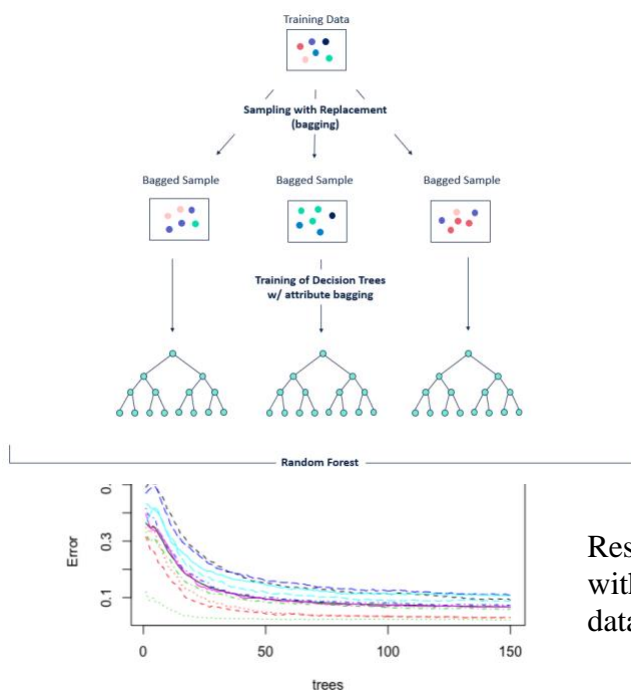
The way this algorithm works is that for each row of the test set, the nearest k neighbors from the training data set (using Euclidean distance) are found. Classification of the test set is then decided by majority vote, with ties broken at random. For this example, I chose K to be the floor of the square root of number of rows of the training set. K can be adjusted and one that has the best accuracy rate should be chosen. A small k means that noise will have a higher influence and a large k will take more processing time. Using the square root of the number of rows is a good place to start.

Results: the knn algorithm ran in approximately 103s with about a 91.95% accuracy rate in predicting the validating data set. This is not nearly as accurate as SVMs but about 5 times faster. Here are some of the misclassified digits: (green-correct, red-incorrect prediction)

KNN Misclassifications

7		9		6		1	
8		1		9		7	
0		6		7		8	

Random Forest:









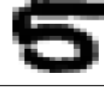


Random Forests use fundamental principles from decision trees and bagging. It builds a large collection of decorrelated decision trees to improve performance. Bagging uses a random component during the tree growing process building many trees on bootstrapped copies of the training data, which limits tree correlation, it then aggregates the predictions across trees. This reduces variance and improves prediction accuracy. I ran the algorithm with 300 trees and looked at the Out-of-Bag Error rate vs number of trees to pick how many trees should be in my random forest. More trees provide stable error estimates but increase computation time. I decided to go with 150 trees as it had similar accuracy rate and ran much faster than 300.

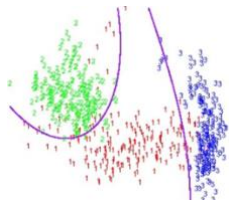
Results: the randomForest algorithm ran in approximately 101s with about a 94.55% accuracy rate in predicting the validating data set. This is slightly more accurate than the KNN algorithm

and takes about the same amount of time. Here are some of the misclassified digits: (green-correct, red-incorrect prediction)

Random Forest Misclassifications

4		7 8		5 4		2
9		7 2		9 6		5
8		5 9		7 5		8

Quadratic Discriminant Analysis:











Discriminant analysis is used to determine which variables discriminate between 2 or more naturally occurring groups. It is assumed that the measurements are normally distributed. This method allows for non-linear separation of data and there is no assumption that there are identical covariances of each class, therefore the covariance matrix is estimated separately for each class.

Quadratic discriminant function: $\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1}(x - \mu_k) + \log\pi_k$

Results: I used the qda function in R to classify the data then predict the validating data set classifications. The algorithm took about 10 seconds to run and was about 94.89% accurate. That's pretty good especially at that speed! Here are some of the misclassified digits: (green-correct, red-incorrect prediction)

QDA Misclassifications

4		8 1		2 9		0
7		9 9		0 7		3
3		2 9		8 8		1

Summary:

Method	Time	Accuracy
NB	71s	85.56%
NN	2308s	96.98%
SVM	527s	97.07%
KNN	103s	91.95%
RF	101s	94.55%
QDA	10s	94.89%

If I am looking for just accuracy, I might dig deeper into neural networks and adjust the parameters to more accurately predict the digits and I can expect that to be very time consuming. If I don't want to spend as much time but accuracy is still important, I would go with the SVMs. If time is a concern, I would use the QDA method.