Machine Learning - CS 7641
Assignment 1

Anthony Menninger
Georgia Tech OMSCS Program
tmenninger@gatech.edu

Abstract

This paper reviews five classifying algorithms with two different datasets. The algorithms reviewed are Decision Trees, Neural Networks, Boosting, Support Vector Machines (SVM), and K Nearest Neighbors (KNN). The first dataset is a 1994-1995 Census Data set with an income threshold label. The second is a set of hand drawn digit images with the actual digit labels. The tunings and performance are reviewed for each dataset and algorithm. The report concludes with a review of the relative strengths and weaknesses of the algorithms for the datasets.

Introduction

Data Sets

The first data set is from 1994 and 1995 current population surveys conducted by the U.S. Census Bureau [1] from the UC Irvine Machine learning site. The classification task is to determine if the user type for each instance has an income above $50,000 or below ($94,000 in today's dollars). The 1994 median annual salary was $16,000 (2020 median salary was $34,600)

There are 40 usable features, with both continuous features, such as age, and discrete, such as married status or sex. The training set has 199,523 instances and the test set has 99,762 instances. Because many of the features in the file are string based, I created transformed instances using the sklearn preprocessing LabelEncoder to translate discrete string values into consistent numeric values. This was necessary for algorithms that only used numeric features. I chose this data set because it seemed relatively large and I expect somewhat noisy with a smallish number of features.

The second data set is The MNIST Database of Handwritten Digits [2]. This consists of instances of 28 X 28 greyscale images of the digits 0-9. One transformation performed was that the values where scaled to a real value between 0 to 1, from an integer between 0 to 255. For all but the neural network algorithm, the images were flattened, creating 784 features, one for each pixel. There are 60,000 training instances and 10,000 test instances. I chose this because it is a good comparison to the first data set, with a very different type of data (images), which leads to significantly more features (784). In addition, each of the features can be thought of as related to each other, as they are all positional in a two dimensional grid, while the first data set features do not have any necessary relation between themselves ie: age is not related to sex.

Decision Trees

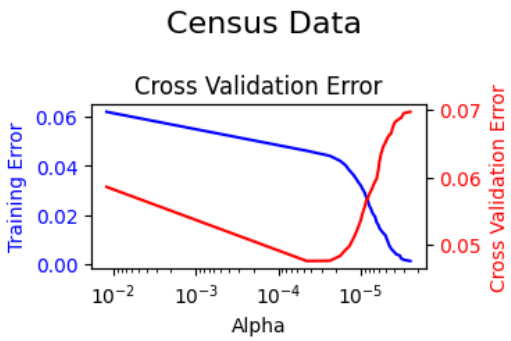

Figure 1: Census Data Cross Validation. The cross validation error starts to rise as the decision tree starts to over fit. The best cross validation α value of 0.0000153 had a test accuracy of 95.24%, which was very close to the actual best testing score of 95.30%.

I used the DecisionTreeClassifier algorithm from the sklearn library. By default, this uses a GINI index to split the data. It was possible to use an information gain entropy for splitting, but this did not make a meaningful difference in results for the datasets so the GINI index was used. A key aspect of sklearn is that it only accepts numeric data for features and treats all features as continuous. Because of this, a feature can appear in multiple nodes within the same path, with different thresholds. It also means this is a binary tree, with each node having only two edges. Other decision tree implementations might allow for discreet features, meaning a feature could only appear once in any tree path and the tree might not be binary.

At first I thought this would make the tree more complex, but I think there is a reasonable tradeoff. A tree that always creates a node for all discrete values will always create the maximum amount of nodes but only adds one level of depth to the tree. The sklearn continous tree may not need to create

nodes for every value, but it will end up with greater depth if more than one node needs to be created.

sklearn also uses a randomizing element, which means that given the same set of data, it may produce different results each run. In order to produce consistent, repeatable results, the random state setting was set to a fixed value to produce the same results each time.

I used the sklearn Cross Validation module for estimating the best solution without using the test data as described in more detail below.

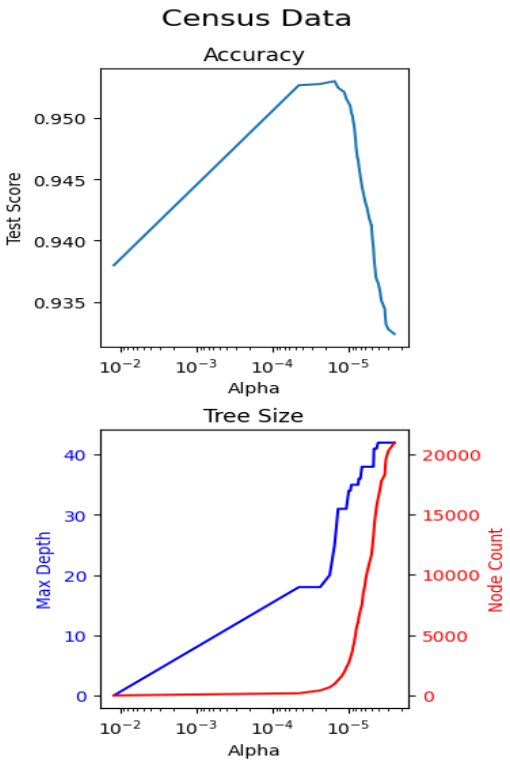

Figure 2: Census Data Decision Tree: The right side of the upper chart shows overfitting as the pruning is reduced and the number of nodes and tree depth increases. An unusual aspect is the left side of the chart shows very high accuracy with very few nodes. This stems from one feature being highly correlated to the labeled output.

To prune the tree, sklearn uses a minimal cost-complexity setting. Each node in a tree has an α value which measures the node misclassifications minus leaves misclassifications divided by the number of leaf nodes. The DecisionTreeClassifier then takes an α minimum setting, pruning all subtrees with a lower α. For both datasets, an accuracy curve was created by fitting the tree with different α values.

For both data sets, I also performed cross validation to use the training set only to determine the best α parameter, with a 5 fold setting (80% of training data used for training, 20% used for validation). I would then use the test data at different α values to confirm the finding.

For the Census Data, **Figure 1** shows the use of cross validation to estimate the best solution, in this case setting for α, without using the test data. The best cross validation α value of 0.0000153 had a test accuracy of 95.24%, which was very close to the actual best testing score of 95.30%. Cross Validation provides a powerful tool for tuning an algorithm on a given training set without reference to test data.

The Census Data, shown in **Figure 2**, shows accuracy improves to a point as nodes are added, then accuracy declines due to over fitting as more nodes are added. Alpha, the pruning parameter, leaves more nodes in the tree as it is decreased. The Census Data also showed a specific characteristic. One of its features (Capital Losses), had a very high correlation with the labeled output, Income above $50k. Thus, a one node tree already showed 93% accuracy. Figure 2 shows this, with the the left side of the chart with few nodes already having a very high accuracy. The Census dataset is skewed, with only about 6% of the instances having a positive label of above $50k.

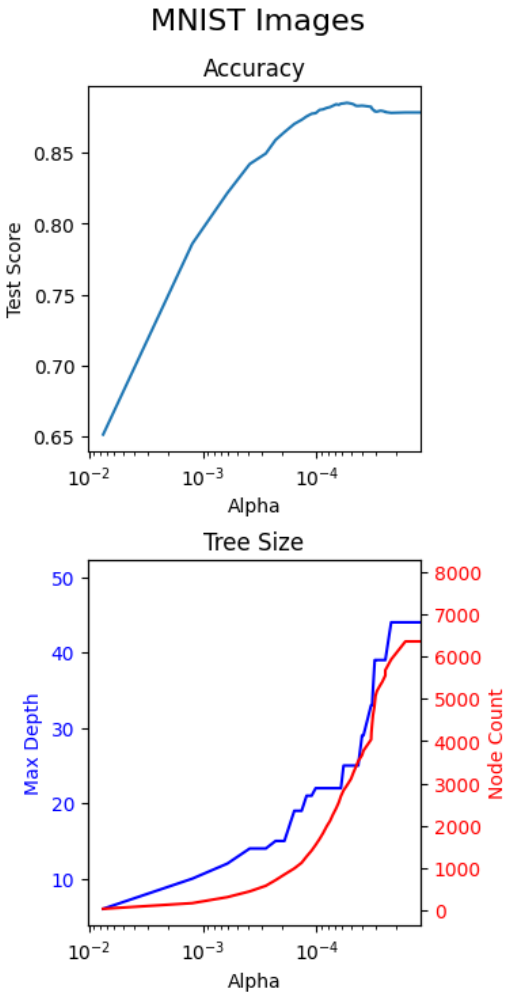

Figure 3: MNIST .

The MNIST image data, shown in **Figure 3**, showed a more normal accuracy curve than the Census Data, with accuracy improving dramatically with added nodes until a

maximum accuracy was reached, then accuracy declines due to over fitting.

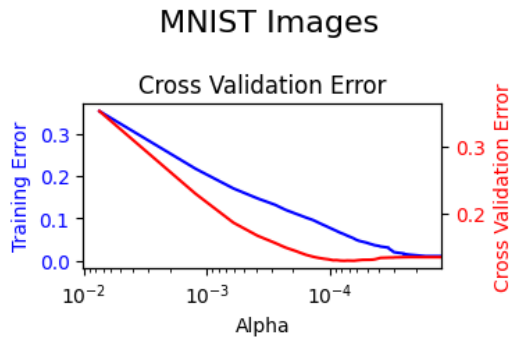

Figure 4: MNIST Images Cross Validation Error: The chart shows continual reduction in training error, while the Cross Validation error reaches a minimum at an α of 0.0000542 and then starts to rise. This was very close to the best α found using the testing data.

Figure 4 shows the cross validation error curves for the MNIST image data. The best cross validation α value of 0.0000762 had a test accuracy of 87.18%, while the actual best testing score of 88.54%. This was a larger gap than found in the Census data, but still relatively close. The maximum accuracy achieved was 88.7%, which was significantly below the census data. A decision tree is looking at one pixel at a time, so small variations in position or size would be very challenging for this algorithm. I expect other classifiers, especially the neural network, to better handle image data.

Neural Networks

For the neural network, I used PyTorch with two hidden layers, the Adam optimizer and Linear layers. I then adjusted the size of the layer, the learning rate, and epoch cycles to find the optimal settings. I used mean squared error loss, which I don't think is a great fit for a binary classification, but had significant trouble getting that type of model to work.

In just the initial setup for the Census data, I quickly determined I would need to scale the data as the solver would often "blow up" with really large values. I used the sklearn StandardScaler module, which removes the mean from each feature then applies a sigmoid function to create values from -1 to 1. This is somewhat suspect for the categorical features, as there is not really a positional relationship, but it was still effective. I also changed my cross validation process. With the small number of positive lables, as discussed in the decision tree section, I used a 4 folds instead of 5. This also speeded up processing, by reducing the number of cross validation splits by 20%. In addition, I created my own cross validation process instead of using the sklearn cross validation tool. Partly this was done because of my custom neural network, but it also allowed me to use multiprocessing to speed up the process.

Figure 5 clearly shows the preference for the smaller layer size and smaller learning rate, although the differences

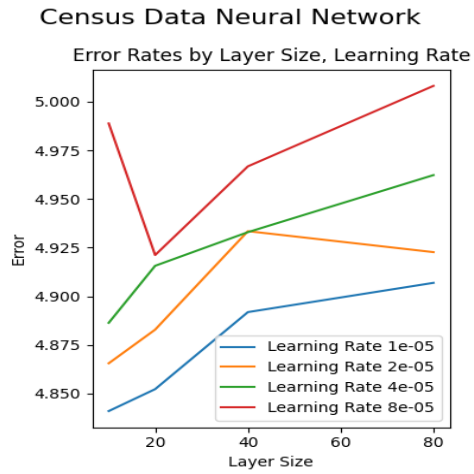

Figure 5: Census Data Neural Network: Built using PyTorch, linear functions and two hidden layers. The lowest error is clearly shown for the smaller learning rate with the smaller network.

in absolute accuracy are quite small. This is a victory still due to the very skewed dataset, with only 6% of the data being labeled positively. The smaller layer size performance is straight out of the class lecture with regard to preferring simpler solutions. I had expected larger layer sizes to be more effective. The charts indicates that further exploration at the lower end was worthwhile.

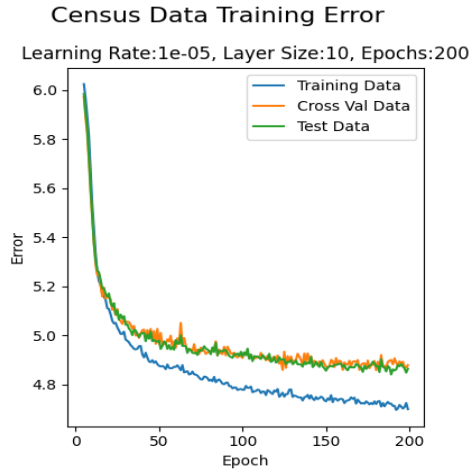

Figure 6: Census Data Neural Network Training Error: It shows that the cross validation error was very close to the real test data error and a good guide for decision making.

Figure 6 shows one of the best performing settings from the initial review. It shows that the cross validation error was very close to the real test data error and a good guide for decision making. The chart also doesn't clearly have a movement up of the cross validation error, indicating more epoch training could be beneficial.

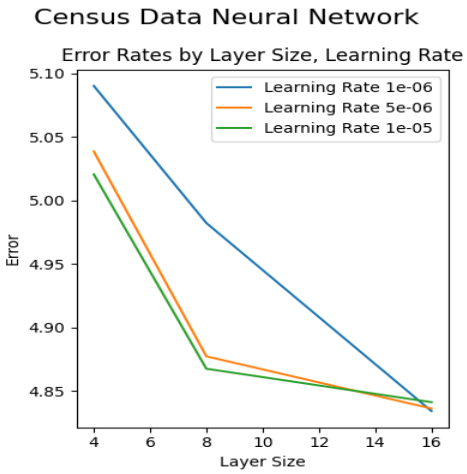

Figure 7: Census Data Neural Network: Lower settings for layer size and learning rate were used in testing. However, the absolute gains were very small, indicating the maximum performance for this dataset.

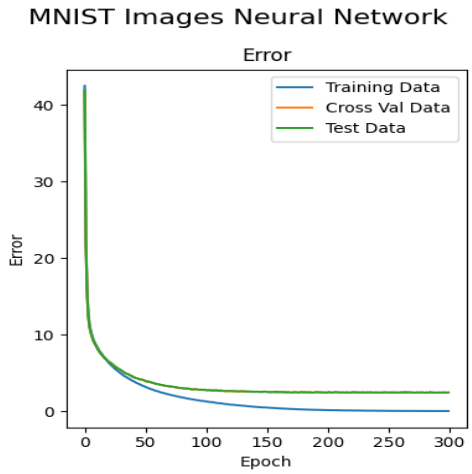

Figure 8: MNIST Neural Network: This uses the flattened images and no convolutional networks, with a cross validation using a 4 fold split. The cross validation error closely tracking the test set error. The best accuracy reached was 97.27%, with 275 epoch of training.

I then tried a new round of testing with lower learning rates, smaller network sizes and longer epoch training. Those results are shown in **Figure 7**. The absolute gains over in performance were very small, indicating that this was the practical limit of performance. The best result was with a learning rate of 0.000001, hidden layer size of 16, with the best result coming around 600 epochs of training. The neural network performed worse than the decision tree.

For the MNIST data, I created a different neural network topology. The PyTorch site has many helpful resources and tutorials [3] providing good guidance for one hot encoding. The MNIST dataset is a very popular one and there are many neural network implementations, but I stuck to the general PyTorch tutorial for a core understanding. I was able to use this as a skeleton to explore the algorithm and the data. I kept the image data flattened, creating 784 input features. Initially, I had two hidden layers, the first with 120 nodes, the second with 84 nodes. For the output, I used one hot encoding on the labels, with one bit for each value between 0 and 9, creating 10 outputs. To convert the 10 outputs to a single integer, the bit with the highest value is chosen. I did not include any convolutional layers to allow for a more accurate comparison between the decision tree and the neural network.

For this network, I used a different loss function, Cross Entropy Loss, to account for the single selection amongst one of ten output bits. I also used the SGD optimizer with momentum.

Figure 8 shows the result of the initial review with the flattened images. Cross validation closely tracks the test accuracy, indicating it is a good tool for decision making. The training improvement plateaued around 275 epochs of training. The best accuracy reached here was 97.27%, which is 9% better than the decision tree. I think this is a result of several things. From the dataset's standpoint, it is a completely continuous data set, with no discrete values, which is better for the linear layers of the neural network. It is also a less noisy dataset, with a reasonable outcome expected for each instance. In contrast, the Census data does not have a necessary relationship, with some instances conflicting with other ones on output labels. I also think the cross entropy loss function is also much better fit for this classification problem.

A key distinction in the MNIST dataset is that each feature has a spatial relationship to the others, in the form of a 2 dimensional grid. It is a common practice to use convolutional networks to take advantage of that relationship. I added two convolutional networks to the beginning of the network. The first added 10 channels and used a kernel size of 5. This means that for each 5 X 5 grid, a 5 X 5 filter is applied that multiplies each grid by the filter number, then sums the results. This is moved over all parts of the image and allows for spatial features to be highlighted. There is a different filter for each channel, with the different filters enhancing aspects of the grid, like edge detection. Without padding the image, this will result in an output that has 4 less on each dimension, reducing the 28 X 28 grid to 24 X 24. There is then a pooling layer that takes each 2 X 2 grid and choses the max value. This reduces each dimension by 2, changing the 24 X 24 grid to 12 X 12. Another convolutional layer was added with 20 channels, leading to an output of 20 X 4 X 4. This was then connected to the previous network.

The results were much better using the convolutional networks. **Figure 9** shows the great increase in performance, with it reaching the the same accuracy of the non CNN network in 13 epochs vs 275 epochs. However, the training time was much greater, with the CNN layers requiring significant processing time. The maximum accuracy also improved by 1.5% to 98.89%. This improvement in accuracy is significant as the the last few percentage points in accuracy can make the difference in many real world settings between

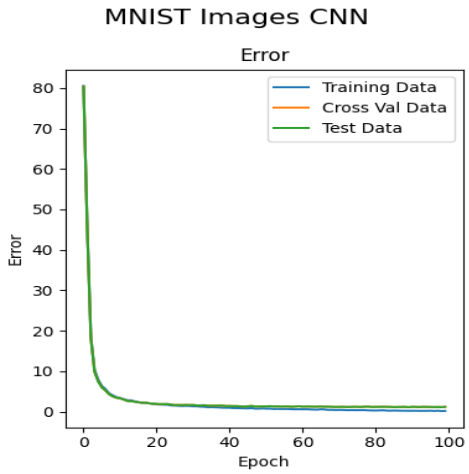

Figure 9: MNIST Convolutional Neural Network: This network used two convolutional layers, creating much better performance. It reached the the same accuracy as the non CNN network in 13 epochs, which took 275 epochs. The overfitting is just starting to be seen around 100 epochs.

success and failure.

Overall, the CNN layers represent use of domain knowledge about image data and demonstrate the flexible power of neural networks.

Boosting

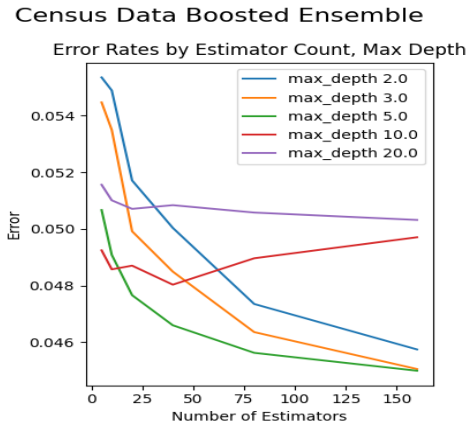

Figure 10: Boosting Census Data Max Depth: This shows error rates with different number of estimators and different Max Depths of the estimators. Max features was set to 3.

I chose to use the sklearn GradientBoostingClassifier for boosting. I set the classifiers loss setting to "exponential" so that it implements the ADABoost algorithm. Without this setting, the classifier uses a more complex loss function and it does a gradient descent. I chose the ADABoost because it was closer to the algorithm discussed in class.

Two of the key considerations when working with ensemble boosting is how many learners to use and how "weak"

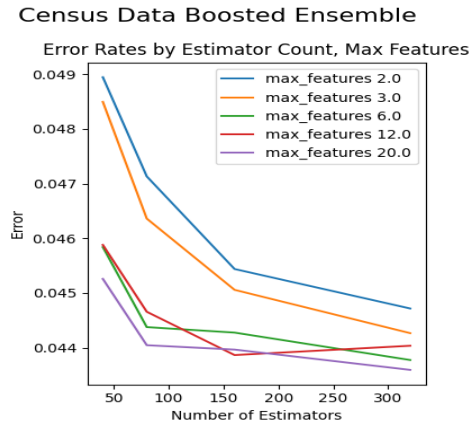

Figure 11: Boosting Census Data Max Features: This shows error rates with different number of estimators vs the maximum number of features for each tree. Maximum depth was set to 3

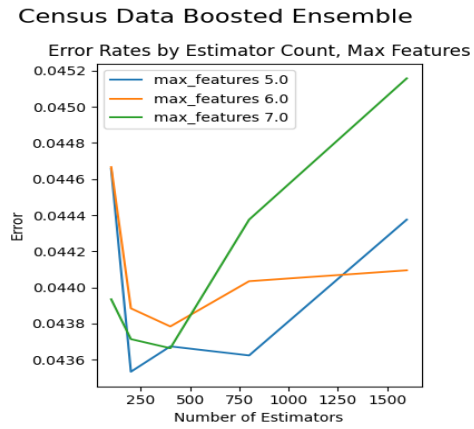

Figure 12: Boosting Census Data Max Features: This shows over fitting after 800 estimator trees. Maximum depth was set to 3

the learners are. The Max Depth setting determines what the maximum depth of each of the estimator trees is. By making this small, the learners become less accurate, but also less prone to overfitting.

For the Census data, **Figure 10** shows how increasing depth up to 5 improves performance, but at depths over 10 performance decreases and becomes erratic. The change at higher levels is somewhat counter intuitive, as the individual learners are more accurate, but is likely due to the fact that the more accurate learners are overfitting. **Figure 10** also shows how adding estimators improves performance, although with diminishing returns.

I then focused on reviewing the number of features setting. The classifier will consider a maximum number of features chosen at random for each tree. Generally, lower numbers make the learners less accurate, high numbers may be better initially but allow for some overfitting. **Figure 11**

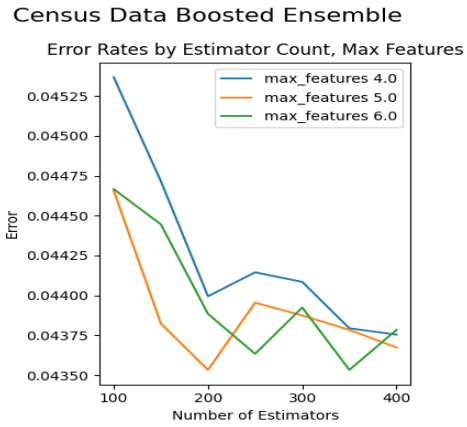

Figure 13: Boosting Census Data Final Review: Max Features of 5 at 200 estimators is found to be the best solution, achieving 95.65%

shows that while considering up to 20 features is the best, 6 features is very close.

Figure 12 shows overfitting after 400 - 800 estimators, leading to a final review with lower feature counts focused on 80-320 estimators.

Figure 13 shows maximum features of 5 with 200 estimators is found to be the most accurate solution, achieving 95.65%. This is better than the 95.30% achieved with the decision tree, showing the power of using relatively weak learners in a thoughtful ensemble.

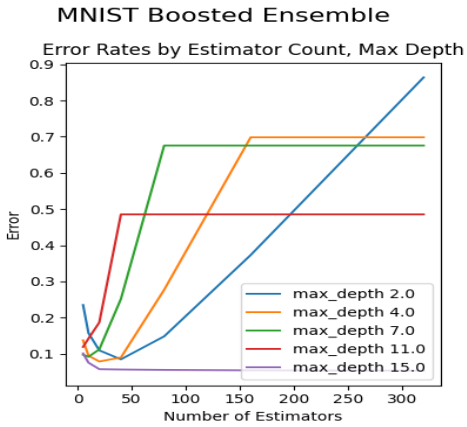

Figure 14: MNIST Data Max Depth: The chart shows overfitting occuring with relatively few estimators. Maximum features was set to 28.

For the MNIST data, I reviewed the estimator count vs max depth to begin with. I set the maximum features to 28, the square root of the features, which is a commonly used starting point.

Figure 14 shows overfitting with relatively few estimators. An interesting point is that the over fitting then reaches a plateau, which may be from the interplay between maximum features and maximum depth. It is also a bit hard to make sense of the max depth, as the best performing are the low of 2,4, and high of 15.

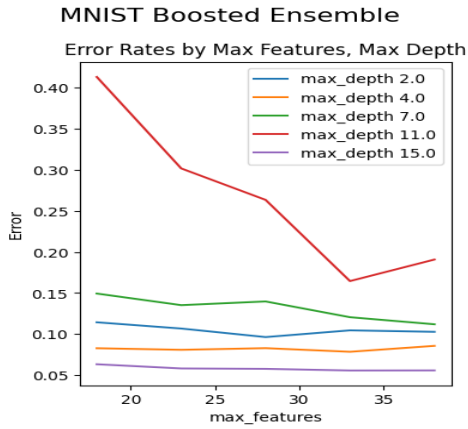

Figure 15: MNIST Data Max Feature vs Max Depth: This shows that the best performing max depth is 15, then 4, then 2. It is odd that the high and low maximum depths perform better. The number of estimators was 25.

More review was then performed testing max features vs max depth, using 25 estimators. **Figure 15** shows the result, with the best results being a max depth of 15, with more features generally performing better.

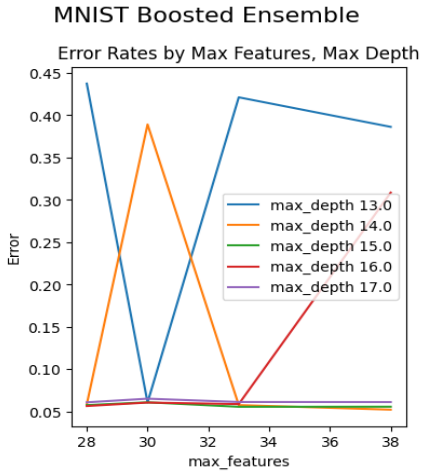

Figure 16: MNIST Data Max Feature vs Max Depth: The number of estimators was 25.

The final review, seen in **Figure 16**, shows the best performance with a max depth of 14 and max features of 38. This achieved accuracy of 94.44%, which is significantly better than the decision tree, but not as good as the neural network. This makes sense, as the boosted ensemble is better able to deal with the large number of features, but still doesn't take optimum advantage of the spatial relationships in image files.

Support Vector Machines

I chose to use SVC from the sklearn library. The key area for review was the use of different kernels. For both datasets, the following kernels were tested: Linear, Polynomial with 3rd degree, Radial Bias Function, and Sigmoid.

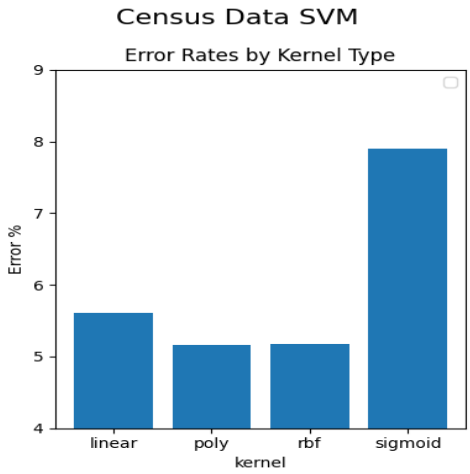

Figure 17: Census Data SVM Classifier: Error rate by different kernel types.

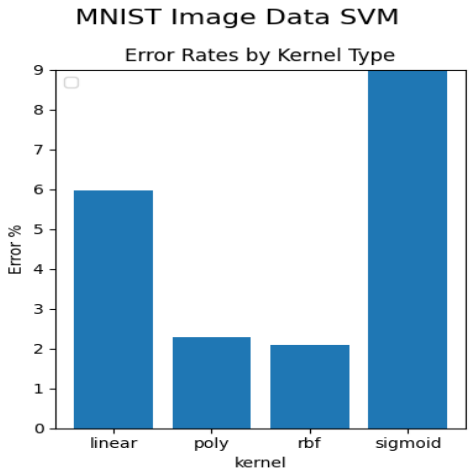

Figure 18: MNIST Images SVM Classfier: Error rate by different kernel types.

The Census Data review is seen in **Figure 17**. This shows that the Polynomial (3rd degree) and Radial Bias kernels performed the best. The slight winner was Polynomial, with a test accuracy of 94.83%. The SVC solver took a significantly longer time to solve than the previous algorithms. The dataset size, with almost 200,000 training instances is a significant challenge to this solution type.

The MNIST Data review is seen in **Figure 18**. On this dataset, the Radial Bias kernel performed the best, with a test accuracy of 97.92%. This was a very good score, but not as good at CNN Neural Network.

K-Nearest Neighbors

I used the sklearn KNeighborsClassifier for the K-Nearest Neighbors classifier, with key settings for review were the appropriate number of neighbors, the distance method and potentially weighting based on distance. One thing to note is that K-Nearest Neighbors is a lazy learner, which in this application means that the fit function, which is used in the classifier models, is very fast, mostly just loading the training data. It is on the predict function that the learner takes its time.

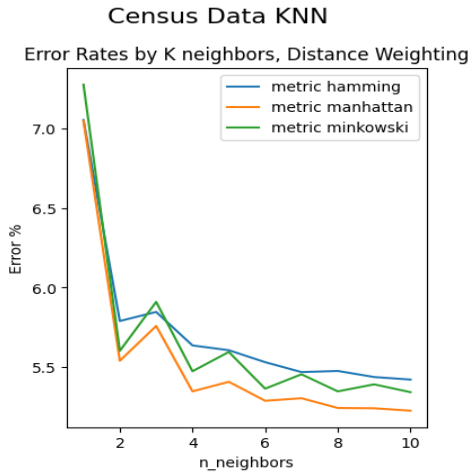

Figure 19: Census KNN K : Error rate by different settings of K and different distance formulas. Neighbors were not weighted by distance.

Figure 19 shows KNN classifier with the Census data. A key tuning function with KNN's is the choice of distance metrics used to find the nearest neighbors. Three were considered for the Census data. Manhattan Distance worked the best, with larger values of K for more neighbors working better. With a mix of continuous (Age) features and discrete (Sex, Working Status), distance or similarity is tricky to calculate. With discrete values encoded as numbers, a false sense of dissimilarity can occur. For instance, a category with 4 values - Employed, Un-employed, Student, Retired might be encoded to 0,1,2,3. When calculating feature distance between an instance with Employed (0) to Unemployed (1), the distance would be 1, but to Retired(3) would be 3. However, this is not a good measure of dissimilarity in this case. Hamming distance, which calculates distance by determining if features match or don't match, would possibly be a better measure, but actually performs the worst, possibly being thrown off by the continuous features. A key issue is that the 40 features do not have weightings, so each feature is valued the same in the KNN solution. This is strongly contrasted with a decision tree, where only the features that provide the most information gain are used.

Another key tuning aspect of KNN's are whether to use distance to weight the neighbors. Another round of review was done on the census data, checking the K values in a higher range, and considering weighting neighbors by their

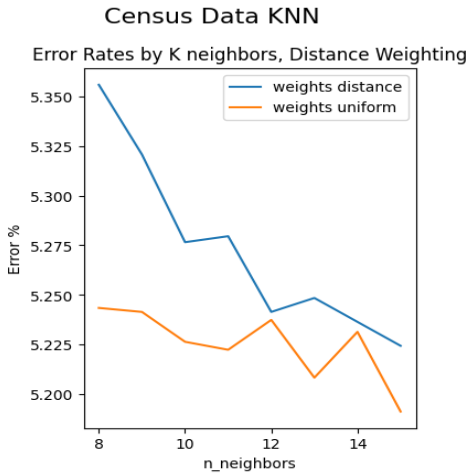

Figure 20: Census KNN Weighting and Distance review: Error rate with different settings of K and weighting by distance or uniform weighting using Manhattan distance.

distance and is seen in **Figure 20**. For the Census data, uniform weight performed better, with the highest accuracy reached at 94.81%.

One way to remove the issue of arbitrarily encoding discrete values is to use one hot encoding, which creates a new binary feature for every value in all discrete features. For the census data set, this increased the number of features by a factor of 20. With the KNN algorithm, this significantly increased the prediction time with no gain in accuracy. While it eliminated the false distances of encoded features, the curse of dimensionality meant that all the added features created a much bigger dimensional space, which limited accuracy improvements.

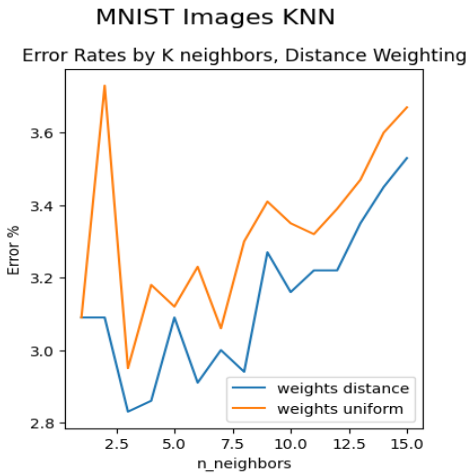

Figure 21: MNIST Images KNN: Error rate by different settings of K and different weightings of neighbors by Minkowski distance. Uniform means all neighbors are weighted the same, distance means neighbors are weighted by the inverse of distance.

Figure 21 shows the MNIST KNN classifier. Using a Minkowski distance metric, the best value of K is 3, with neighbors weighted by inverse of distance. Overall, this performs very well, with an overall accuracy of 97.17%, the second best of all algorithms. The strong performance for the MNIST images is driven by several things. The algorithm effectively uses all the feature data, because all are calculated in the distance metric. With all features being continuous and originally on the same scale, the distance metric is an effective way to compare all features. The intuition is that similar numbers will be closer for each pixel feature than dissimilar ones. For example, two zeros may not line up perfectly, but they will have closer pixels that a one and a zero. Weighting neighbors by their distance enhances this, making less likely neighbors less impactful. The small K also makes sense because numbers may be shifted in the image. For example, two images of a 1 with one shifted to the left would have a big distance, as almost no pixels are a match. In this case, a large K would allow other images with a big distance to be selected, like a 0 which does cross over a 1 in several places. A small K restricts this selection to more likely true matches.

This algorithm's ability to generalize is very dependent on the training set in a clear way, in that the classification of a test instance directly uses similar training instances. The number of neighbors, K, the distance metric and weighting can be adjusted to produce a form of generalization.

Summary

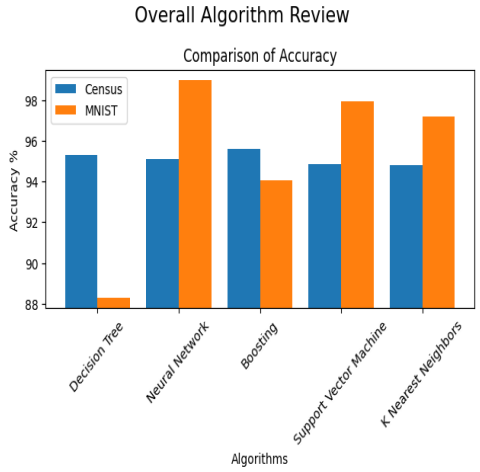

Figure 22: Overall Accuracy Comparison: This compares the accuracy of each algorithm on the test data for each data set using the training data and best tuning parameters for that algorithm and dataset.

Figure 22 reviews the algorithms overall accuracy on each data set. This uses the best tuning parameters after learning on the training data with cross validation, then testing using the test data for each set.

For the Census data, the boosting algorithm performed best. The Census data was a skewed dataset, with positive

responses being 6% of the data and all the algorithms were able to reach 93-94% accuracy by returning negative for all items. Getting above this basic threshold was challenging due to the noisiness and possibly the mixed discrete and continuous features. Boosting, specifically ADABoost, as mentioned in the reading and lectures, is able to handle noise and still generalize fairly well. KNN performed the worst, which makes sense with a noisy dataset and challenging distance metric for arbitrary discrete values.

For all the algorithms, I don't think I fully optimized the data presentation of the discrete features in the Census dataset. I tried one hot encoding of the discrete features for the KNN algorithm without improving accuracy and greatly increasing prediction time, but did not try it on the other algorithms. The interplay between discrete feature representation and the algorithms is critical and can strongly influence the outcome.

For the MNIST data, the neural network with two convolutional layers had the highest accuracy. The convolutional layers are especially constructed to take advantage of the spatial relationship in the MNIST image data, consuming the 2d image arrays without flattening. I doubt the upper limit of generalized accuracy was met, with the significant other network topologies that could be tested. The basic decision tree performed the worst by a large margin. It was a poor fit due to the relatively large number of features (784) which were strongly related spatially. In comparison, boosting performed much better due to its ability to generalize better over the feature set.

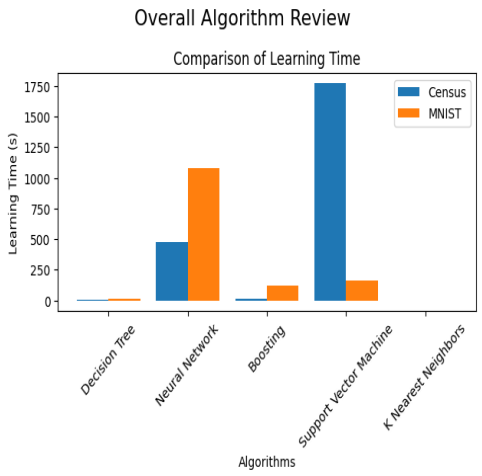

Figure 23: Learning Time Comparison: This compares the time to learn for each algorithm on the training data using the best tuning parameters for that algorithm and data set.

Accuracy is not the only critical metric, with many real world applications needing to trade off accuracy with time to learn and time to predict. **Figure 23** shows the comparisons of time to learn on both datasets for all algorithms. KNN took the least time to learn in that it is a lazy learner. The "learning" or fitting step simply loads the data with some space optimizations and indexing, with no real learning occurring. The quickest eager learner was the decision tree, which is also the simplest. The neural network and the support vector machines took the longest time to learn, which is not surprising in that they have the most weights and tunings that are being adjusted during learning.

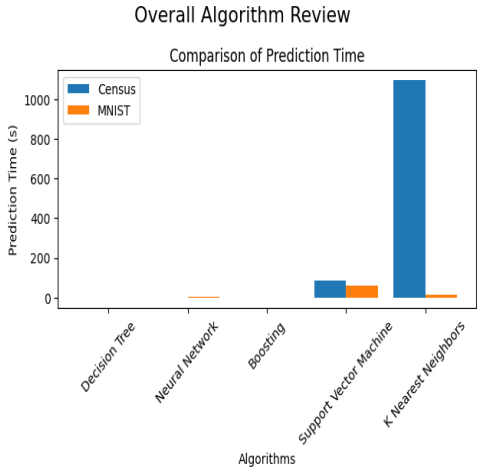

Figure 24: Prediction Time Comparison: This compares the time to predict results on the testing data for each dataset using each algorithm with the best tuning parameters.

Figure 24 shows the time to predict using the algorithms after they are fitted. There is a complete reversal for the KNN algorithm for the Census data, with it taking by far the longest time to predict. As a lazy learner, it is effectively learning by finding each test instances' nearest neighbors. It effectively has to "learn" every test instance, with no learning benefit accrued to other instances. However, it performed faster than SVM on the MNIST data, most likely due to the fact that the distance function was well behaved with continuous and comparable features for KNN and the dimensional calculations for the SVM with 784 features were complex.

Boosting and the Decision tree are the fastest predictors, with their simple models performing very quickly. Neural networks, which have orders of magnitude more weights than trees or boosted ensembles, still perform very quickly, with their prediction step being straight forward. Neural networks also lend themselves to specialized hardware using massive parallel computations. Interestingly, support vector machines take the longest time for predictions of the eager learner algorithms and also were among the longest in the learning time.

This was a very satisfying exercise due to the ability to try the same data sets on multiple algorithms and try each algorithm on multiple data sets. I found the Census dataset a bit frustrating and would be interested to try a less skewed dataset in the future, potentially with only discrete features. Throughout the exercise, feature review for the Census data may have been very useful, which I believe we will review some of that later in the class. I had no exposure to SVM's before this class and would consider them in the future due to their strong performance.

References

1 Census Income (KDD). url: https://archive-beta.ics.uci.edu/ml/datasets/census+income+kdd.

2 The MNIST Database of Handwritten Digits. url: http://yann.lecun.com/exdb/mnist/.

3 Pytorch Tutorial on Training a Classifier. url: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

4 Analysis Code used in this paper. url: https://github.com/BigTMiami/ML_GT_CS7641.