OVERVIEW

In order to better explore situations where some supervised learning algorithms may be better than others, I chose two distinct classification problems with significant fundamental differences. One problem is hard for humans (dataset #1), and one is extremely easy for humans (dataset #2). I used the popular sklearn and matplotlib python libraries in order to input, clean, store, and process the datasets through supervised learners, and to generate result data. Some trivial tasks were completed using Microsoft Excel. Excluding labels, all the data was normalized using standard functions in sklearn to [0,1] to reduce numerical bias in the different types of features in the data, which improved both the processing speed and performance of the algorithms. The data rows were shuffled before processing to avoid bias in the ordering of the dataset, and extensive cross validation (with an 80/20 train-test split) was used throughout the analysis during hyper-parameter tuning and algorithm evaluations.

DATASET #1

My first classification task, using the Bank Marketing Data Set containing customer marketing data for a Portuguese bank, was to predict whether a person (single instance) will sign up with the bank in the future. What I found interesting about this classification problem is that a human untrained in this domain would likely struggle to perform this task manually. There are features in this dataset that are likely noise, e.g., one feature is the last day of the week (which week, is unknown) the person was contacted. These were purposely left in the dataset to observe which learner is best at sifting through the noise. A single column 'duration' regarding the duration of the most recent call with the customer was removed for preprocessing per notes provided with the dataset, which explain the feature makes negative classifications (customer does not subscribe to the bank's services) trivial, with 0 duration always resulting in a true negative. This removal preserved the difficulty of the classification task. Additional preprocessing included transforming categorical features into numerical labels and converting the data to be comma-separated. This dataset contains a total of 41186 instances.

DATASET # 2

My second classification problem, using the Letter Recognition Data Set, was to predict, given image data as input, which capital letter of the alphabet a base image represents. The features were all provided numerically, and each summarize some aspect of the base image. In other words, each record of the dataset corresponds to identifying information about a single image of a capital English letter with various fonts. A human can solve this problem trivially through their own cognitive natural language processing, however, the learners must perform their classification empirically, which is why it's an interesting classification problem. The total number of instances in this dataset is 20,000.

KEY DIFFERENCES BETWEEN THE DATASETS

CLASS DISTRIBUTION

The bank dataset has a very clear imbalance with the distribution of labels in the population, as shown in the figure to the right. This presents a challenge for the classifiers, because traditional metrics such as accuracy/error will be skewed by

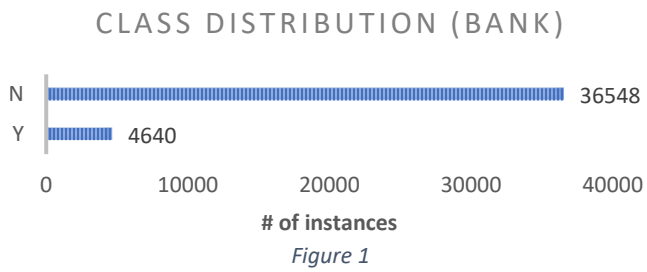


Figure 1

the prevalence of negatives in the set, even though there are only two possible outputs. In order to combat this, overfitting had to be carefully and critically examined. In contrast, the letter dataset has a much more balanced population, as shown in the chart in *Figure 2*. There is still some imbalance, but the distribution has adequate representation for each letter. It is important to note that that the bank dataset contained twice as many instances as the letter dataset, and the learning curves shown later in the report provide evidence the bank set likely would *not* benefit from more training examples, but the letter dataset *would* benefit from more training examples, given the same distributions.

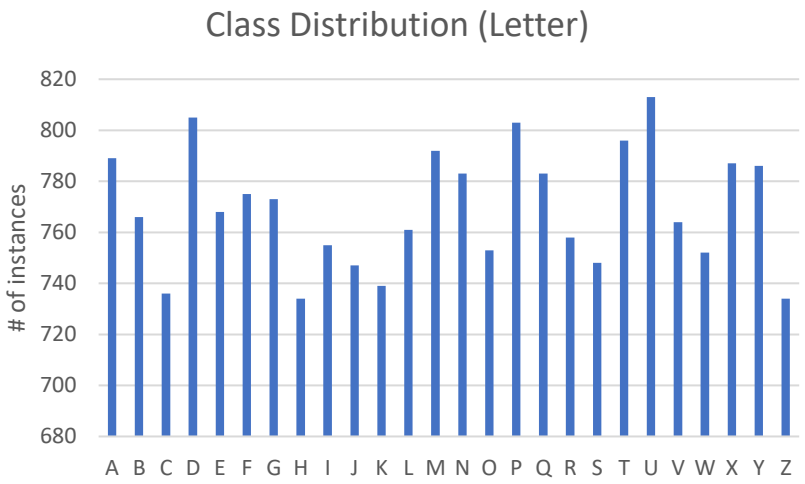


Figure 2

"Noise"

At first glance, it seems obvious that the bank dataset would contain more noise than the letter database, and this is supported empirically throughout the rest of the analysis. Intuitively, this is likely because the letter database provides clear information about the image data it represents, and there are less "missing variables" that may exist in the true feature space (outside of what exists in the datasets). The image features are very detailed and distinct compared to the bank features, and less likely to be 'noise'. Additionally, each letter instance only varies slightly from font to font, whereas each bank customer may be profoundly different from one another, even though they have the same binary label.

Types of features in the data

The bank dataset contains a large portion of categorical features (e.g., level of education and occupation), as well as numerical features (e.g., age). These features are also distinct between each other; a person's age is on a far different scale than their employment variation rate, for example. This could create issues with distance-based learners, despite using normalization to try and mitigate this potential problem. The letter dataset, in contrast, contains strictly numeric feature data, and each feature is similar in nature with each other (e.g., y-box vertical position, height of box, etc.). Both datasets have 15+ features and therefore, equally high dimensionality.

Chance of random correct prediction

The chance of choosing a correct label for a given input varies greatly between the two datasets. In the bank dataset, there is a 50% chance of correctly choosing the label, whereas in the letter dataset, there is a ~4% (1/26) chance of randomly choosing a choosing the correct capital letter given an input instance. Therefore, the accuracy metric is much more useful in the case of the letter dataset. In the bank dataset, returning 100% negatives will result in high accuracy if the test data contains the same distribution as the population, so it is more difficult to tune and evaluate the bank classification results.

Decision Trees Classifier

The process used to test each algorithm follow a similar structure: run a benchmark test with an untuned classifier, tune hyper-parameters using n-fold cross validation curves (with an 80/20 train-test split, n depends on train time), test the tuned classifier using a learning curve with the same cross-validation split to compare train and test performance, and finally, run a final test on a single split of data. For decision trees, the DecisionTreeClassifier from sklearn was used, and there were two key hyper-parameters that were tuned: **max depth**, and **max leaf nodes**. These parameters represent pruning the tree either in the form of max tree depth, or a maximum number of leaf nodes, respectively.

Bank Dataset

When the bank classification was run using a default DecisionTreeClassifier, training accuracy was approximately 99-100%, however, the test accuracy was hovering between 80-83%. This can be reasonably attributed to overfitting. By default, the classifier had no limit on max depth or max leaf nodes, i.e., no pruning is used for the default classifier. Thus, a tree could grow very granular, causing it to overfit the data. In the validation curve over values of max depth, I observed that cross-validation accuracy improves up to a max depth of 5, and the accuracy begins to decrease after 5.

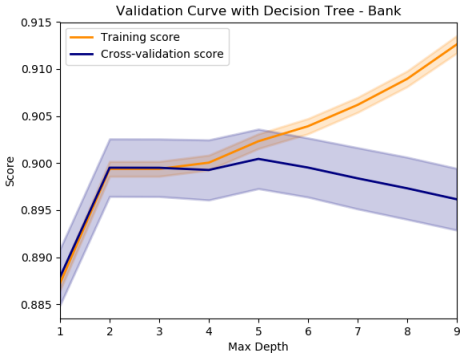


Figure 3

When a validation curve was generated over max leaf nodes, a similar behavior was found at around 20 max leaf nodes. However, each form of pruning effectively reduced overfitting on their own and using both methods in tandem resulted in lower accuracy than a single method, so the I decided to keep **max depth** to **2**, and **max leaf nodes** to **unlimited/None**.

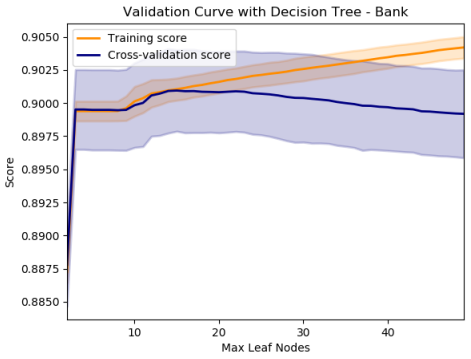


Figure 4

This simple pruning method strongly improved the accuracy scores of the classifier on the bank dataset over 100 iterations of cross-validation. As shown on the learning curve to the right, we find that training and cross-validation score (i.e., prediction accuracy) converges at around 90%, with no bias or variance. It's important to note that the learner does not improve as training data increases past roughly 15,000 training examples, because both scores have converged by then to the same rate. Most of the classifiers seem to converge at this level for the bank dataset, so additional metrics will be examined in later sections.

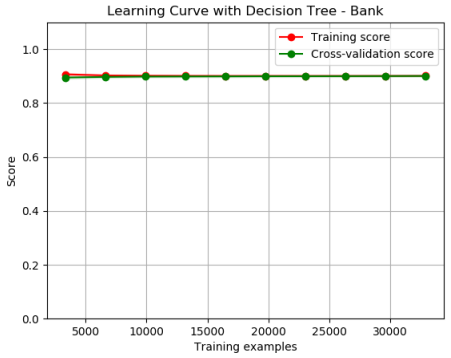


Figure 5

LETTER DATASET

Once again, the analysis for the DecisionTreeClassifier began with a benchmark test using a default decision tree with no pruning, however, something interesting occurred during the benchmark - the decision tree performed quite well. For verification, I performed more validation testing over the pruning hyper-parameters (max depth and max leaf nodes).

As evidenced by the curves shown in *Figure 6/7*, a max depth of around 25 resulted in a plateau of performance. Upon further inspection, the default decision tree converged to around 25 depth when left unpruned, so pruning via max leaf nodes resulted in strictly worse results under that threshold.

A similar pattern was observed when generating a validation curve over max leaf nodes. Pruning max leaf nodes resulted in strictly worse accuracy scores, up to a plateau eventually where the tree was not being pruned.

The conclusion drawn from these results was that **pruning does not work on the letter dataset**, because the dataset is not susceptible to overfitting like the bank dataset. The closer the classifier was able to fit to the training data, the higher the validation scores the classifier was able to achieve. The learning curve shown to the right shows the performance of the default classifier and suggests that more training data would benefit the classifier. This also suggest that other learners that can better fit the data would likely be better algorithms for this dataset.

After tuning the classifier, a stable error rate of approximately 12% was converged to. The results of the errors for each tuned classifier will be summarized at the end of the report as well.

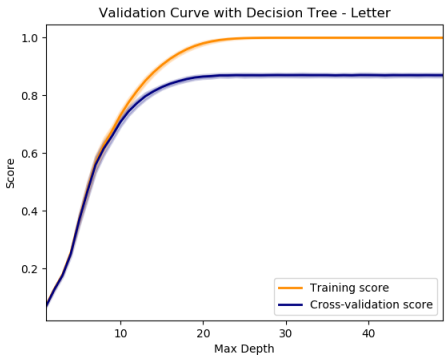


Figure 6

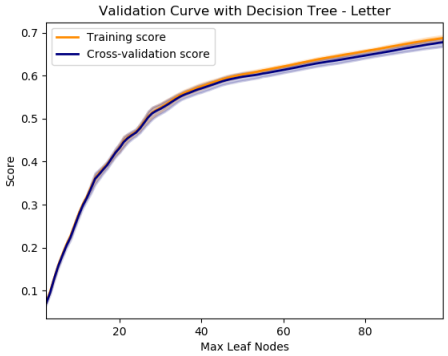


Figure 7

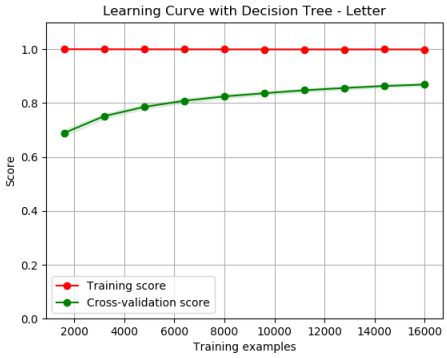


Figure 8

K-NEAREST NEIGHBORS CLASSIFIER

Following the same pattern set forth in the Decision Trees analysis, I used sklearn's default KNeighborsClassifier as a benchmark, and tuned the number of neighbors used (**'k' in k-neighbors**), as well the **distance metric** used by the classifier to improve the results from the benchmark. By default, the classifier is 5-neighbor, and uses the Euclidean distance metric (technically, the classifier uses the 'minkowski' distance metric with a power parameter of 2, which is equivalent to Euclidean distance).

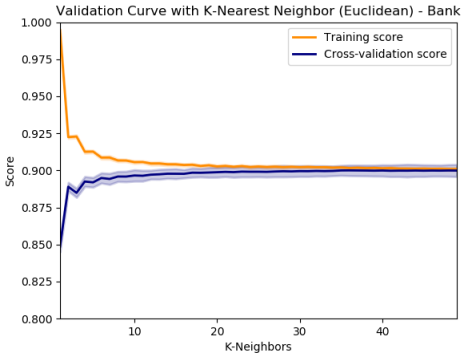


Figure 9

BANK DATASET

The benchmark results for the default KNeighborsClassifier were close to the performance of the tuned DecisionTreeClassifier. To closer examine this, a validation curve was generated over the value of 'k', as shown in *Figure 9*.

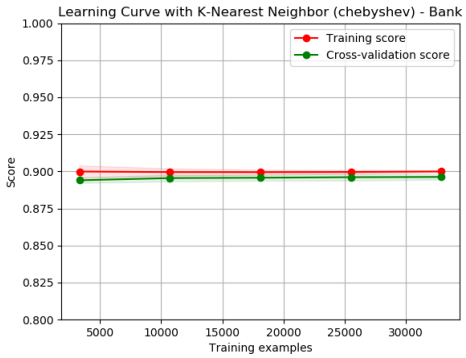


Figure 10

The results show overfitting occurring between the range of k = [1,20], and a convergence between the train and test scores at roughly **k = 30**. Thus, a slight increase in performance may be reached by setting k to 30. The first curve was generated using Euclidian distance, however, the same pattern was observed at k = 30 when the curve was generated for the Chebyshev distance metric (plot not shown).

To better compare the two distance metrics (since the validation curves were nearly identical), I generated two learning curves to examine which metric better matched the training performance. Although the difference was very slight, the Chebyshev metric resulted in a small amount of persisting variance (shown in *Figure 10*), whereas the **Euclidean** metric curve did not have this variance, but instead closely matched the training score.

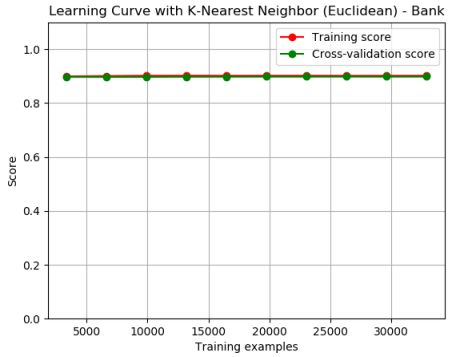


Figure 11

In conclusion, the KNeighborsClassifier was not able to significantly increase its accuracy score compared to the DecisionTreeClassifier for the bank dataset, and quickly hit an upper limit on performance, as shown in *Figure 11*.

LETTER DATASET

When the same validation testing was performed on the letter dataset as the bank dataset, different results were observed than what was found in the bank dataset results.

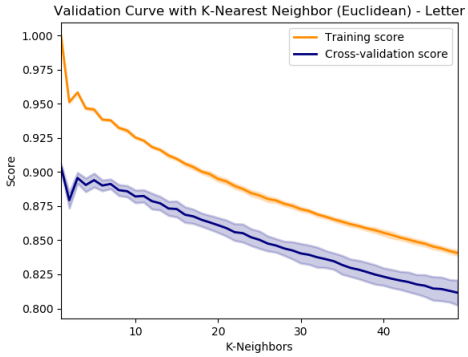


Figure 12

Instead of the default value of 5-neighbors being too low, it was, in fact, too *high* for the dataset. At k = 2, a sharp decrease in accuracy was observed, followed by a sharp increase at k = 3 (still worse than k = 1), and a steady decline as k increased past 3. The fact that the nearest neighbor (**k of 1**) provides the best generality for guessing new letters is remarkable; this suggests that the training data is comprehensively representative of new data instances.

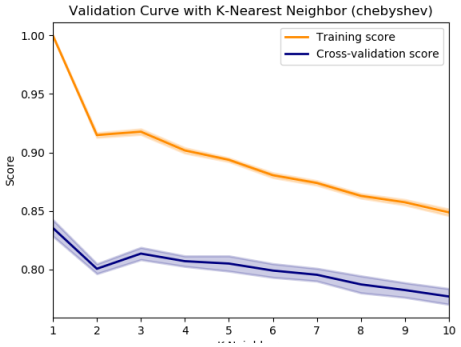


Figure 13

Once again, when this same plot was generated using the Chebyshev distance, the shape of the graph was unchanged, however, the accuracy score was translated downwards with strictly worse accuracy scores (as shown in the plot to the right). To verify the difference in accuracy scores between the two metrics, two learning curves were plotted with k = 1, but with

the two different metrics. The results were nearly identical; the training error remained zero for both plots, however, except the Chebyshev cross-validation score plot was translated downwards around 7%, which helped me decide to keep Euclidean distance.

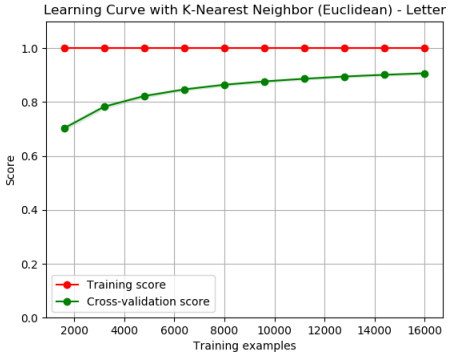


Figure 14

The pattern of fitting the letter training data as closely as possible remained the best way to reduce bias and variance for classification of the letters. The learning curve in *Figure 14* provides evidence that the classifier would likely benefit from having more training data, even more-so than for the DecisionTreeClassifier. The final classifier had **k set to 1** and used the **Euclidean** metric.

Neural Network Classifier

A neural network classifier was one of the most challenging algorithms to tune, primarily because of training time required. Even over a small number of folds used for cross-validation testing, training time took several hours using the sklearn MLPClassifier. The parameters I tuned included the **number of hidden layers**, the **number of neurons** at each hidden layer, the **activation function**, and **the number of iterations**.

Bank Dataset

The first parameter tuned after running a benchmark test (which provided an accuracy score under 85%) was the hidden layer size. Under a validation curve, it was observed that accuracy rates for a 1 layer network plateaued at 15+ layers. Upon adding additional layers, I found the results to be extremely erratic, and I also found the results to never go past 90% accuracy score regardless of the number of layers and neurons, at the additional cost of computational resources. Because the bank dataset is "noisy", the neural network tended to overfit the data, which limited the performance of adding additional layers and hidden layer neurons. I found a stable performance at a network with **2 layers**, with **15 neurons in the first layer, and 2 neurons in the final layer** for the final "yes" or "no" classification.

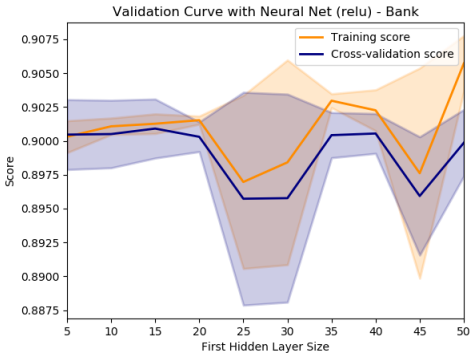


Figure 15 – Hidden Layer Validation Testing

After comprehensive testing with max iterations, I found that limiting the iterations resulted in either no change to the algorithm's performance, or a negative impact on the score, so the graphs were not included here. I found that limiting the number of hidden layers was much better at reducing overfitting than limiting the iteration count. It was interesting to observe that the neural network converged relatively quickly for both datasets (within 3000-5000 iterations at the most), however, the SVC classifier had an extremely difficult time converging for the bank dataset, even with 200,000 max iterations allowed.

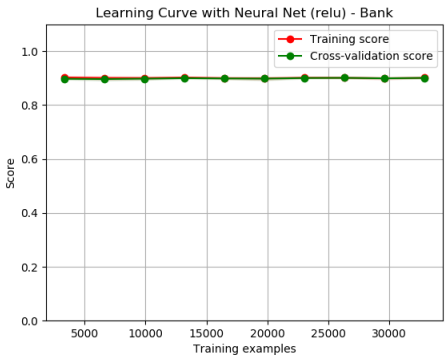


Figure 16

The above tests were performed with a network that used an **ReLU activation function**. I ran the same tests using a logistic activation function, however, the results were nearly identical. As a final test, I generated a learning curve for both activation functions to determine which function to use, and the results were again, identical. With all else equal, the ReLU function is more computationally efficient, so I opted to use it in the final tuned network.

In conclusion, the neural network was not able to provide significantly different results in terms of accuracy than a KNN or Decision Tree Classifier in terms of accuracy, and more training examples aren't likely to improve accuracy based on the learning curve in *Figure 16*.

LETTER DATASET

My strategy for creating an accurate neural network based classifier for the letter set was influenced by the results from the previous classifier tests. A neural network is uniquely equipped to fit training data more closely than the previous two learners.

As previously stated, the MLPClassifier has a single parameter, "**hidden_layer_sizes**", that controls the number of layer and the number of neurons (i.e., perceptitrons) at each layer. This made validation testing extremely difficult, compounded with the excessive training time required. I took a balanced approach of testing the best number of neurons at each layer individually and noting when increasing the number of layers resulted in a decrease of the accuracy plateau, regardless of the number of neurons. I found that 3 layers was the upper limit of the accuracy increase, with a final network of **80-50-26**, each number representing the number of neurons at the respective layer, from left to right.

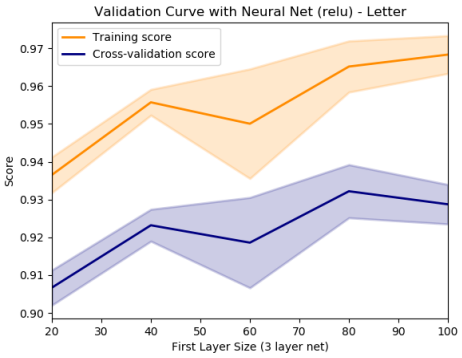


Figure 17 – Example Validation Test (Neural Net)

As found in the bank dataset, the letter dataset had strictly worse accuracy scores when iterations were limited below the required iterations for convergence with the default learning rate and stopping criteria. Intuitively, this makes sense, because the letter dataset is not very susceptible to overfitting and forcing early convergence would predictably *not* increase accuracy scores.

To compare activation functions, two learning curves were generated with layer-tuned neural networks, one with a logistic function and one with an ReLU function. The ReLU net performed marginally better in terms of accuracy at higher training examples, so I chose to use the ReLU as the tuned activation function.

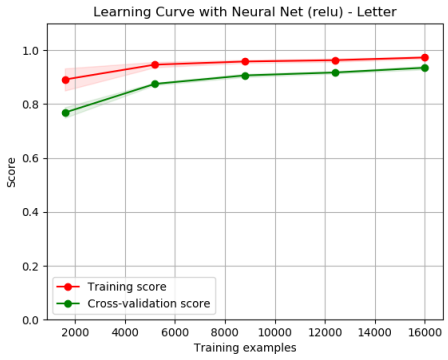


Figure 18

The final learning curve displays excellent results, only limited by the number of training samples available. By extrapolating the graph past the available training examples, it's very likely the classifier could converge to near perfect accuracy for both training and cross-validation scores. The learning curve in *Figure 18* is trending upwards for both the training score and cross-validation score.

Support Vector Machine Classifier (SVC)

The SVC classifier from sklearn was also extremely difficult to tune, again due to a large amount of time and computational resources required. There are many different parameters that could be tuned, however, the hyper-parameters I chose to focus on were the **penalty parameter (C)**, **maximum number of iterations,** and **kernel type**.

Bank Dataset

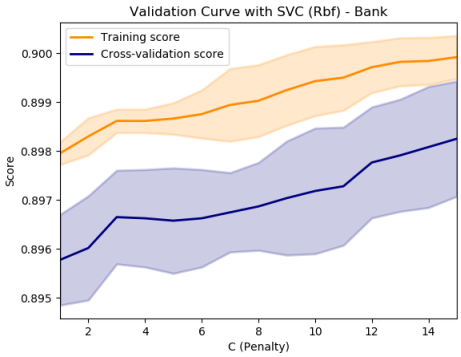


Figure 19

Tuning the SVC for the bank dataset was the most challenging part of the analysis. Most of the time, there was absolutely no improvements found upon changing parameters, and any improvement gained by tweaking the hyper-parameters were marginal at best.

The two **kernels** I tested were a linear kernel and an rbf (radial basis function) kernel. In order to test each kernel, validation curves were generated for maximum iterations, and the penalty parameter (C). The rbf kernel SVC had a slight increase in accuracy as the **C value was raised to 15**. After running the validation curve for values of C above 15, accuracy score stopped increasing. When the same tests were run with a linear kernel, the results were marginally worse, so the **rbf kernel** was used in the final tuned SVC.

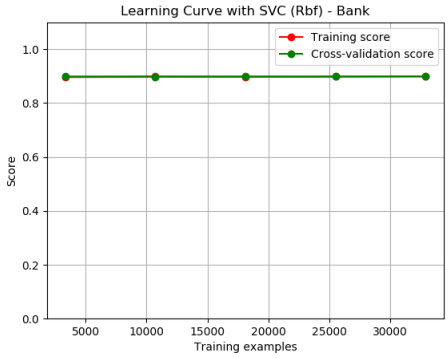


Figure 20

Like the observations regarding the **max iterations** for the neural network classifier, I found iterations to be a poor method of tuning, resulting in both inconsistent and inaccurate results when it was tuned below the iterations required for convergence. Altering the penalty parameter was more effective at tuning the performance of the algorithm. As the penalty parameter increases, the margin of the SVC decision boundary decreases, increasing the likelihood that a sample in the training set is classified correctly. To make the classifications more lenient, the value of C could be decreased, however, increasing C to 15 increased accuracy, and accuracy what the algorithms were being tuned for in this analysis.

Through the various testing, the rbf kernel had slightly less variance than the linear kernel, and the SVC had performance like the other classifiers, as shown in *Figure 20*. There are differences, however, between the learners with respect to precision, recall, and AUROC, which will be discussed after each learner has been discussed individually.

Letter Dataset

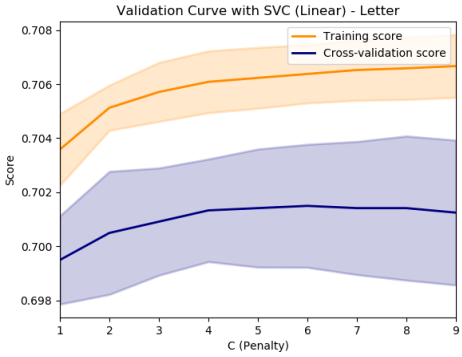


Figure 21

Directly contrasting the difficult tuning of the SVC in the bank dataset, the SVC performed extremely well on the letter dataset. Once again, the linear and rbf kernels were examined via learning curves, and penalty parameter / maximum iteration validation curves.

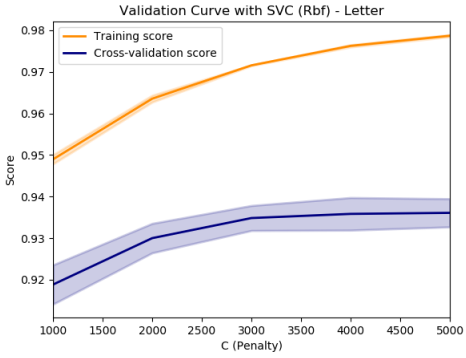


Figure 22

After running some initial testing, it was immediately apparent that the linear kernel performs very poorly on the letter dataset (see the example validation curve in *Figure 21*). In contrast, the **rbf kernel** performed phenomenally, with high accuracy scores for validation and training.

The penalty parameter (C) was able to be drastically increased without overfitting the dataset (a common theme with the letter set). After a lot of time spent expanding the range of C, I found the accuracy to converge at a penalty parameter of **5000**. As stated for the bank dataset, I did not find any value in limiting the **maximum iterations** allowed and have omitted the validation curves over iterations from the analysis for this reason.

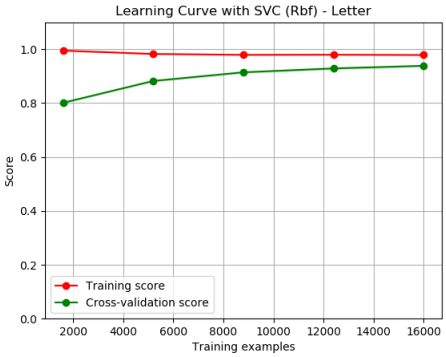


Figure 23

The SVC was able to maintain a very high accuracy score as training examples increased, for both training and cross-validation scores. If there were more training examples available, it may even be able to achieve near-perfect classification, even on samples not yet trained on. This is evidenced by the learning curve shown to the right; as training examples increase, the training score and cross-validation scores appear to be converging close to 1.0.

AdaBoost Ensemble with Decision Tree Weak Learners

Last but *certainly* not least up is the AdaBoost classifier, using Decision Trees as a base estimator. There were several parameters to tune for the AdaBoostClassifier from sklearn; namely, I tuned the **base estimator** DecisionTreeClassifier's **max tree depth**, **max leaf nodes**, and the **number of base estimators** used by the ensemble classifier (i.e., AdaBoostClassifier).

Bank Dataset

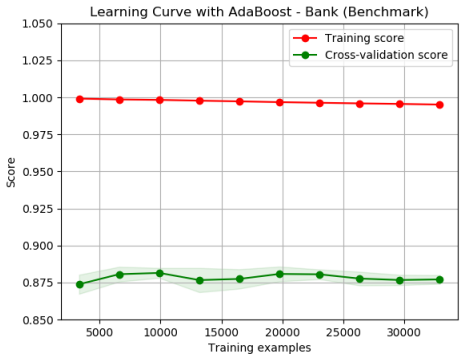


Figure 24

An initial benchmark run resulted in a high variance between the cross-validation score and the training score; which is likely attributed to overfitting the training data. The training accuracy was 1.0, with the test accuracy hovering around 0.8. For illustration purposes, a benchmark learning curve is provided in *Figure 24* to illustrate the benchmark methodology used for each classifier, including the classifiers discussed before the AdaBoost classifier.

Of course, the first idea to reduce this overfitting was to use pruning on the base estimator (i.e., weak learner) of the ensemble. The challenge was that the validation curve requires a parameter name, however, the variable that was required to be tuned was a parameter of a parameter (e.g., base_estimator.max_depth). To circumvent this issue, I created an array of base estimators with hyper-parameters to test (max depth and max leaf nodes) and iterated over the array when generating the validation curve, each iteration setting the base estimator to a DecisionTreeClassifier in the base estimator array.

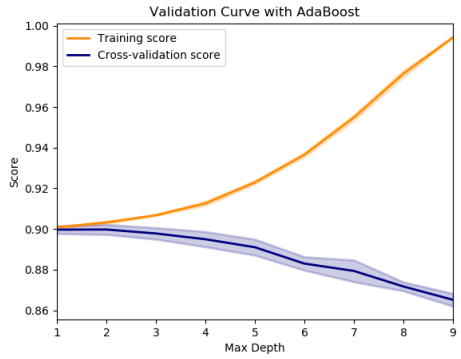


Figure 25

The results of the validation testing, as shown in *Figure 25* with max depth on the horizontal-axis, were very similar to the results found during the DecisionTreeClassifier tuning. The difference being, at max depth of 1 there is the least amount of overfitting. Even at max depth of 2, there is slight overfitting occurring. Similarly, any limit on the max number of leaf nodes results in overfitting, with a validation curve that nearly mirrors the max depth validation curve.

A possible explanation for this is that the classifier is performing best when a "clustering-like" behavior is occurring. It may be putting people into different categories, and based on that category, provided a yes or no based on similar people it's been trained on, rather than devise a multi-layer decision process.

When the number of estimators was tuned, a performance plateau was found at 50 estimators, with no significant changes in results occurring past 50.

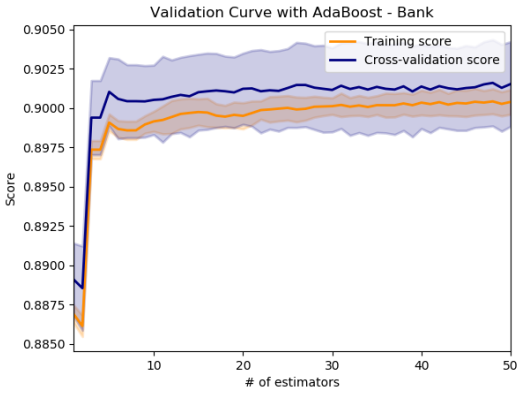


Figure 26

By generating a learning curve over a tuned AdaBoostClassifier with **50 estimators**, with each estimator a DecisionTreeClassifier with a **max depth of 1**, the variance was reduced between the training and cross-validation scores, and once again, the accuracy of the classifier remains fairly constant as training examples increase (*Figure 27)*.

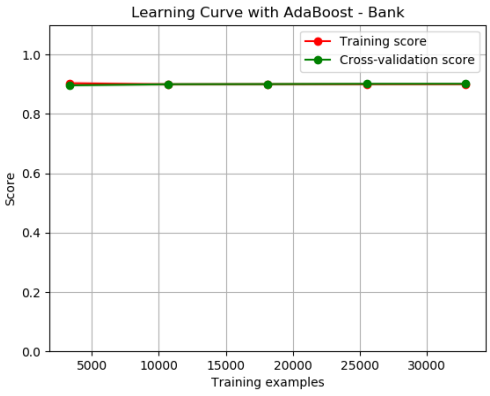


Figure 27

LETTER DATASET

Finally, the AdaBoostClassifier was tuned and tested on the letter dataset, with some interesting results.

Ensemble classification is usually a good approach for when there is a lack of training data. The results from the other classifiers on the letter dataset suggests that boosting will be a good way to reduce the gap in accuracy related to the minor lack of samples in the letter dataset.

I started with tuning the base estimator again, using validation curves on the max depth of the decision tree estimators. I found that pruning the depth did in fact result in better accuracy, surprisingly, performing best at a **max depth of around 20**. Pruning max leaf nodes resulted in strictly worse scores, so I did not use that pruning parameter of the weak learner.

The final parameter to tune was the number of estimators to use. There was an upper bound on performance found at **100 estimators** in this case (see *Figure 29* for the validation curve).

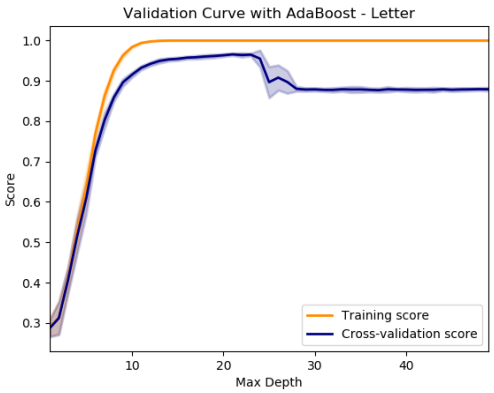


Figure 28

Using 100 estimators with a decision tree of max depth 20, I generated a final learning curve, and found nearly perfect results. The

classifier was able to get 100% accuracy on the training data, and the cross-validation score almost converges to the same rate. With more data, it's likely the classifier could approach 100% accuracy in cross-validation score.

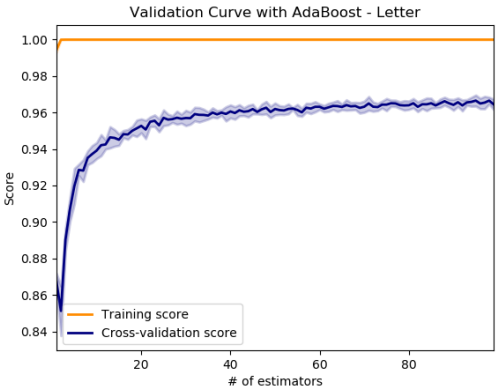


Figure 29

After tuning the hyper-parameters of the AdaBoostClassifier with decision tree base estimators, a final learning curve was generated to observe how the learner behaves as training examples increase. In *Figure 30*, the training curve shows the cross-validation accuracy slowly approaching 1.0, with a small amount of error leftover.

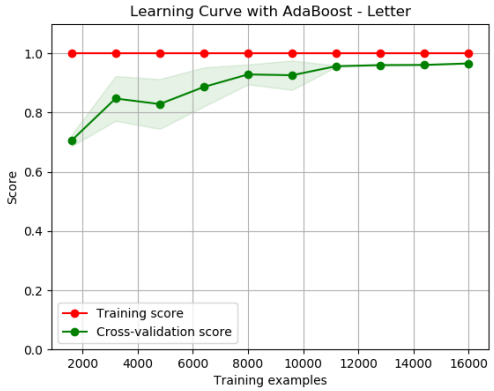


Figure 30

WHAT DOES THIS ALL MEAN???

BANK DATASET

To summarize the results, it's important to revisit the differences between the two datasets.

As discussed in the introductory section of the analysis, the two datasets have vastly different population distributions. The bank dataset is especially problematic regarding this point – it's very difficult to determine which classifier is the "best" for this dataset based on accuracy alone. Even defining "best" is tricky here; how exactly does one determine what "best" results are for marketing?

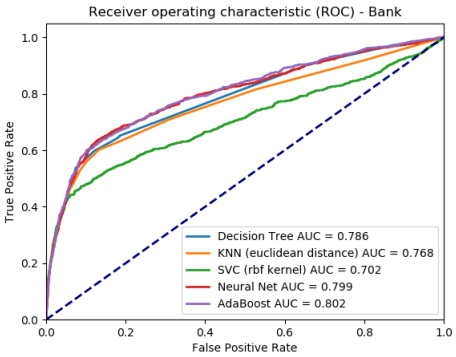


Figure 31

Given the limitations of the data, there is likely no way to provide perfect accuracy for the bank dataset, and as such, some error must be allowed through. There are two types of errors with respect to a binary classification problem, such as the bank problem: type I errors (i.e., false positives), and type II errors (i.e., false negatives). To determine which is "better" we must examine the purpose of the analysis. If the bank, for example, determines the risk of a false positive is very low, we could "improve" our results by sacrificing accuracy to allow more potential positives through, even at the cost of having more false positives. Through my testing, I did observe that overfitting the bank set, in general, tended to create more type 1 errors and have a higher number of overall positives, however, the hyperparameter tuning was based on accuracy/precision.

To examine the performance of the algorithm separately from accuracy alone, a Receiver Operating Characteristic (ROC) curve can be used (*Figure 31*). The curve displays the tradeoff with false positive rate as the classifier increases its true positive rate. The area under this curve (AUC), represents the probability that a true positive is labeled a positive by the classifier. For the bank dataset, AdaBoost is the classifier most likely

```
Benchmark Default AdaBoost Test
Train Accuracy = 0.995083459788
Train Confusion Matrix:
[[29213    35]
 [  127  3575]]

Test Accuracy = 0.878489924739
Test Confusion Matrix:
[[6966  334]
 [ 667  271]]
```

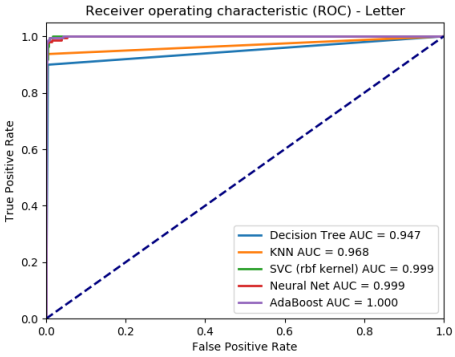
```
Tuned AdaBoost Test - Single Iteration
Train Accuracy = 0.900303490137
Train Confusion Matrix:
[[28837   382]
 [ 2903   828]]

Test Accuracy = 0.901796552561
Test Confusion Matrix:
[[7234   95]
 [ 714  195]]
```

Figure 32

to correctly classify a true positive, followed closely by the neural net (the decision tree did not do too shabby either…). All the classifiers are comfortably better than random (the diagonal line on the graph).

Further, we can take the tuned AdaBoostClassifier, after determining it fit the dataset the best, and compare it to an untuned classifier with respect to precision and recall. From the confusion matrix $\begin{pmatrix} True- & False+ \\ False- & True+ \end{pmatrix}$ in *Figure 32*, the **benchmark classifier had a recall of .29 and precision of .45** on the test data, whereas the **tuned classifier had a recall of 0.21 (ouch) and precision of 0.67**, *despite the higher accuracy.* In other words, the benchmark recalled more true positives, at the cost of being less precise overall. In short, one must ask what is more important, avoiding false positives, or avoiding false negatives, when tuning a classifier. The classifiers in my analysis were tuned towards precision, but other classifiers may have better recall.




```
Precision:
[ 0.97452229  0.93670886  0.94964029  0.96511628  0.97122302  0.98734177
  0.9047619   0.99310345  0.98473282  0.9691358   0.98787879  0.94666667
  0.91724138  0.97315436  0.98684211  0.96747967  0.98802395  0.97972973
  0.96551724  0.92857143  1.          0.99358974  0.99310345  0.98051948
  0.94        1.        ]
Recall:
[ 0.94444444  0.96732026  0.97058824  0.99401198  0.97122302  0.98113208
  0.97435897  1.          0.9020979   0.98742138  0.98787879  0.97931034
  0.94326241  0.97315436  0.94936709  0.92248062  0.97633136  0.9602649
  0.98245614  0.95705521  0.96153846  0.97484277  0.97959184  0.9869281
  0.9527027   0.98314607]
F1 Score:
[ 0.95924765  0.95176849  0.96        0.97935103  0.97122302  0.98422713
  0.9382716   0.99653979  0.94160584  0.97819315  0.98787879  0.96271186
  0.93006993  0.97315436  0.96774194  0.94444444  0.98214286  0.96989967
  0.97391304  0.94259819  0.98039216  0.98412698  0.98630137  0.98371336
  0.94630872  0.99150142]
```

Figure 33 – ROC + Tuned AdaBoost Test Results

Letter Dataset

The letter dataset does not require as careful of an examination; tuning for accuracy was enough for fantastic results. Due to there being 26 classes, high accuracy is nearly impossible to achieve consistently through random selection, especially with a semi-uniform distribution of classes, such as in the letter dataset. We can see that the AdaBoost classifier achieves a 1.0 AUC. This does not guarantee perfect results, but the classifier is good enough such that it has nearly 100% chance of predicting a class correctly without the false positive rate increasing. Examining the precision, recall, and F1 score (blend of precision and recall), we can observe that the classifier is excellent for all classes, even achieving some perfect precision/recall scores. The classification problem in the letter dataset was much easier for a classifier to learn than the bank dataset, overall.

Final Thoughts & Considerations

Despite the fact the two classification problems are profoundly different, AdaBoost won out so far for both. However, some other classifiers were extremely close. Assuming the difference is chalked up to randomness, which one should be picked?

In *Figure 34* (finally the last one), a final comparison is made. For the bank datasets, the error rates are equal amongst the classifiers, however, time to train and compute was negligible for the AdaBoost classifier, making it the winner for the bank dataset. For the letter dataset, again, the AdaBoost classifier wins in terms of both minimization of error, and shortest combined time to run both train and query, assuming one cares about both. TLDR; AdaBoost is the champion of these datasets (for precision)!

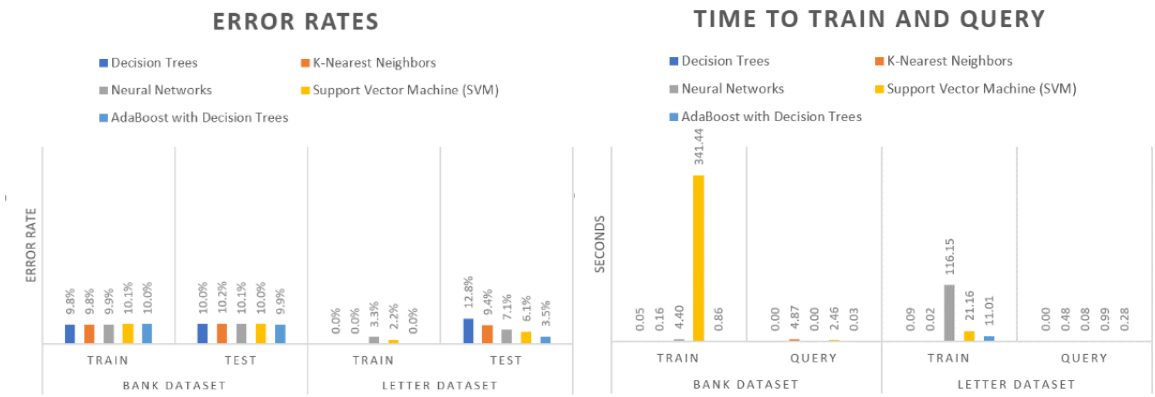


Figure 34 – Error Rates and Train/Query Times Compared