Joshua Philipose

Dr. Charles Isbell

CS 4641

2/10/19

Supervised Learning Analysis

Classification Problems

Credit Card Defaults (abbreviated as CC)

The credit card defaults binary classification problem attempts to determine if a credit card client will default on their next credit card payment. The dataset consists of 23 attributes, ranging from credit, gender, age, several previous bill statements, amounts of previous payments, and other relevant data points. With 30,000 entries (6,636 defaulting, 23,364 not defaulting), this dataset requires attention to efficiency and runtime. Predicting credit card defaults was an interesting problem to me as I spent the previous summer working as a technical product manager for Citibank's credit card rewards program, and I am interested in working on credit cards in the future. There was also the imbalance of classifications one can expect in real world problems as the dataset had almost triple the number of non-defaults as defaults, so lessons learned here would apply to future problems I encounter. Furthermore, this classification problem has direct applications in industry, as financial institutions need to manage the collective risk of their clients and allocate monetary reserves accordingly.

King-Rook vs. King-Pawn (abbreviated as KK)

The king-rook vs. king-pawn binary classification problem attempts to determine if white, who has both a king and a rook, can win the game against black, who has a king and a pawn. However, the twist that makes this problem interesting is that black's pawn is always on a7 – one move away from promoting to a queen. As the problem's premise is that it's white's turn to move, the problem essentially boils down to the ability of white in preventing black's pawn from promoting while maneuvering a checkmate or draw. As an avid chess player, I find this problem fascinating because of its confinement to a complex rules-based environment. Furthermore, I've actually been in this position in game before, and I'm curious as to the win prediction for setups different to the one I had. There are 3196 instances, almost evenly split between 1669 instances with a white win and 1527 instances with a black win. For each of these instances, there are 36 attributes describing the board and piece positions. Because of this dataset's use of true/false values and a column of three categorical values, I had to use one-hot encoding to transform the dataset into numerical format to run on algorithms provided by scikit learn.

Algorithm Analysis

Each of the five implemented supervised learning algorithms (decision trees, boosting, neural networks, support vector machines, and k-nearest neighbors) was run multiple times, with different hyperparameters, sample sizes, and data conditions. I chose to vary sample sizes to determine the impact lack (or excess) of data would have on our accuracies, and the hyperparameters I was familiar with were varied to deepen my understanding of their impact. Accuracy was calculated by the percentage of instances classified correctly, as correct classification was my main goal. All algorithms were run in a Jupyter notebook, implemented by scikit learn, and run on data from UCI's machine learning repository. The test/train split was 70/30 for all algorithms after random selection, with 5 folds whenever cross validation was used.

Decision Trees (Pre-Pruning)

Decision tree algorithms produce trees that classify new inputs based off categorical decisions about their attributes. The attribute is selected for a decision is determined by the information gain provided by each attribute at that node. To begin my experimentation with decision trees, I used a vanilla version of scikit learn's decision tree classifier on both my datasets. This function provided me with the following data:

	Training Accuracy	Testing Accuracy
Credit Cards	.999	.728
Chess	1.000	.998

	Training Time (s)	Training Prediction Runtime (s)	Test Prediction Runtime (s)
Credit Cards	.78	.01	.01
Chess	.01	.001	.001

After these initial experiments, I began to manipulate hyperparameters to gauge their impact. Due to the lack of post-pruning options within scikit learn, I focused on pre-pruning options, despite their lower theoretical efficacy. First, I changed the maximum depth of the decision tree in a range from 1 to 9, with a step size of two, as demonstrated in figures 1 and 2. As you can see, training accuracy crossed testing accuracy when maximum depth was approximately 6 in fig 1, indicating that a maximum depth of 6 is optimal for the credit card dataset. As expected, testing accuracy decreases as max depth increases, because a higher maximum depth allows for more overfitting to the idiosyncrasies of the training sample distribution. This is further proven by the fact that training accuracy is positively correlated with maximum depth. In fig 2, we see a beautifully interesting phenomenon; training accuracy and testing accuracy increase in lockstep as maximum depth increases. This indicates that the distribution of the training sample is near identical to the distribution of the testing sample. Because chess is a closed, rule-bound environment, there's negligible noise, and variances will be low if both players play optimally.

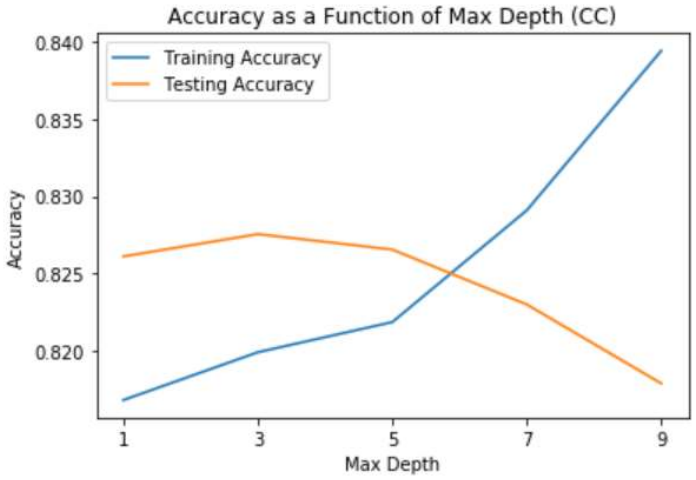


Fig 1

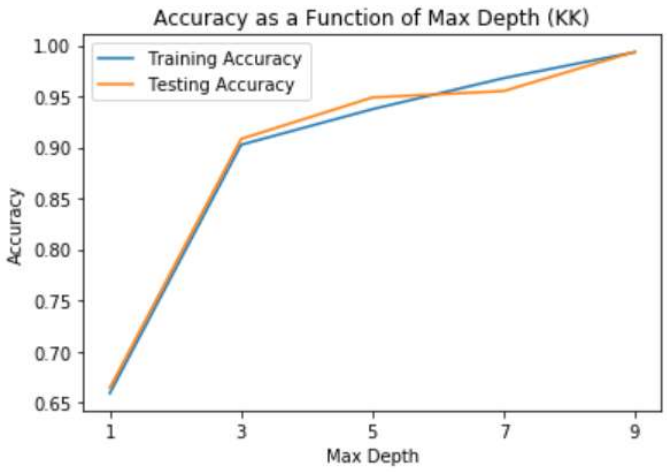


Fig 2

Another method of pre-pruning is changing the minimum sample split. The minimum sample split is the minimum number of instances needed to split after a certain node. For example, if minimum sample split is 6, there would need to be 6 samples at a node in order to split them further – if there are less, that node will result in a leaf node. In fig 3, we see that testing accuracy crosses training accuracy for our credit cards dataset when minimum sample split is approximately 65. This means that once the minimum sample is 65, further node generation will result in overfitting to the nuances of the training sample. In fig 4, we see a different effect. In our chess data set, we want to be as fitted to the training data as possible, since the distribution of the training set is the same as that of the test set. Therefore, as minimum sample split increases, thereby decreasing fit to the training data, accuracy for both sets decreases.

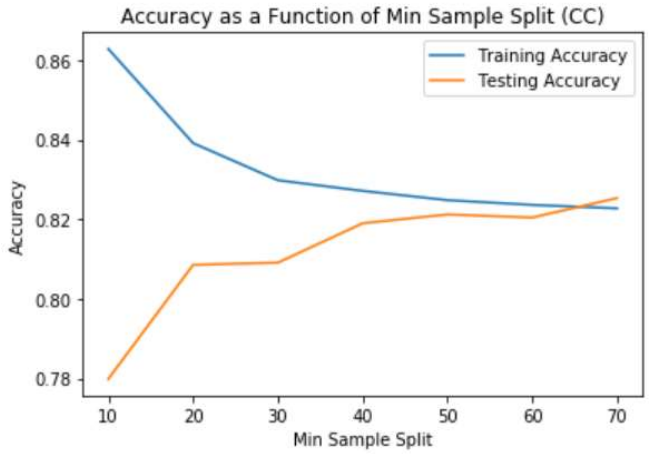


Fig 3

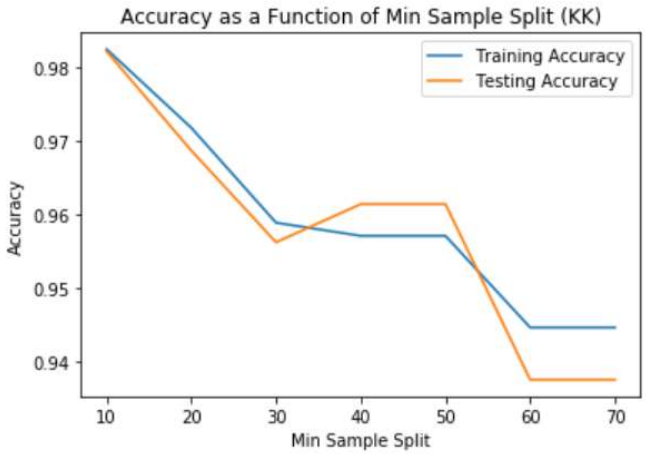


Fig 4

After experimenting with pre-pruning, I wanted to find the optimal levels for the maximum depth and minimum sample size hyperparameters. In order to accomplish this, I used cross validation to come up with an accuracy score for a decision tree crafted with each combination of the maximum depth and minimum sample hyperparameters, and then used the tree with the best score. The optimal tree for our credit cards dataset ended up having a maximum depth of 3 and a minimum sample size of 10, while its accuracy on test data was .827 compared to the accuracy of .728 for the vanilla tree. For our chess data set, our optimal tree had a maximum depth of 9 and a minimum sample value of 10, and it correctly classified the data with accuracy of .98 on the test set, while the vanilla tree had an accuracy of .99. The optimal tree most likely performed worse than the vanilla tree for our chess dataset because of the fact that a minimum sample size of 0 was not an option for the cross-validation optimizer, thereby not allowing the tree to perfectly fit the training distribution, which as mentioned before, perfectly fits the test distribution. This unexpected result perfectly exemplifies the need for domain knowledge when assigning the hyperparameters to a learning algorithm and defining the hypothesis space to search.

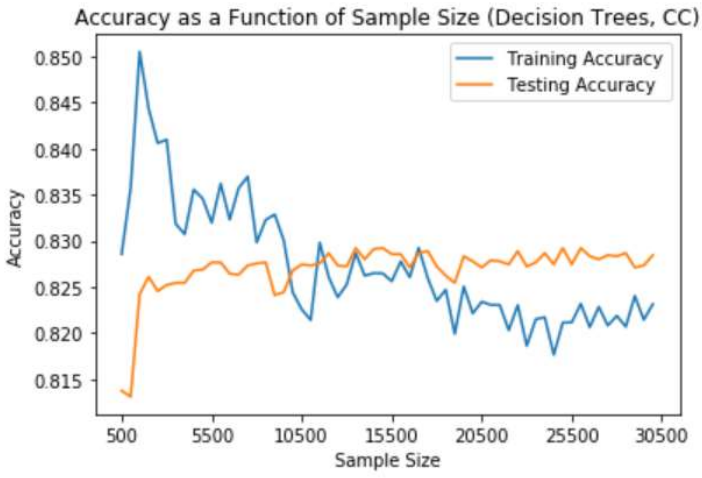


Fig 5

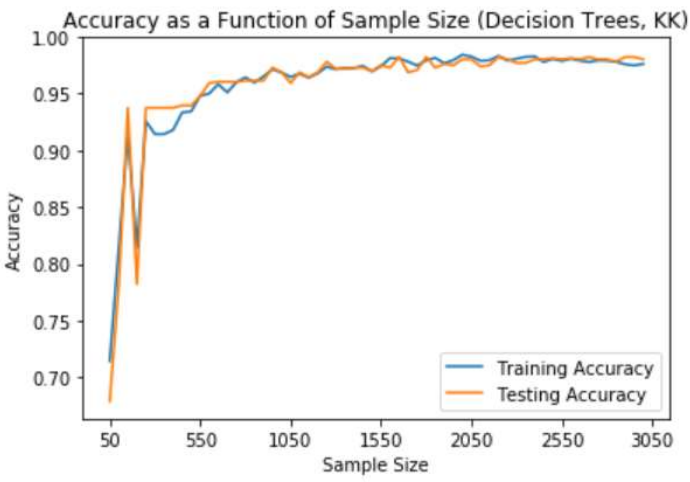


Fig 6

The final hyperparameter I altered was the sample size used to train the decision tree. The sample sizes you see above were further divided into the classic 70-30 training/validation sets, and the resulting decision tree was tested on the entire original test sample. As you can see in fig 5, the optimal sample size for decision trees with our credit card dataset is 10,000 samples and above, for this is where testing accuracy starts to beat out training accuracy as the training sample's distribution approaches the testing sample's distribution. In fig 6, we can see that the optimal size of our chess dataset is about 750 and above, for the same reason. However, we can see that the learning curve sharply increases, and then plateaus with only a slight incline after 750 samples. This is because our algorithm has already accurately approximated

the test sample distribution to a large extent with 750 samples, and while further instances flesh out that approximation, the core knowledge is already all factored in.

Boosting (AdaBoost)

The implementation of the boosting algorithm I chose to use was adaptive boosting, also known as AdaBoost, with decision trees as the underlying weak learner. The AdaBoost algorithm essentially exponentially increases the "focus weights" on instances that our weak learner performs poorly on, making them the focus of our weak learners in the next iteration of the algorithm. We typically further weaken the underlying decision trees by limiting their maximum depth and employing other aggressive pruning strategies, resulting in an ensemble of diversely fitted but individually weak learners. In theory, through successive iterations, our ensemble of diverse weak learners perform much better than a single "strong" learner, as each weak learner becomes better at utilizing particular attributes. As I began my experiments with boosting, I ran the vanilla AdaBoost algorithm on both my datasets and came up with the following values.

	Training Accuracy	Testing Accuracy
Credit Cards	.845	.806
Chess	.999	.992

	Training Time (s)	Training Prediction Runtime (s)	Test Prediction Runtime (s)
Credit Cards	9.93	.21	.09
Chess	.47	.03	.02

As we can see, though our testing accuracy was on par with or greater than our testing accuracy for decision trees, our runtimes were an order of magnitude higher.

After these initial tests, I started changing hyperparameters to see how the results would vary. I first changed the number of estimators, which are the weak learners composing the ensemble used by AdaBoost. For both our datasets, the training accuracy was lowest with only one estimator, which is consistent with the theory. However, we see an interesting phenomenon occurring in fig 7, as test accuracy peaks with one estimator, and trends downwards afterwards. This may be because the max depth parameter for AdaBoost was 4, which is already optimal as a decision tree alone. Adding further decision trees to the ensemble when one is already optimal may cause confusion in classification. In fig 8, we see the learning curve expected by AdaBoost theory. Since the decision trees are throttled with a maximum depth lower than the optimal maximum depth for the chess dataset, an increased number of estimators allows each tree to fit itself to a different subset of the total attributes, allowing the ensemble to be a better predictor as the number of estimators increases.

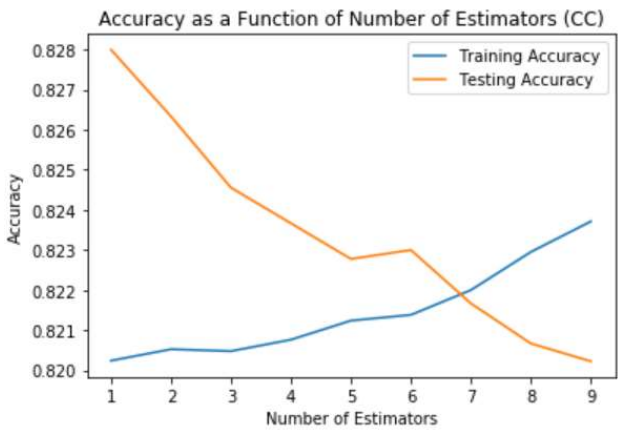


Fig 7

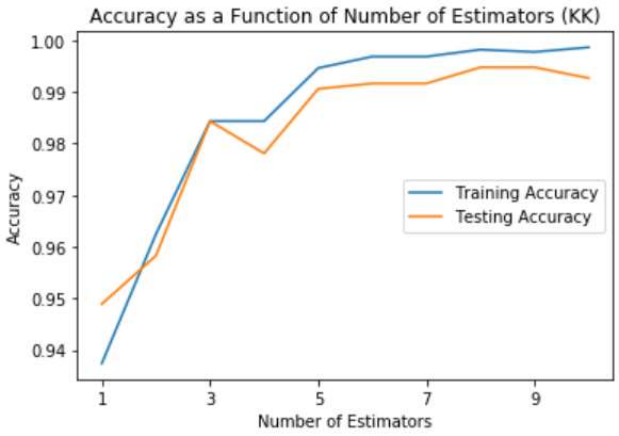


Fig 8

As mentioned before, we manipulated the number of estimators used in our boosting algorithm. Now, we want to look at changing the hyperparameters of the estimators themselves, namely our underlying decision trees. To this effect, I manipulated the maximum depth of the decision tree estimators, then cross validated the resulting trees with 5 folds to achieve the validation scores seen in fig 9 and fig 10. Fig 10, while it looks spectacular, is not what it seems to be on first glance. The cross-validation scores on change by approximately .005 from peak to trough, indicating a fairly smooth horizontal line across max depths that is greatly exaggerated by the scale of the y-axis. This indicates what we already know – the chess dataset is not prone to overfitting, because as the max depth increases, so too does the fitting to the training data. However, fig 9 shows us that as our boosting algorithm fits individual learners to the nuances of the training data, overfitting occurs, and accuracy drops.

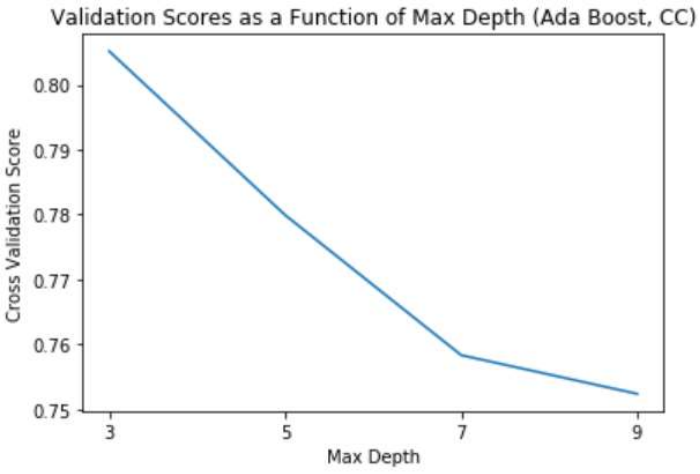


Fig 9

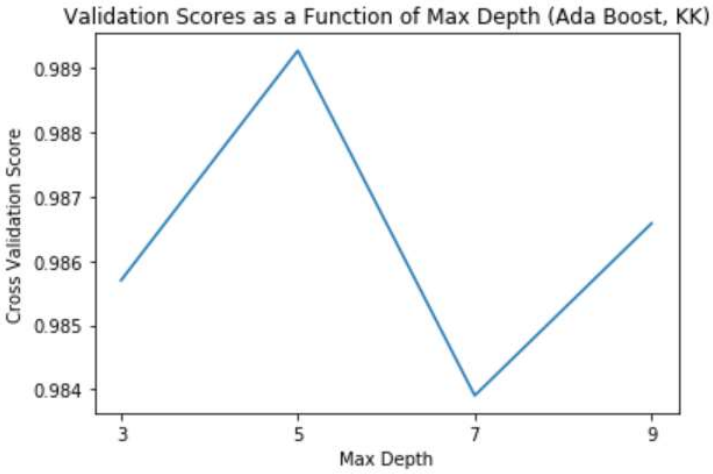


Fig 10

Next is an experiment done to observe the effects of sample size on accuracy of our boosting algorithm. This experiment reinforces what we learned with decision trees: the sample size we need for optimal accuracy in the test set is approximately 10,000 instances for our credit cards dataset and 750 instances for our chess dataset.

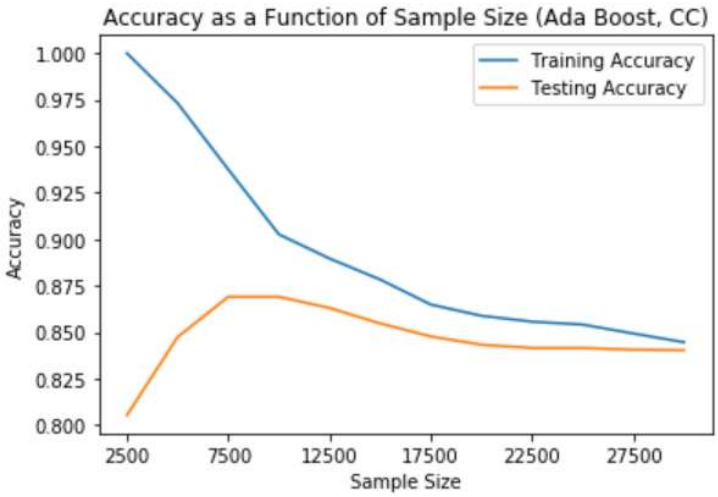


Fig 11

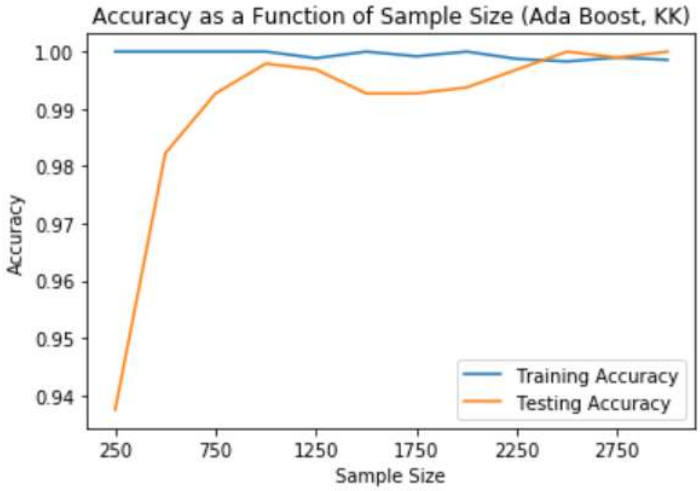


Fig 12

Neural Networks (ANN)

Artificial neural networks are graphs composed of nodes attempting to replicate the function of neurons in biological brains. Each node accepts inputs that trigger certain outputs according to the node's activation function, which are then passed to the next layer of nodes. According to error functions, each node's weight is updated in a process known as backpropagation. If the nodes are perceptrons, each node is a linear classifier – an interesting effect of this fact is that a

single perceptron can perfectly classify a sample if the sample is linearly separable. Neural networks are therefore extremely powerful – in fact, they can represent any arbitrary function.

To begin my experimentation with decision trees, I used a vanilla version of scikit learn's neural network classifier on both my datasets. This function provided me with the following data:

	Training Accuracy	Testing Accuracy
Credit Cards	.771	.777
Chess	.993	.986

	Training Time (s)	Training Prediction Runtime (s)	Test Prediction Runtime (s)
Credit Cards	1.64	.05	.03
Chess	2.03	.01	.005

Thus far, our boosting algorithm has given us the most accurate results, at the expense of taking the most time. Our decision tree algorithm has also taken the crown in runtime (or lack thereof), leaving the much-renowned neural network in third place out of the three algorithms we've covered so far. Let's take a closer look at how neural networks perform with varying hyperparameters.

It is known that a neural network with only one hidden layer can converge upon a solution regardless of the sample. With that in mind, I varied the size of the single hidden layer from 20 to 200 nodes, in increments of 20. Let's begin with fig 14, as it is the easier of the two to understand. As the number of hidden layers increases, the neural network has more capacity to extract value from the data it is given, resulting in an increase in both training and testing accuracy as expected. As the number of perceptrons increase, the functions it can represent become more and more complicated. This may help explain our results from fig 13 – there are some optimal amounts of perceptrons that can represent functions analogous to our target function, but at times the amount of perceptrons result in a function complexity that cannot fit the data. It also may be that our credit cards dataset is very difficult to linearly separate, and that we need far more data to feed into our neural network.

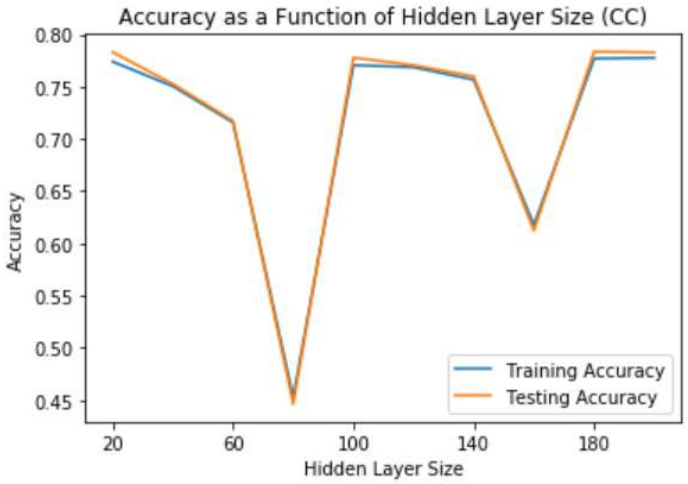


Fig 13

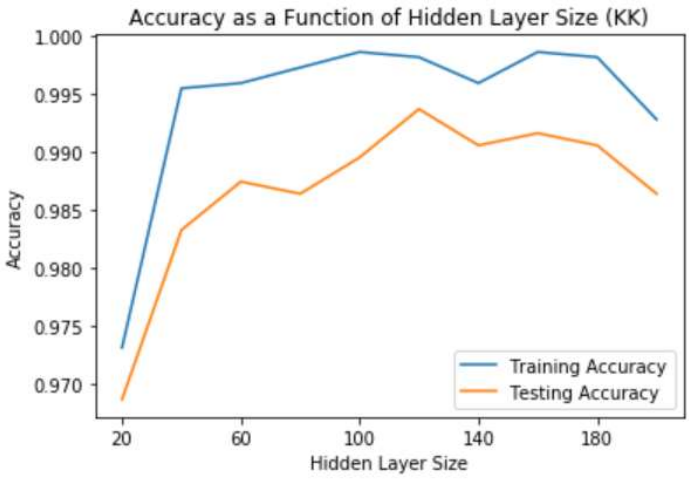


Fig 14

Now that we've gained more insight into how the number of hidden layers affects the accuracy of our algorithm, let's look at how our sample size does the same. Fig 16 shows us a clean learning curve, revealing to us that as sample size increases, neural networks are able to approximate the test distribution and target function more accurately. This curve also reinforces the same information we learned from prior learning algorithms – 750 samples and above is optimal for our chess data set. Unfortunately, fig 15 seems to be suffering from the same ailment that is causing the large oscillations in fig 13. The results in fig 15 directly contradict our established notions of ideal sample size for the credit

card data set and may be attributed to the random initialization of values that are characteristic of neural networks. Since I kept my random seed constant across the entire project for sake of consistency, we may have just gotten unlucky with our initial weights for our neural network, and then settled in local minima afterwards for certain iterations. The graph in fig 15 does roughly match the graph in fig 13, so we do know that there's consistency between the two.

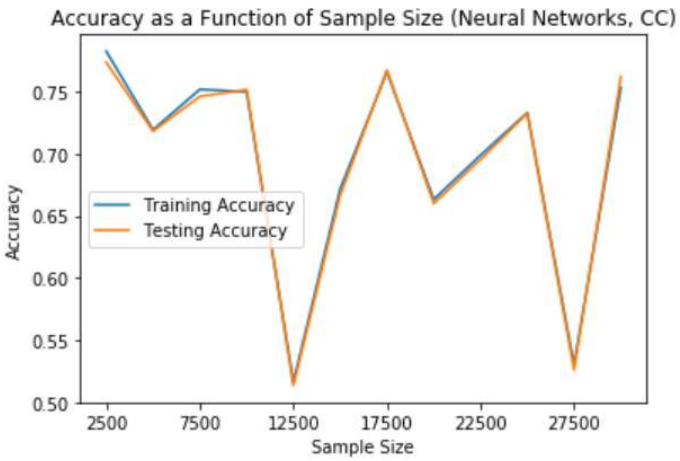


Fig 15

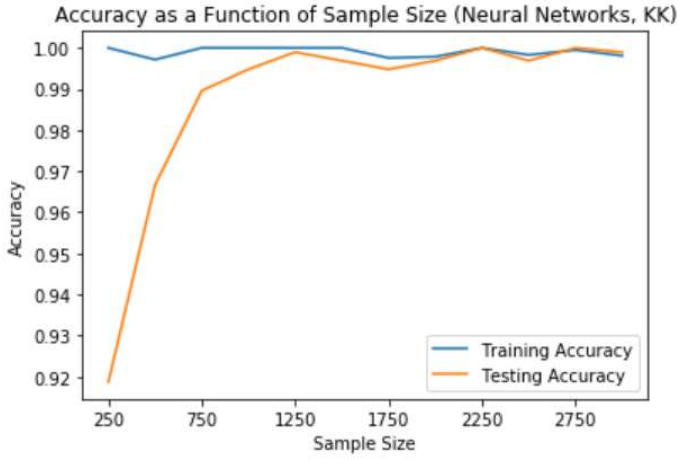


Fig 16

After running the above experiments, I became curious about what the optimal neural network for our datasets would look like. To obtain the optimal neural network, I ran a cross validation setup to obtain the validation scores of multiple neural networks with varying learning rates and hidden layer sizes. For this, I'll spare you the graphs and tables – the optimal neural network my experiment turned up with ended up being the default neural network classifier.

Support Vector Machines (Sigmoid and RBF Kernels)

Support Vector Machines attempt to classify instances by linearly separating beforementioned instances into classifications with a hyperplane. If it is not possible to separate the instances in the current dimension of operation, SVM uses a custom kernel function to project those instances into a dimension where it is possible to separate the data with a hyperplane. SVM is unique among the algorithms we have discussed thus far because instead of only trying to find a hypothesis that fits the data, it also looks for the consistent hypothesis that offers the least commitment to the data. As I'm experimenting with both sigmoid and RBF kernel functions, I ran the vanilla version of both on my datasets and came up with the following data.

SIGMOID	Training Accuracy	Testing Accuracy
Credit Cards	.656	.660
Chess	.902	.925

SIGMOID	Training Time (s)	Training Prediction Runtime (s)	Test Prediction Runtime (s)
Credit Cards	8.24	5.57	2.30
Chess	.373	.23	.10

RBF	Training Accuracy	Testing Accuracy
Credit Cards	.994	.796
Chess	.935	.946

RBF	Training Time (s)	Training Prediction Runtime (s)	Test Prediction Runtime (s)
Credit Cards	192.57	43.37	20.62
Chess	.35	.19	.08

As you can see, while our accuracy is passable it does not measure up to the boosting algorithm previous employed, and has runtimes that far supersede those of neural networks. The RBF kernel in particular has quite a long runtime for our credit card dataset, most likely due to the difficulty of projecting 30,000 datapoints into a plane where it is linearly separable. This hypothesis is backed up by the fact that the chess dataset, which is both cleaner in terms of prior predictions and lesser in volume. I also want to emphasize that the two kernel functions I selected were preset, but SVM is completely supportive of individuals using their own kernel function. The amount of domain knowledge that can be used to optimize SVM through a custom kernel function is far more than the other algorithms we've covered so far.

After experimenting with both kernels, which were both admittedly picked because their runtimes were faster than the other kernel functions for my datasets, I chose to proceed with the RBF kernel. The next hyperparameter I wanted to manipulate was the penalty hyperparameter, typically called c. As the penalty parameter increases, the target margin between the linearly separated instances increases. This results in a tighter fit to the training data and increased runtime as more hyperplanes must be considered when the margin is lower. As the penalty parameter decreases, the opposite occurs.

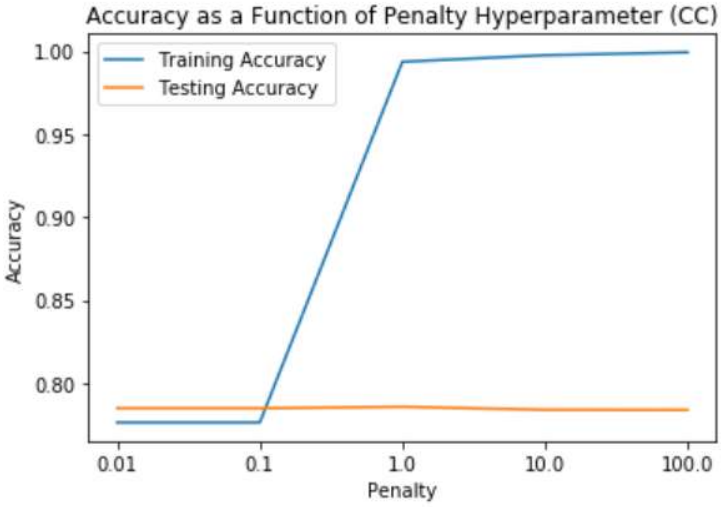


Fig 17

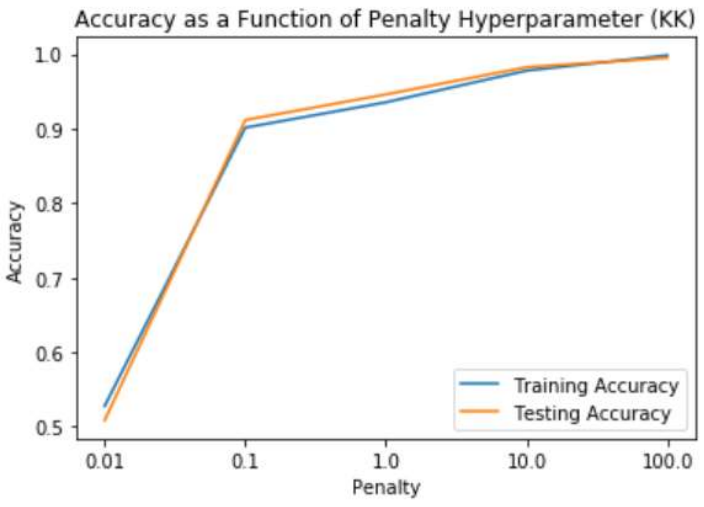


Fig 18

Fig 17 shows us an interesting detail about our credit card dataset. Regardless of our penalty parameter, and by extension margin, we come up with the same testing accuracy. We can infer that the hyperplane SVM used to separate the data was very similar in all of these cases. Therefore, we can infer that there's a margin larger than the maximum margin we use already separating the instances in the dimension we're currently exploring. In fig 18, we see that as our penalty parameter increases, accuracy in both test and training also increases. This is expected, as we know from our previous experiments that our test and training distributions are nearly identical in our chess dataset, and of course as our required margin decreases and we fit closer to our training data, we will establish higher accuracy for the training data, and by extension to our test data by virtue of this dataset.

Finally, let's vary sample size and see what values we get. Fortunately, our results confirm the results of our previous 3 algorithms: our credit cards dataset needs 10,000 or more instances to be optimal, and the chess dataset needs 750 or more instances to reach an optimal result. I did not cross validate on SVM because the accuracies obtained were on par with those of the neural network algorithm, and because the runtime of a single SVM function call implied a far greater runtime when cross validations were pulled into the mix.

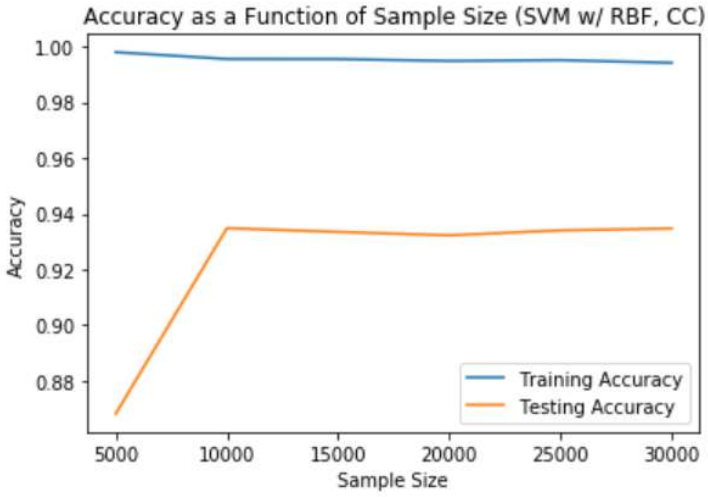


Fig 19

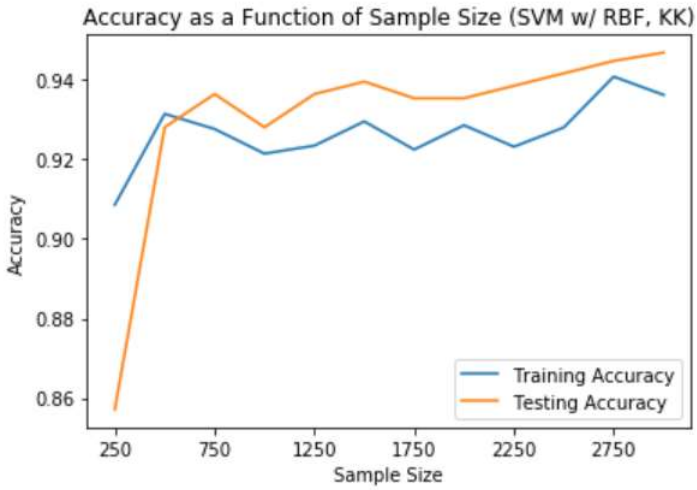


Fig 20

k Nearest Neighbors

K-nearest neighbors operates by finding the k instances "closest" to a new input, and then having those k instances vote on the classification of the new input, weighted by some distance function. To start off my experiment with kNN, I ran the vanilla kNN classifier function from scikit learn.

	Training Accuracy	Testing Accuracy
Credit Cards	.814	.759
Chess	.978	.958

	Training Time (s)	Training Prediction Runtime (s)	Test Prediction Runtime (s)
Credit Cards	.09	2.86	1.34
Chess	.02	.70	.30

Right off the bat, you can see that the above table is different than those of the other algorithms, because the runtime section shows us that querying takes us far longer than training. This is because kNN doesn't employ a training algorithm, but simply stores all training data. Query takes a while because this is where the actual work is done: the k nearest neighbors to the new input must be found, and a classification vote must be cast and weighted. kNN is not better than the other algorithms in accuracy or runtime – boosting remains the most accurate, and decision trees remain the quickest algorithm.

In figures 21 and 22, we see the results of varying k. In both figures, we see that when k is 1, we have perfect training accuracy. This makes perfect sense, because kNN literally stores the training data – if k is one, it simply returns the one value that we already had stored for those attributes. As we increase the value of k, more and more instances are considered in our final classification, reducing our training accuracy but increasing our testing accuracy. For example, when k is 5, it makes sense that training accuracy is reduced – we're considering the training point itself and 4 values that aren't the training point, which gives us a shakier answer than the training point alone under k = 1.

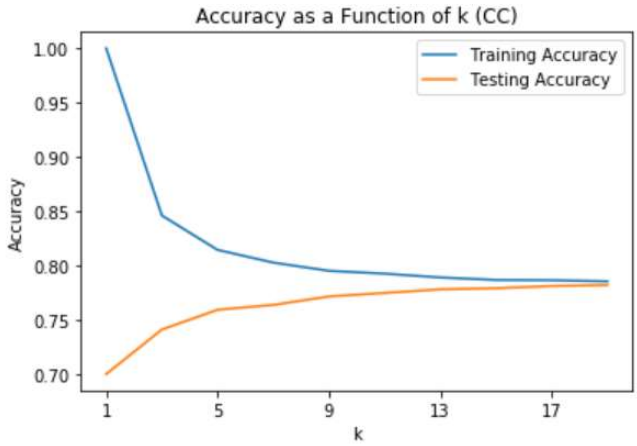


Fig 21

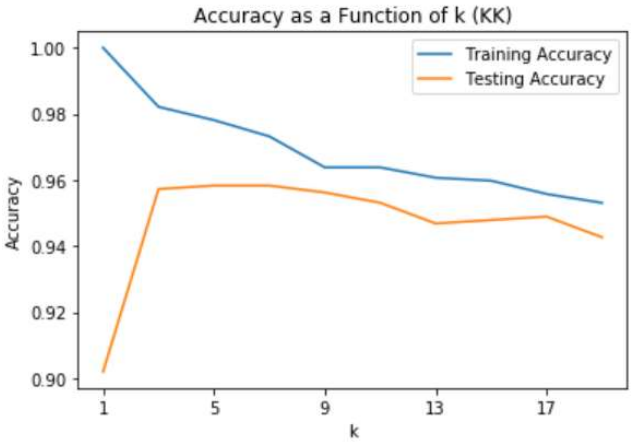


Fig 22

However, up till now we have been using scikit learn's default distance function, which uniformly weights every instance in the dataset. However, this doesn't make sense, as an instance that is quite far from our input may be selected if k is large enough, and that instance should not have the same voting power as an instance close to our input. Therefore, we'll run an experiment to determine the differences in accuracy between a uniform distance function and a distance weighted distance function, which weights each vote cast by an instance by the inverse of their distance to the new input. As we can see from the figures below, our distance weighted kNN algorithm (indicated by the orange line) performs better than the uniformly weighted kNN algorithm. The figures also bolster the sample size insights gained during the prior four algorithm analyses.

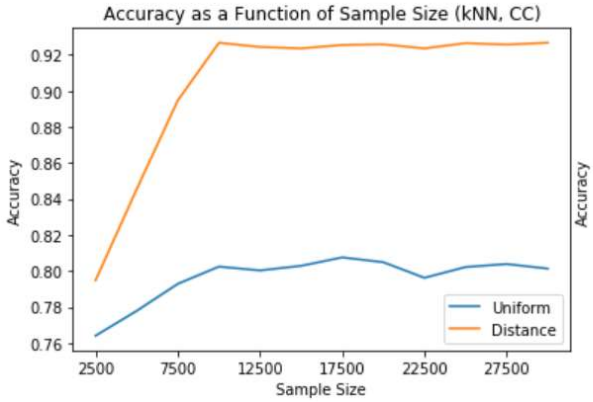


Fig 22 above, Fig 24 below

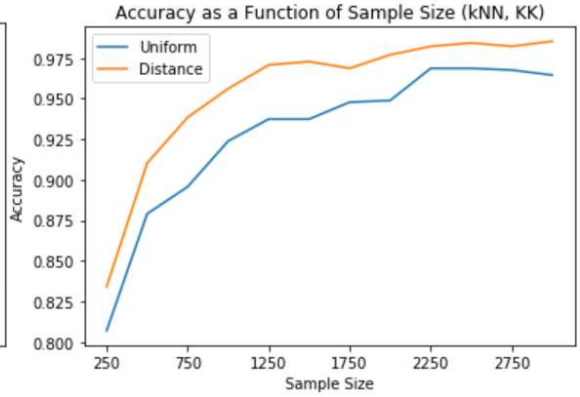
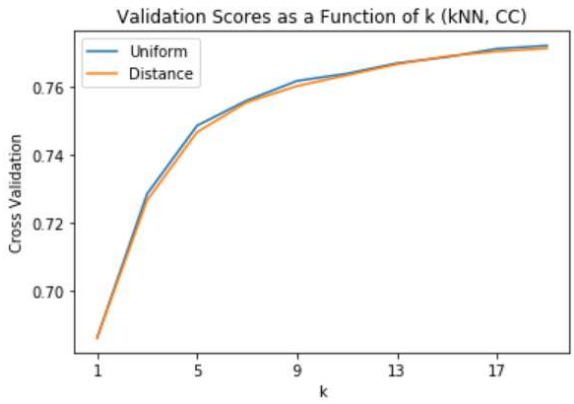
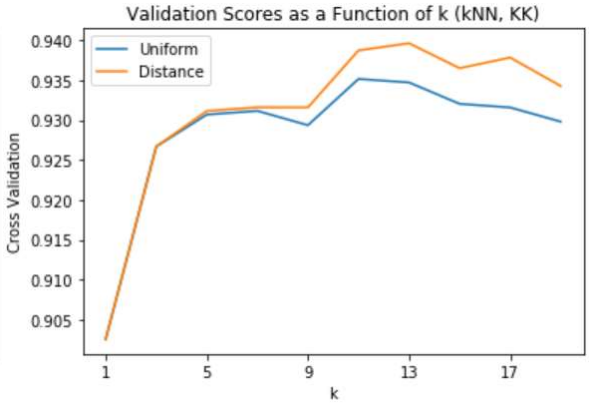


Fig 23 above, Fig 25 below





It should be mentioned, however, that the kNN algorithm we're using still isn't optimal, as it assumes that all attributes in an instance are equally important. More accurate kNN algorithms will weight each attribute accordingly when considering distance weighting.

Conclusion

Overall, every algorithm proficiently classified both datasets, and I'm glad we were able to see the relative strengths and weaknesses of them all. Furthermore, I gained a good amount of confidence in creating these projects from scratch, as we weren't provided any type of framework, and I had to write everything using my wits and googling ability. I'll definitely be using some of these classifiers to explore other datasets in the future. While some quirks had to be overcome with the chess dataset, such as the first 904 instances all evaluating to a white victory, confusing the decision tree learner until I further scrambled the instance sampling, I'm happy both datasets were completely different so that we could gauge the efficacy of these supervised learning algorithms in different data conditions.