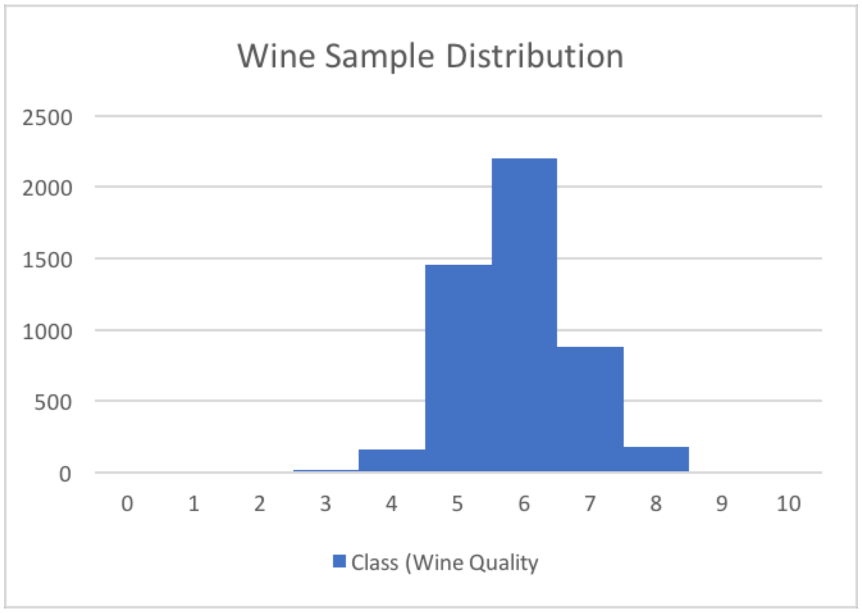
Alan Li
Project 1
CS4641

Supervised Learning Analysis

Datasets

Bank Marketing: This data is taken from the marketing campaign of a Portuguese banking institution. There are 20 attributes relating to the client, the nature of contact, and time of contact, among other things. The task is to predict whether the client eventually subscribed to a term deposit with the bank. The distribution in the bank marketing dataset is somewhat large, containing 41188 training samples. The data is unbalanced, containing exactly 4640 "yes" and 36548 "no." Another bank dataset exists, from which there are 45211 samples to compute. The data is also unbalanced, containing exactly 5289 "yes" and 39922 "no." The difference between the two datasets is that the latter, which I will call "bank-simple" has only 17 attributes instead of 20. The purpose of bank-simple is to determine how simplifying the representation of the problem can affect the classifier.

Wine Quality: This data is taken from rated wines from north Portugal. There are 12 attributes that are related to the chemical properties of the wine—all of which are quantitative. The task is to predict the rating of the wine (0 terrible – 10 excellent) based on the chemical attributes of the wine. There are in total 4898 samples, making this dataset somewhat small. From the problem definition, there should be 11 different classes, but the data itself is heavily unbalanced.



Wine Quality Simplified: I constructed another dataset called wine-simple, which is the same as the original set, except with the classes condensed into three sections: bad (0-3), medium, (4-6), and good (7-10). The data is still unbalanced (3, 430, 67 in low, medium and high from a random 500 sample). The purpose of this set is to evaluate how the number of classes in a problem can affect the accuracy of the classifier.

Importance of Chosen Datasets:

The banking dataset is important from a commercial standpoint because applying supervised ML techniques to the data will not only create a model that can predict the likelihood of getting a contract from a potential customer, but also one that can reveal certain traits or find trends that can help contractors target better customer bases. Knowing which kinds of customers are more likely to commit to contracts can also give banks insight when constructing their policies. The wine dataset is more of a niche interest, but nevertheless impactful. Developing a working model for these datasets can reveal exactly which components of certain wines contribute to the wine's appeal. This kind of information can help winemakers develop better wines.

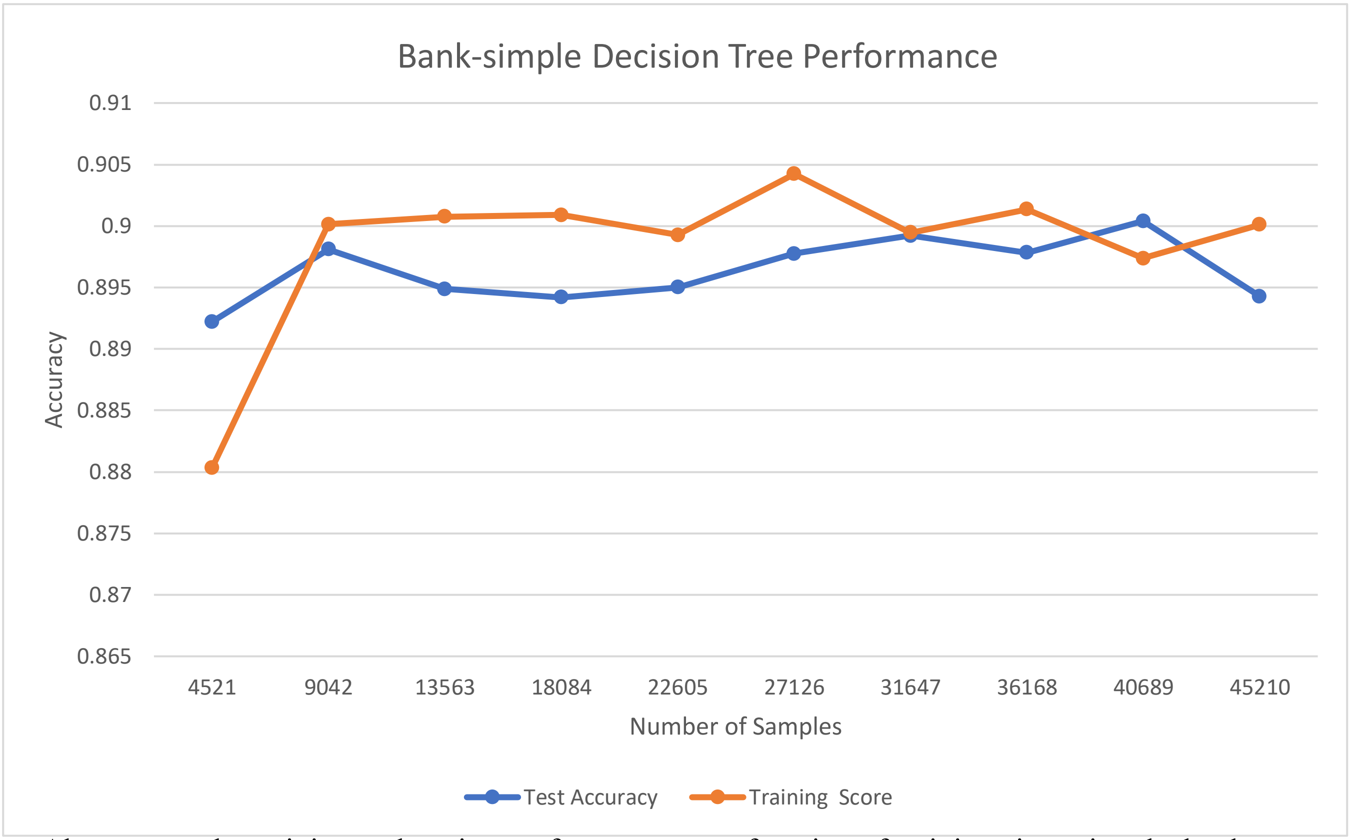
From a strictly machine learning standpoint, the banking data set is an interesting classification problem because there are many attributes involved. From the curse of dimensionality, this fact should make the task of developing a model difficult. However, some of these attributes, as I will later discuss, may be irrelevant in obtaining an accurate hypothesis. In addition, some of these attributes were strings. I thought it would be an interesting challenge to transform string attributes in such a way that they are not only valid in my algorithms, but also as close as possible to their original definition. The wine dataset is interesting because the large breadth of classes makes it particularly difficult to classify. Not only are there many classes, but the data is sparse in certain areas. As seen from the sample distribution in the graph above, some classes do not even occur. A unique challenge in this dataset will be to train a learner that can recognize and identify outlier cases (such as when a wine is extremely highly or terribly rated).

Decision Trees

Bank:
The approach I took in attempt to create a good decision tree was to do some pre-pruning. To do this, I took the bank-simple dataset and even further simplified the representation, turning it from a 17-attribute problem into a 12-attribute problem. My idea behind this move was that removing irrelevant or less impactful attributes would let the decision tree focus on the important attributes. Next, I used parameter tuning with cross validation. Parameter tuning is another way of pruning because the parameters being tuned limited the tree's depth, size, and node splitting thresholds. I tested my methods against using a decision tree with default parameters:

Dataset	# of nodes	Train time(sec)	Train acc.	Test time	Test score
Bank-simple	157	9.1877	0.9001	0.0817	0.8942
Bank-simple (default)	6529	0.1906	1.0	0.0863	0.8538
Bank	151	10.9519	0.9199	0.0936	0.9080
Bank (default)	4283	0.2484	1.0	0.1039	0.8892

The table above displays the results of the various decision trees on a shuffled 70-30 split of the data. At first glance, it might seem that the performance is remarkable; however, given the data's distribution, a naïve guesser that outputs "no" every time would achieve a 0.8830 score on bank-simple and 0.8873 on bank. Nevertheless, every tree except bank-simple with default parameters showed anywhere from a slight to noticeable improvement. The data shows that tuning parameters does improve performance slightly. My hypothesis as to why this is so is because the tuned trees are significantly pruned, and thus less-likely to over-fit the training data. The tree sizes between tuned and not tuned trees support my idea, as well as the training scores being closer to the test scores than in the not pruned trees. A drawback to the performance improvement is training time. I used Randomized Search CV instead of Grid Search CV because it runs faster; even so, training times were orders of magnitude longer. Another note is that bank-simple had slight, yet consistent performance deficits when compared to the 20-feature bank. This seems counter-intuitive due to the curse of dimensionality, but my idea is that the attributes removed were important in determining the output. Since I'm not a banker, I could but only guess which attributes were the least relevant.



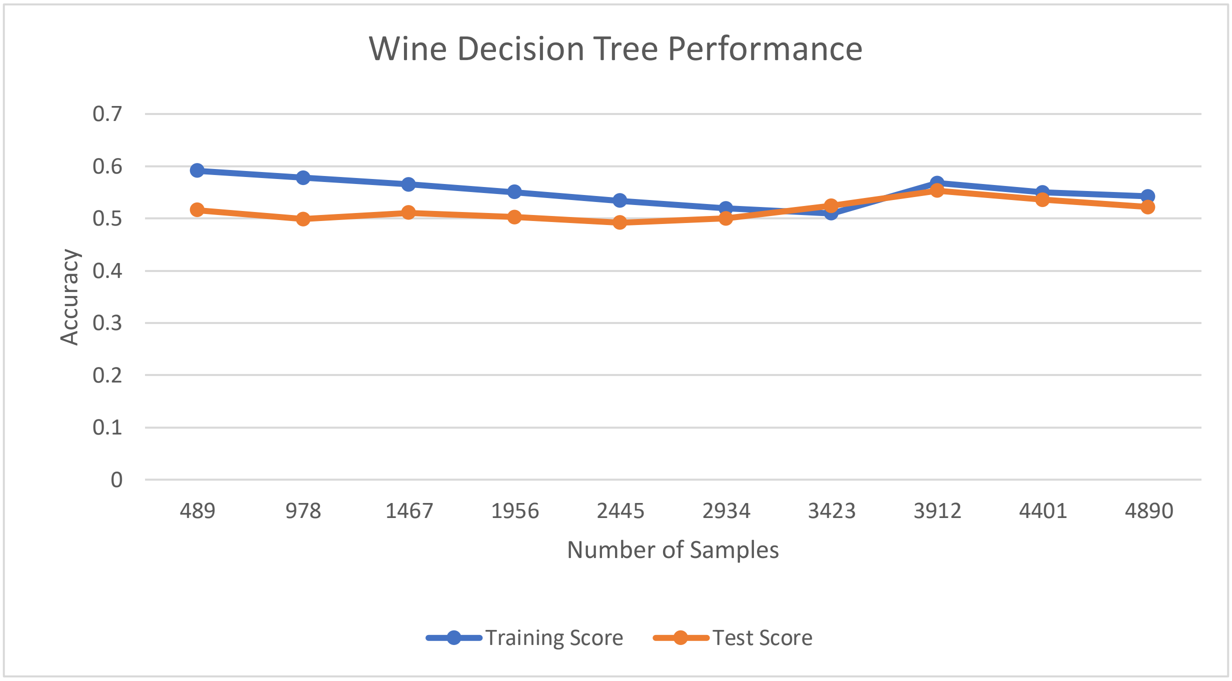
Above were the training and testing performances as a function of training size using the bank-simple decision tree with parameter tuning. We can see from the graph that the training and test performances converge somewhat. This is because the tree in bank-simple was pruned; thus, the output tree was general, i.e., it did not over-fit the data. One would think that test accuracy would improve because of increased samples, yet that is not necessarily the case. One possible reason is that Randomized Search CV is random, and thus did not always find the most optimal set of parameters. Another reason comes from Haussler's Theorem. I did an extremely rough estimate on the number of semantic hypothesis in the syntactically infinite space. Below is the number of possible values for each attribute:

Age	Job Type	Marital Status	Edu. Lvl	Credit Default	Avg. yearly balance	Has Housing loan	Has personal loan	Contact duration	# previous contacts	Days passed after contact	Outcome of last campaign
60	11	4	4	2	5000000	2	2	7200	10000	365	4

Assuming an agnostic learning setting, to get an error of 0.1 and a certainty goal of 0.1, we require at least 1300 different training examples. While my rough sketch of the hypothesis space is by no means accurate, it does appear that the theoretical lower bound on my learning setting is much lower than the number of training samples in my set. Thus, it makes sense why my learner is not always making progress as the number of training samples increases. Of course, this doesn't answer the question as to why my learner doesn't seem to learn with error rates less than 0.1, but one idea is that the samples in the training set aren't distinct enough, i.e., most of the examples fall within a small portion of the syntactic hypothesis space. One way to improve performance is to better encode string attributes to match more closely with their definitions. For my learning algorithm, I just used "0, 1, 2…" for qualitative attributes. No doubt this method caused my learner to get sidetracked because my method assumes that the values of the attribute are evenly spaced, or that the average of 1 and 3 is 2, etc.

Wine:

Dataset	# of nodes	Train Time	Train Score	Test Time	Test Score
Wine-simple	67	0.8357	0.8030	0.1003	0.8102
Wine-simple (default)	837	0.0266	1.0	0.0658	0.8081
Wine	43	0.7636	0.5420	0.1149	0.5183
Wine (default)	1889	0.0333	1.0	0.0669	0.6061

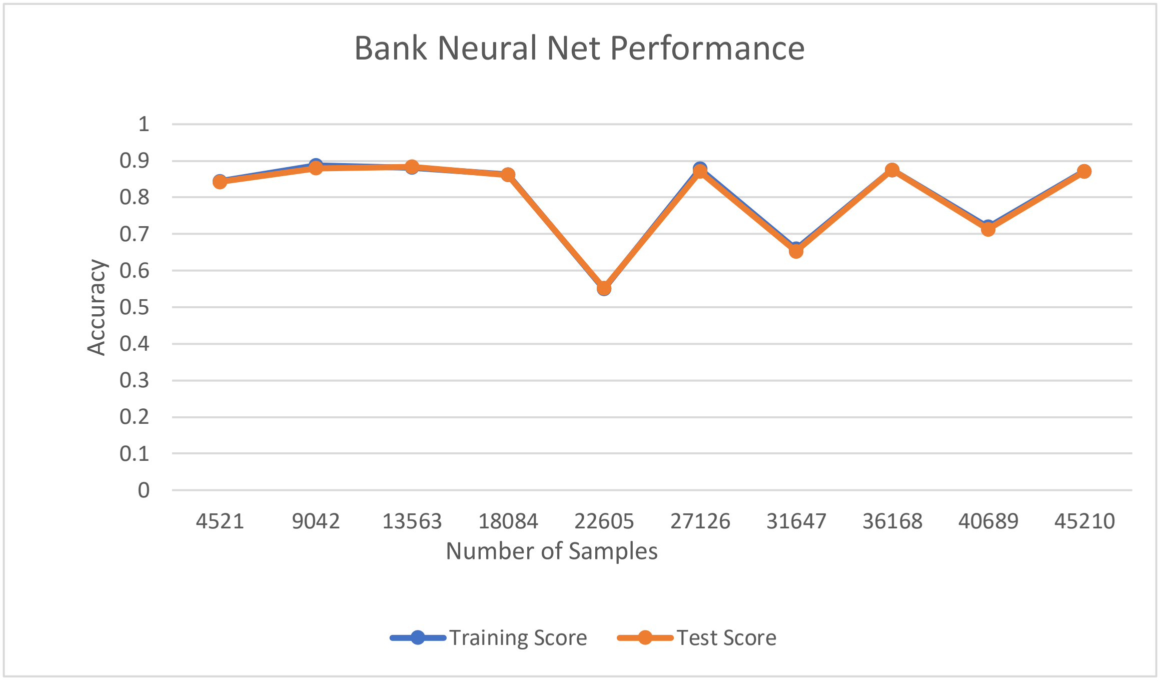


Above are the performance charts for the wine datasets and a graph displaying the training and testing performance as a function of training size. From the table, it's clear that simplifying the problem into 3 classes drastically improved performance. The graph above shows converging lines between test and training scores, which is entirely expected as we go through the pruning process. An interesting note is that the parameter tuned-wine tree did significantly worse than the default wine tree—by almost 10 points. I believe that this is due to the domain of the problem itself; wine is so complex that 43 nodes is not enough to create an accurate classifier. The default tree has almost 2000 nodes, which would allow the tree to consider the subtleties of wine tastes.

<u>Neural Networks</u>

Bank:

Dataset	Train Time	Train Score	Test Time	Test Score
Bank	200.0827	0.8886	0.0751	0.8840
Bank (default)	1.613	0.8238	0.0662	0.8744
Bank-simple	160.6713	0.8837	0.1389	0.8866
Bank-simple (default)	0.5903	0.8436	0.0705	0.8838



Above are the performances of the various neural network train/test scores, as well as a graph representing the accuracy of one of the neural nets as a function of training size. The scores are quite poor. The difference in accuracy between a naïve "no" output and my trained learners are -0.0033, -0.0129, 0.003, and 0.0008 respectively. In addition, the training times for my parameter tuned neural networks were around 3 minutes. The graph above does not show an upward trend as the training examples increase, and there seems to be a lot of noise in the data as my test scores fluctuate a lot. My reasoning behind this is because of randomized search CV, which I used to tune my neural nets. As I mentioned with the decision tree, randomized search CV does not test every combination of parameters, and thus might not have found the best parameters each time. Also, given the data from above, I can conclude that tuning the parameters is not worth it, given how the training time is orders of magnitude longer, with a marginal increase in performance.

Wine:

Dataset	Train Time	Train Score	Test Time	Test Score
Wine	9.5014	0.5055	0.0671	0.5102
Wine (default)	0.4600	0.4454	0.0618	0.4503
Wine-simple	7.8918	0.7683	0.0252	0.7945
Wine-simple (default)	0.2194	0.7802	0.0662	0.7959

Above are the data for the wine dataset. The performance without cross validation tuning is considerably worse compared to the decision trees with regards to the default wine dataset. One reason for why my neural nets are performing worse is because both datasets have a many attributes, some of which contain little to no information. While decision trees are good at splitting nodes based on high information attributes, neural nets can get caught up in the noise of low-information attributes. This idea is backed by the fact that my bank-simple learner performed better than the full bank learner, despite the opposite being true for my decision trees.

Another possible reason for low performance is that my neural nets had three layers. I tried to do a trial in which I increased the number of layers in my neural net:

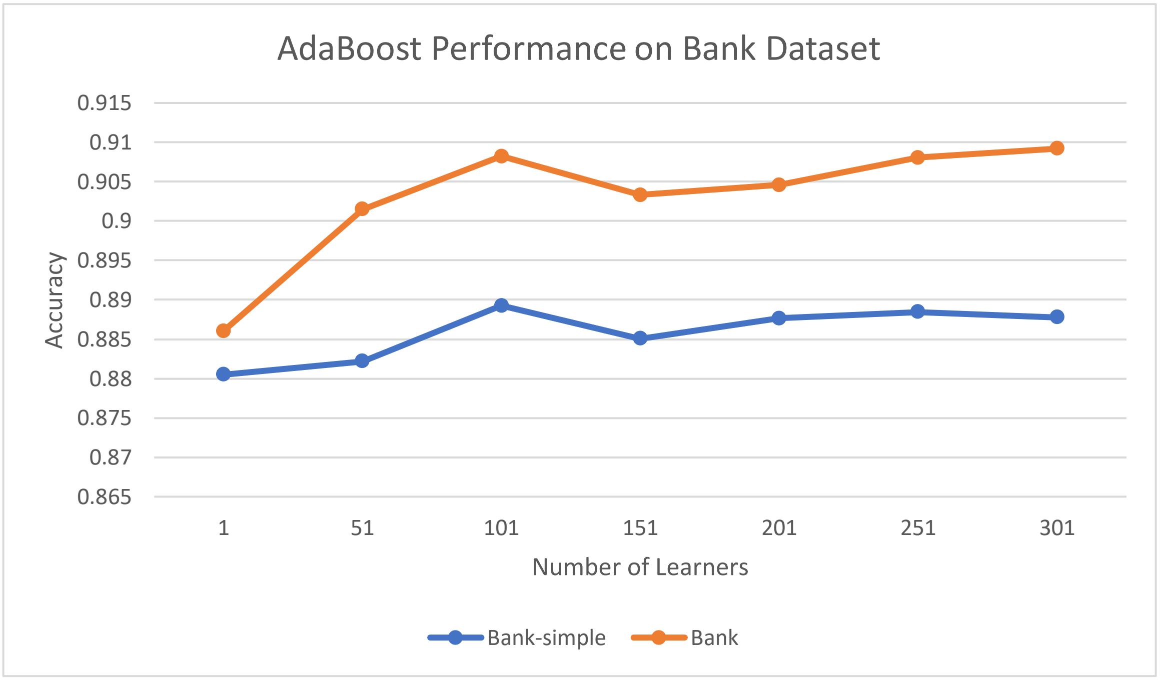
Dataset	Layers	Train Time	Train Score	Test Time	Test Score
Bank (default)	21	39.3244	0.8900	0.5313	0.8858

Not only did the training time increase dramatically, the performance could not classifier better than the naïve "no" output. Maybe if I had more time, I would experiment with different layers, as well as with other parameters such as learning rates, momentum, etc.

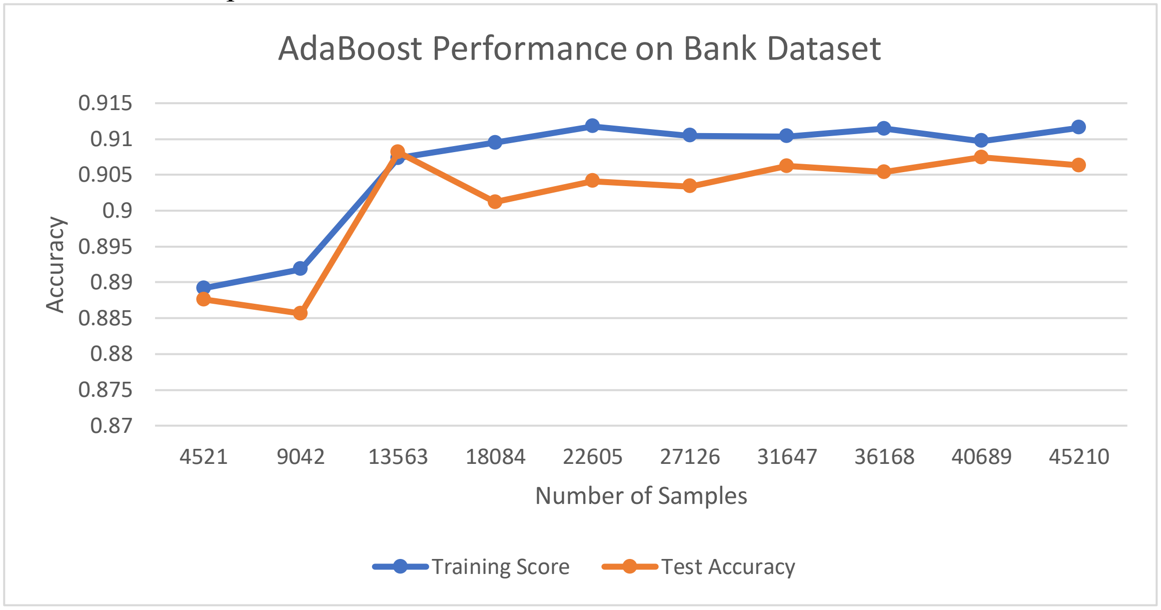
<u>Boosting:</u>

Bank:

Dataset	# Learners	Train Time	Train Score	Test Time	Test Score
Bank	301	7.7482	0.9082	0.3614	0.9032
Bank-simple	301	8.331	0.8913	0.3654	0.8868

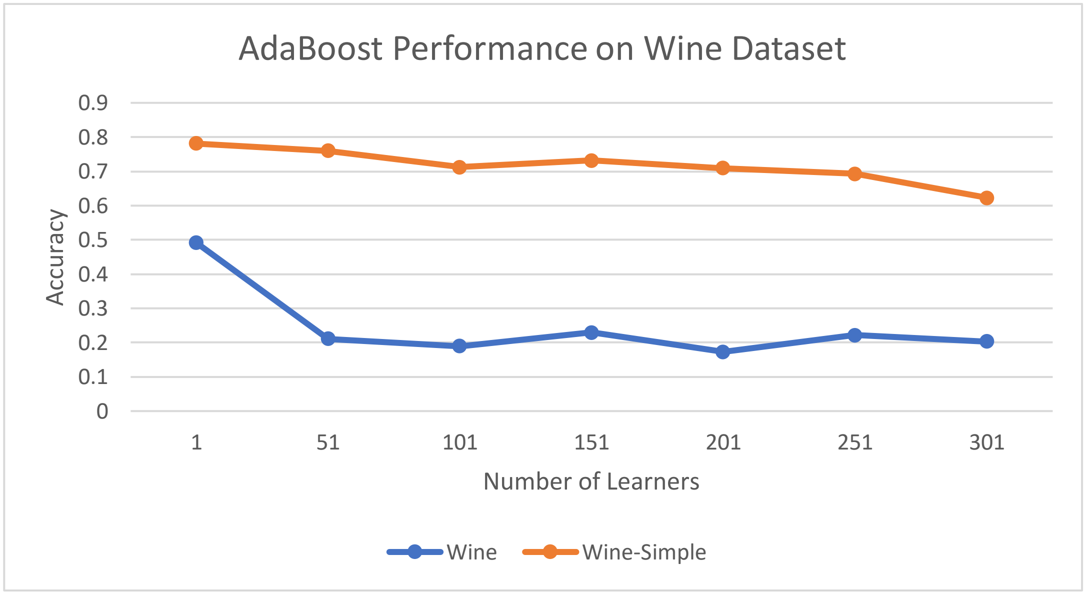


Above are the performance chart and graph for my boosting algorithm. The implementation I used was AdaBoost, with an aggressively pruned base-learner without parameter tuning. From the graph above, we can see that the dataset with full attributes does noticeably better than my simplified version of the dataset. Again, this is probably because I removed attributes that had lots of information. In terms of performance in comparison to the regular decision tree, the boosting algorithm does better, especially with full attributes. This is expected, as the purpose of boosting is to use simple learners to perform better than a single, complex learner of the same type. From the graph above, we can also observe that more leaners leads to generally higher performance. Again, this is entirely expected, as using more learners will help the algorithm focus on the examples it missed before.



Above is the graph of the boosting algorithm's performance as a function of training size. The performance increases generally as the number of samples increases. This is entirely expected, as a simple learner is looking for general rules about a dataset—rules it can get more easily if it uses the entire dataset.

Wine:

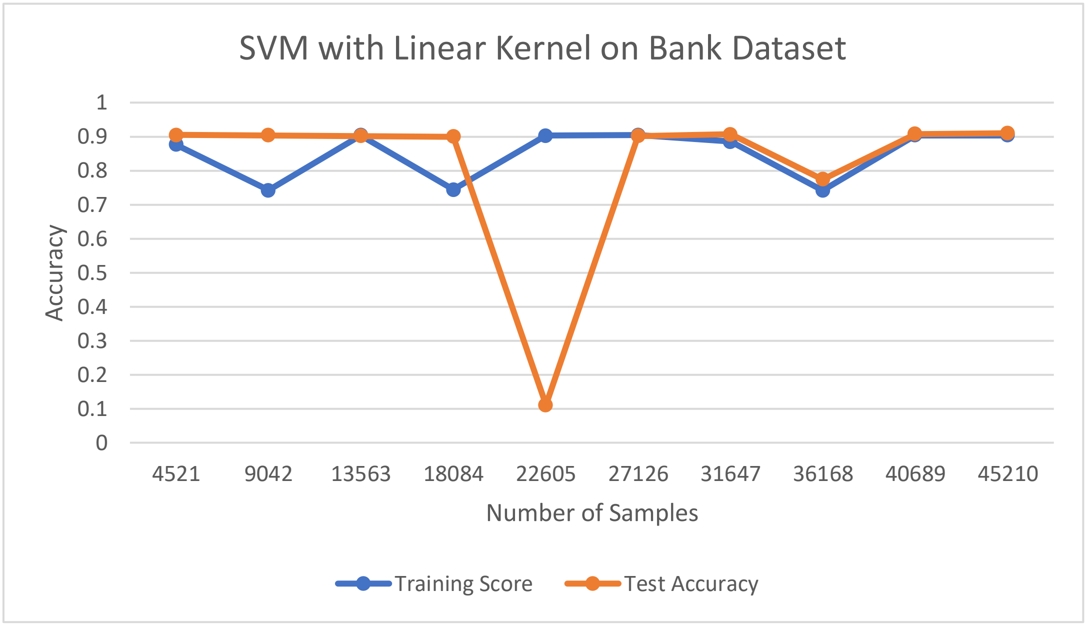


Above is the boosting performance graph as a function of number of learners. Surprisingly, the boosting algorithm not only gets worse as the number of learners increases, but it also performs consistently worse than any other learning algorithm thus far. To address the first point, we must remember that AdaBoost focuses on misclassified samples. The problem with that in the wine problem is that most wines fall within the 5 or 6 class; thus, the new learners were focusing on the details of rare wines instead. To address the second point, we look to the domain of the problem. It's hard to find "general rules" about wine qualities because there are none. Wine tasting is a sophisticated practice that that requires attention to subtle details and depth—two things a basic learner does not have. It's hard to think of ways to improve my boosting implementation with regards to the wine dataset. I could increase the complexity of my learners, but that would defeat the purpose of boosting. Maybe a better approach would be bagging. Using complex learners on subsets of the data would at least allow the learners to capture the subtleties of the data.

In general, boosting with random forests generally takes longer to train, but has better performance than a single, complex decision tree. However, boosting cannot apply to domains in which simple general rules are hard to extract, such as wine tasting.

Support Vector Machines:

Dataset	Kernel	Train Time	Train Score	Test Time	Test Score
Bank	rbf	85.4880	0.8865	8.7124	0.8888
Bank	linear	4.6275	0.9072	0.0265	0.9117
Bank-simple	rbf	58.2230	0.8823	6.0939	0.8844
Bank-simple	linear	3.5320	0.7514	0.0203	0.7842



Thus far, the support vector machine with a linear kernel has the best performance out of any algorithm (above 91 points). In addition, the training time and testing times are fast compared to other approaches. The good performance means that the bank data is structured in such a way that it can be linearly separated. I can reasonably infer that the attributes I removed in bank-simple had important information because the linear SVM performed extremely poorly. With regards to the outlier value at 22605 training size, I have no idea how that happened. It almost seems as if the SVM did the correct slice, but swapped the "yes" and "no" regions.

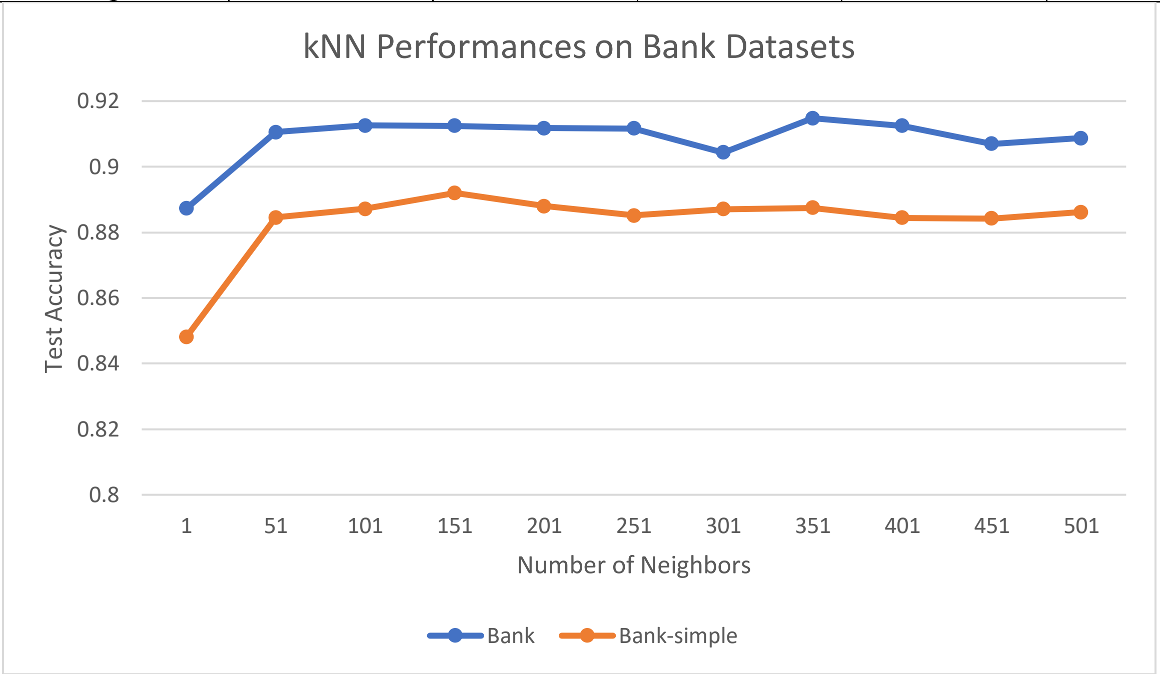
The radial basis function kernel did consistently average, staying slightly ahead of the naïve "no" output. Some ways to improve the SVM approach is to play around with the parameters, or to explore other kernel functions. The reason I did not do this is because of time constraints. For example, I tried to do tests with a polynomial kernel but had to kill the program after an hour of training. Maybe with some domain knowledge it is possible to create a custom kernel that will prioritize the most important attributes; unfortunately, I do not possess such knowledge.
Wine:

Dataset	Kernel	Train Time	Train Score	Test Time	Test Score
Wine	rbf	0.9319	0.5388	0.1551	0.5544
Wine	linear	1.014	0.3081	0.0017	0.3544
Wine-simple	rbf	0.5750	0.8120	0.1117	0.8115
Wine-simple	linear	0.6131	0.7374	0.0017	0.8006

Above are the charts for the SVM implementations on the wine datasets. Other than having fast training/testing times, there is nothing particularly impressive with regards to performance. However, unlike the bank dataset, the linear kernel SVMs seem to have trouble with the wine dataset, especially with the not simplified dataset. This performance trouble could be linked with why the boosting algorithm was performing poorly—wine tasting too complicated to be modeled with a linear function. This idea is supported by the fact that the radial basis function kernel SVM did 20 points better than when using the linear kernel.

K Nearest Neighbors:

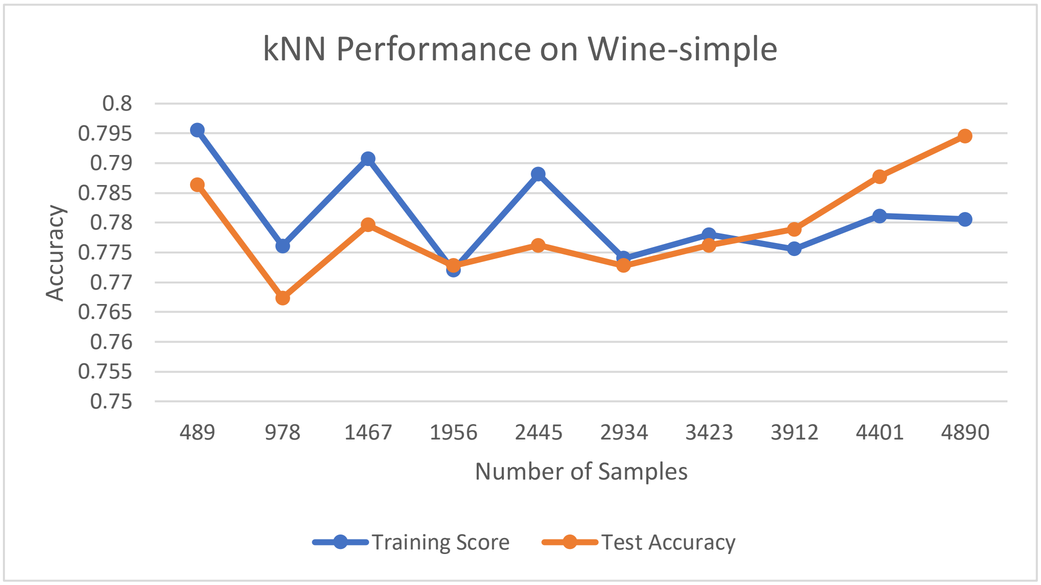
Dataset	Neighbors	Train Time	Train Score	Test Time	Test Score
Bank	500	0.3176	0.9098	4.3687	0.9097
Bank-simple	500	0.0889	0.8865	0.1696	0.8848



From the data above we can see how the kNN algorithm gives us a pretty fast training and decent testing time, as well as a very high score for the bank dataset. It is not surprising that the testing time for kNN is much longer than the training time, given how the kNN algorithm is supposed to train almost instantly. A possible reason for the good performance is that all the attributes in the problem are important. Since the kNN algorithm has no information loss, it is taking every attribute into account, whereas some algorithms ignore what it perceives to be irrelevant attributes. Again, from the graph, we can see that bank-simple is performing consistently worse. Since all the attributes in the problem are important, losing some would reasonably cause kNN to perform worse. Something else to note is that the number if neighbors doesn't seem to have a large impact on performance. My guess as to why is because the data is so dense (over 40,000 samples) that there are almost always going to be enough neighbors in the vicinity, whether it be the closest 50 or closest 500, such that the output will not be affected by irrelevant neighbors.

Wine:

Dataset	Neighbors	Train Time	Train Score	Test Time	Test Score
Wine	150	0.0058	0.4536	0.0538	0.4598
Wine-simple	150	0.0053	0.7805	0.0495	0.7945



Above we have the chart for the kNN performances on the wine datasets and the graph representing kNN performance as a function of training size. kNN does not seem to give any impressive results in either dataset; however, the training and testing times are fast. By looking at the graph, we notice that testing accuracy increases with the number of samples used for training. This is entirely expected, as similar wines often receive similar scores.

Conclusion:

It's hard to say which algorithm was the "best," but from a test score standpoint, SVMs generally preformed the best on the banking dataset, and the decision tree did the best on the wine set. However, the training times for some datasets took over a minute, whereas kNN took about 5 seconds to produce a score about 0.01 score lower. Of course, other algorithms may not have performed as well because of my own lack of experimentation and optimization. I think that each algorithm had its own drawbacks and strengths, and that the "best" approach depends on what kind of standards we have for performance, train/test times, and versatility.