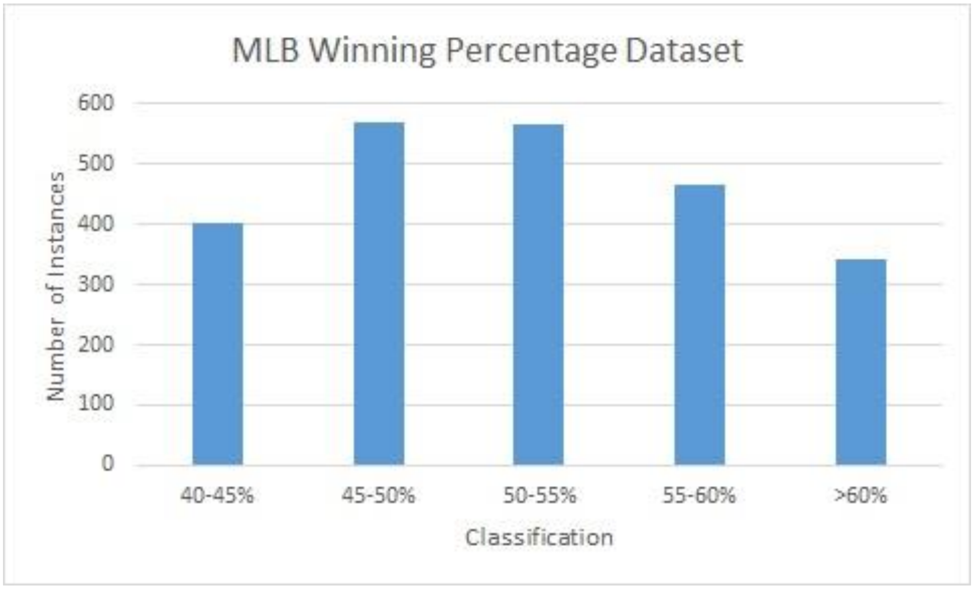
Ethan Shernan

CS 4641

Supervised Learning Analysis

Problem Descriptions

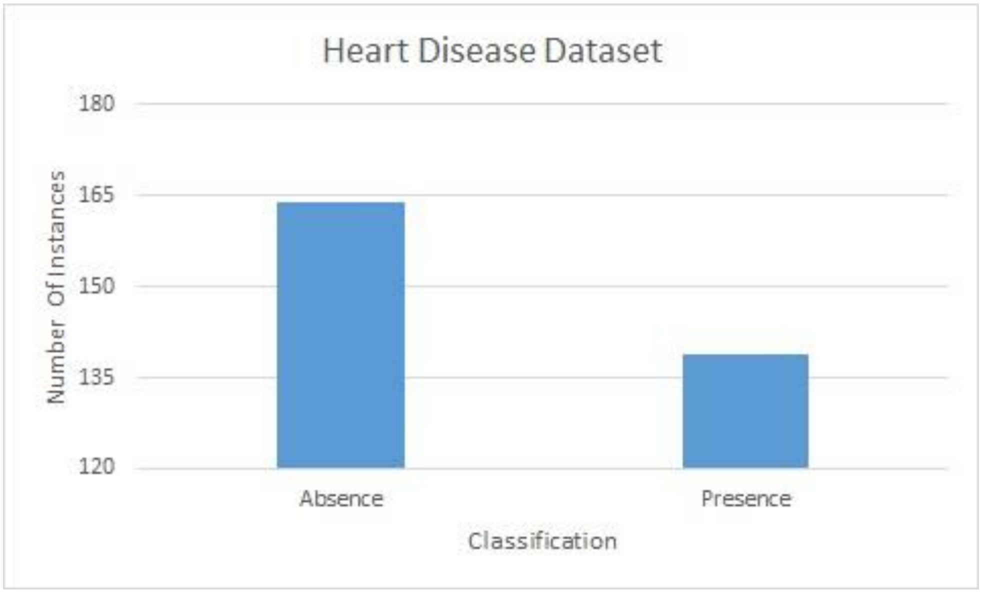
MLB Team Winning Percentage



For a different style of dataset, I looked to a database that I personally owned and certainly would enjoy studying. Awhile back I had set up a database of team and player statistics for Major League Baseball dating all the way back to when statistics first began being collected. After testing various ways of using this database to produce interesting results on Machine Learning algorithms, I settled on classifying team winning percentages as an interesting method of observing supervised learning.

This dataset contains five classes representing ranges of winning percentages (I.E. 40-45%, 45-50% etc…). The attributes for the data were various statistics collected from each team in an attempt to capture the various different aspects of a baseball team. Counting statistics like total runs allowed, runs scored, home runs etc… were supplemented by rate statistics like batting average, earned run average, and slugging percentage. Overall, there were 9 attributes and 5 classes spanning over 2349 instances. My idea behind this dataset was to look at the effects of multi-class data and their effects on the algorithms. I was also curious as to how normal distribution data would work, as the nature of the sport and the statistics I collected resulted in a nice, fairly clean, normal distribution. What ended up happening was a wide range of success (and failure) across all of the algorithms. Prediction rates ranging from 64% to 51% - from acceptable to coin flip - made this dataset particularly intriguing to study.

Heart Disease Classification



The Heart Disease dataset provided various attributes of collected patient data related to the possibility of the patient having heart disease. In total, there are 14 attributes ranging from the obvious of age and sex, to more complicated measurements like cholesterol levels, reactions to exercise, and blood sugar levels. In the original dataset, there were four classes representing a severity of heart disease from 0-4; 0 representing no presence, and 1-4 representing some level of heart disease. I preprocessed this data into two classes: simply 0 and 1, representing absence and presence.

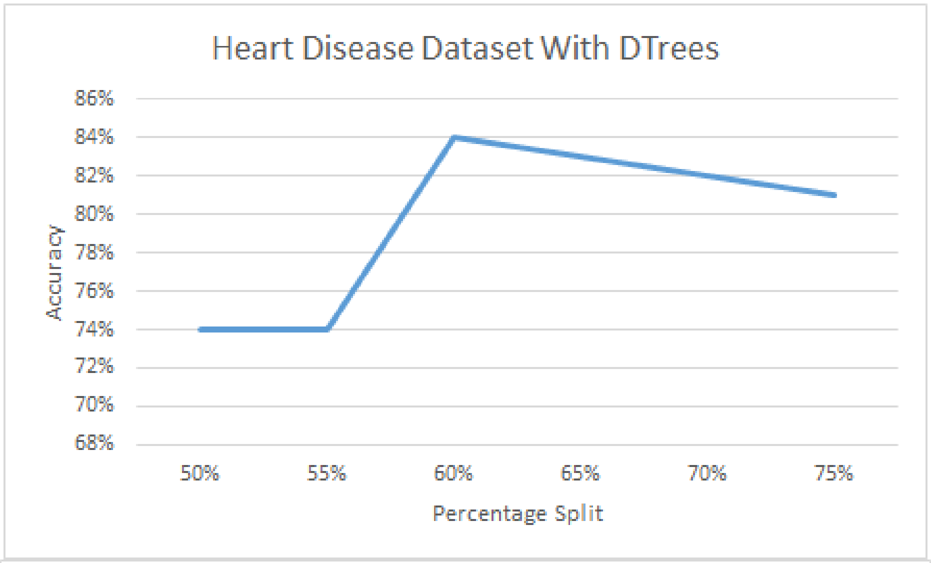
From a practical standpoint, this dataset has incredible applications in the medical industry. By taking a subset of symptoms and feeding it through a reliable model, it could aid in diagnosis for future patients. From a machine learning perspective, this dataset was interesting to me because there were a significant amount of attributes (14) and few instances (303). I figured this would be the perfect example of "The Curse of Dimensionality", and that I wouldn't be able to find good results with any of the algorithms until I pruned the attributes that didn't aid the building of the model. I was able to achieve prediction rates from 88% to 78%, all within a - somewhat - acceptable range, but still revealing interesting facts about the various algorithms while not being trivial.

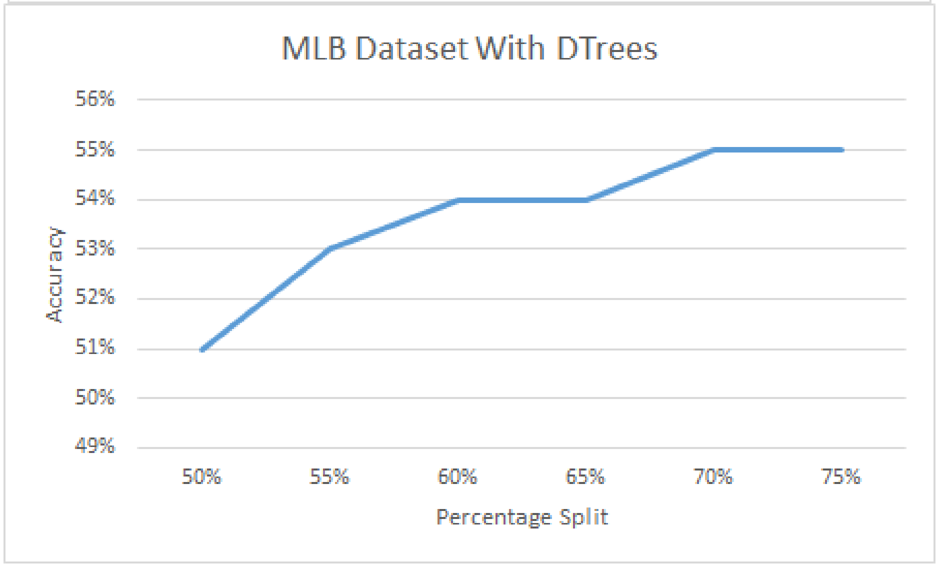
Methodology Notes

Each dataset was automatically split into various percentages of training and testing data using Weka's "Percentage Split" tool. This way, I could use similar data for both testing and training while being able to easily change the amount of data used for each. For certain experiments I also performed a Cross Validation technique on the entire dataset and compared the results to the percentage split method.

I categorize "accuracy" as the percentage of correct predictions made on the test set given the trained model. Because neither of these datasets took particularly long to train with these algorithms (and my newly built computer :)), I am focusing more on accuracy rather than time. Unless otherwise stated, the training set was 66% of the overall data and the testing set was 33%.

Decision Trees





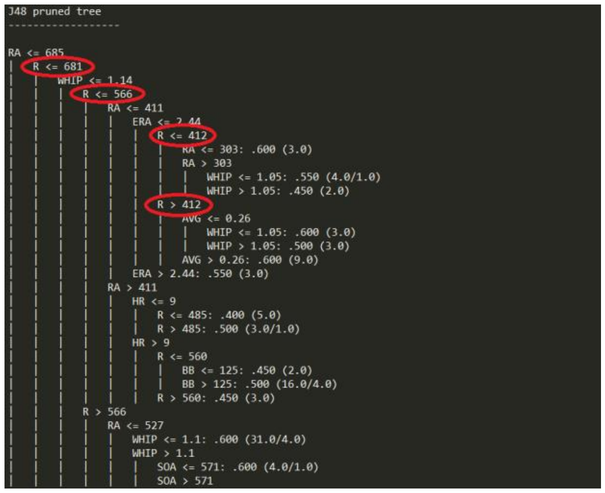
Accuracy vs. Training/Testing Data Split

Decision trees overall performed very poorly on both datasets, regardless of pruning. For the heart disease dataset, it was by far the worst algorithm performing over 7% (unpruned) worse than the average percentage correct across all of the algorithms. It also was almost 11% worse than the best performing algorithm on this dataset. For the MLB data, the unpruned tree actually seemed to fare a little better, relative to the other algorithms. While it had a poor 53% prediction rate, it was only 2% off from the average performance, and did perform better than Boosting and KNN. In both cases pruning resulted in a very slight increase in performance (1%) and increased modelling and testing times on the MLB dataset.

So why was the Decision Tree the worst algorithm for Heart Disease Classification? The reason why this particular dataset was so poor with decision trees most likely had to do with how many discrete values each attribute could possibly take on. We see in the unpruned tree that the “cho” attribute (cholesterol level) is the last attribute before the leaves of the tree to make a decision on, and it takes 152 *distinct* values. The inductive bias of a decision tree is to keep high information gain attributes towards the top of the tree, and thus, an attribute with so many values that's at the bottom of the tree doesn't give us much information. In fact, in the pruned tree, "cho" isn't even included and we lose any information that attribute may have given us.

This phenomenon can also cause a problem with overfitting. With 152 distinct values spread across 303 instances, the algorithm only knows what approximately two instances look like for any of these distinct values. If both of those instances are classified the same (despite what the other attributes say), then further classifications may fall into the trap of being overfit into whatever those two instances resulted in. While this isn't exactly a demonstration of "The Curse of Dimensionality" it is certainly a result of not having enough data to cover *one* of the dimensions of the problem.

This effect is actually amplified in the MLB dataset. The nature of all statistics in baseball are to be numerical counts of some sort of value, naturally giving us many discrete values for our attributes. SOA (strikeouts by pitchers) with a whopping 954 distinct values sat at the bottom of any of the branches it was included in the tree. However, we notice something else interesting about this dataset.



```
J48 pruned tree
------------------

RA <= 685
|   R <= 681
|   |   WHIP <= 1.14
|   |   |   R <= 566
|   |   |   |   RA <= 411
|   |   |   |   |   ERA <= 2.44
|   |   |   |   |   |   R <= 412
|   |   |   |   |   |   |   RA <= 303: .600 (3.0)
|   |   |   |   |   |   |   RA > 303
|   |   |   |   |   |   |   |   WHIP <= 1.05: .550 (4.0/1.0)
|   |   |   |   |   |   |   |   WHIP > 1.05: .450 (2.0)
|   |   |   |   |   |   R > 412
|   |   |   |   |   |   |   AVG <= 0.26
|   |   |   |   |   |   |   |   WHIP <= 1.05: .600 (3.0)
|   |   |   |   |   |   |   |   WHIP > 1.05: .500 (3.0)
|   |   |   |   |   |   |   AVG > 0.26: .600 (9.0)
|   |   |   |   |   ERA > 2.44: .550 (3.0)
|   |   |   |   RA > 411
|   |   |   |   |   HR <= 9
|   |   |   |   |   |   R <= 485: .400 (5.0)
|   |   |   |   |   |   R > 485: .500 (3.0/1.0)
|   |   |   |   |   HR > 9
|   |   |   |   |   |   R <= 560
|   |   |   |   |   |   |   BB <= 125: .450 (2.0)
|   |   |   |   |   |   |   BB > 125: .500 (16.0/4.0)
|   |   |   |   |   |   R > 560: .450 (3.0)
|   |   |   R > 566
|   |   |   |   RA <= 527
|   |   |   |   |   WHIP <= 1.1: .600 (31.0/4.0)
|   |   |   |   |   WHIP > 1.1
|   |   |   |   |   |   SOA <= 571: .600 (4.0/1.0)
|   |   |   |   |   |   SOA > 571
```

MLB Pruned tree with repeated attributes

Despite the fact that there were fewer attributes than the Heart Disease Dataset, the tree was significantly larger. Of course, with more data, there are more distinct values for each attribute. However, the other interesting factor is the 5 classes that the MLB set has. While none of these sets overlap, they do lie very closely to each other in terms of what the data means. The difference between a team finishing with a winning percentage of 49% and 50% is a little more than one win. While every algorithm has to make distinct decisions between classes, decision trees in particular have to do their best to separate classes at each level of the tree in accordance with its inductive bias. If separating these proves to be difficult, then more levels of the tree will be needed. Noticing that the same attributes would come up repeatedly along the same branch ("R" in MLB pruned for example) means that each individual attribute does not provide us with all of the information that we need to make a decision. This is why decision trees worked particularly poor with the MLB dataset.

While cross-validation did not improve the performance of Decision Trees, there are some things I can change in the data to make it perform better. In the MLB set, I could remove some of the "counting" statistics (total number of strikeouts) and instead use "rate" statistics (strikeouts per game). This would provide a smaller range for the numbers to lie within and would greatly reduce the number of distinct values each attribute could be. Pruning should have removed some of the attributes that didn't provide significant information gain, but I could also do this manually for a similar effect.

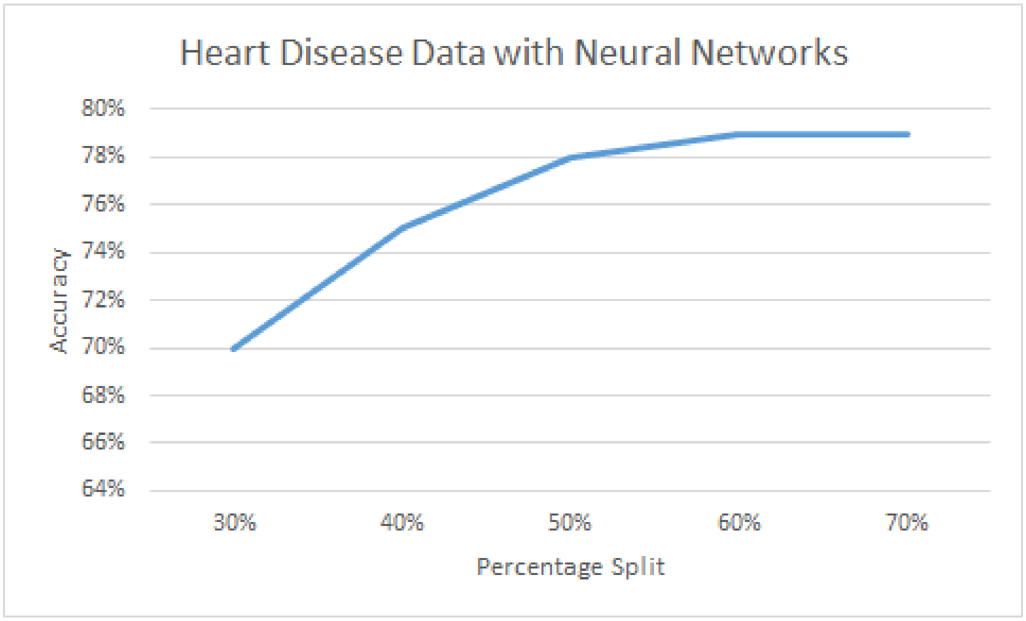
Neural Networks

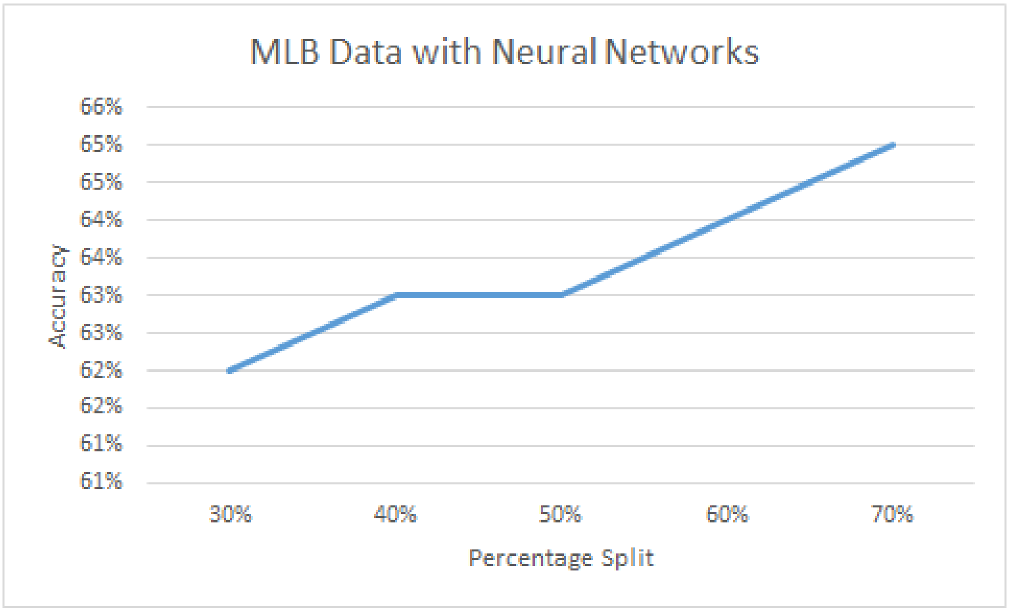
Neural networks provided one of the most interesting comparisons between the two datasets. It was *by far* the best algorithm for the MLB dataset reaching the peak prediction rate of 64%. This was almost 10% better than the average and 5% better than the next best algorithm. However, for the Heart dataset, this algorithm was only slightly better than the decision tree at 81%, and took longer to produce than the equally performing Boosting technique. This significant discrepancy really starts to reveal interesting things about neural networks and how my data really looks under the machine learning lens.

These results lead me to believe that Neural Networks are much more sensitive to how much data you have relative to the other algorithms. Just from looking at the running times you can tell Neural Networks need to do more processing in order to create a working model which seems to lend itself to my theory.

This makes sense as well when we consider the structure of a multilayered perceptron network. For each node that we are attempting to distribute the training error over (as in back-propagation), we have weights that connect the layers together. Each node has one weight for each attribute, and as data gets more and more difficult to classify, more and more layers are added to the network. This means that complexity grows exponentially as more information is provided, leading to a very data hungry algorithm.

Changing the percentage split of how much data is used for training further backs this up. Reducing the training percentage to 10% for the MLB set cuts running time in half, but also drops accuracy 6%. Similarly, increasing the training percentage to 77% for the Heart Disease set increases accuracy by 3%.





This is all a good example of "The Curse of Dimensionality", much like the decision trees were. Without enough data to cover all of the attributes used by the examples, the Heart Disease dataset performed much more poorly than the MLB dataset. Thinking back to how neural networks are very data hungry, I can see this being amplified for this dataset.

The best way to increase the effectiveness of Neural Nets would have simply been more data. Interestingly enough, cross-validation actually made the heart data *less* accurate, which further leads me to believe that more coverage amongst the various dimensions of data needed to be obtained. It should be noted that Neural Networks also took the longest to build for both datasets. While not costly on its own, this fact combined with the lack of accuracy made it one of the worst algorithms for the Heart Disease dataset.

Boosting

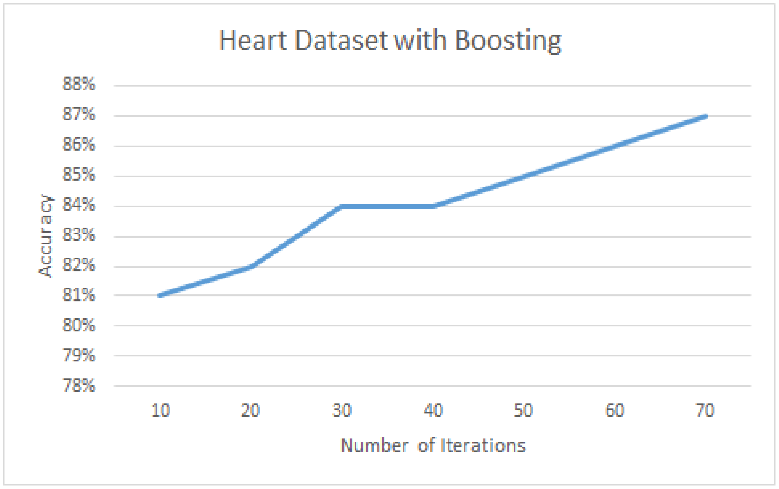
Boosting provided another very interesting result on my MLB dataset. With an accuracy of 52%, it actually performed *worse* than the standard decision tree model. It also took almost 10 times longer to train than the Decision Tree model, and was only very slightly better than KNN in terms of pure accuracy. The final tree it generated was of similar size to the unpruned tree of the standard Decision Tree algorithm.

As for the heart disease dataset, boosting sat around the middle of the pack amongst the algorithms. It did equally as well as the neural networks at 81%, but only took half the time to build the model, putting it slightly ahead. It should be noted that while the neural networks tended to make more false positive classifications, boosting was about even on false positives and false negatives.

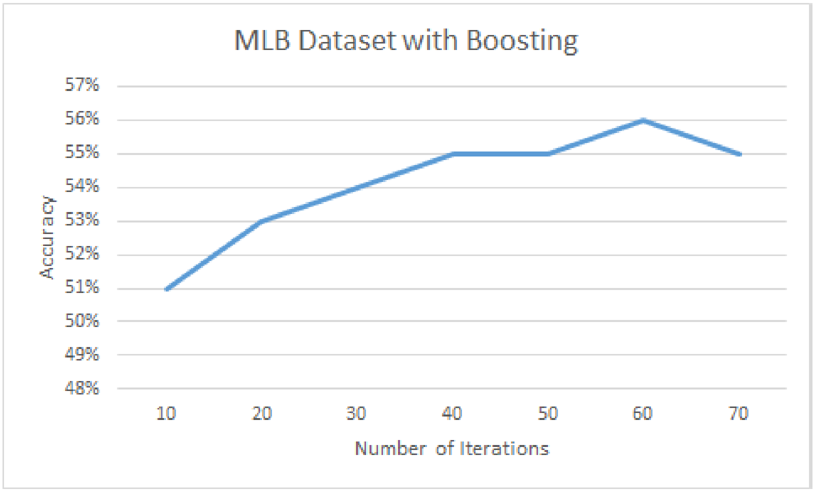
Since, at its heart, boosting used decision trees to make its predictions, many of the flaws found with decision trees applied directly to the results found with boosting. Boosting attempts to make a generalized model by creating small models from different chunks of the dataset, then combining them together. However, as we saw in the decision tree analysis, those models may not necessarily be the best ones to go on. While we can assume at least some learning is performed by each small model (as per the definition of a weak learner), the MLB results show us that the total model is more than the sum of its parts.

However, all was not lost with this algorithm. Boosting was more sensitive to how long I actually had it run when compared to the other algorithms. This most likely had to do with the fact that the underlying weak learner was flawed to begin with. By increasing the number of iterations, boosting was able to do better on the MLB set than decision trees. On the heart set, we see an even more exaggerated increase in performance that puts it up with the best algorithms for that dataset.

The differentiation between the performances on these two datasets leads me to a few conclusions about boosting. First of all, while boosting seems to fall to some flaws of the underlying weak learner, they can be overcome somewhat by the simple brute force approach of giving the algorithm more time to work. Second, boosting seems to be more sensitive to some underlying flaws in the data than others. For example, due to the much improved performance on the heart dataset as opposed to the MLB one, I believe that the very closely situated classifications may have been a major problem for the boosting algorithm. Remember, the difference between a 49% team and a 50% team is incredibly small, and both decision trees and their boosted counterparts really seemed to struggle with this.

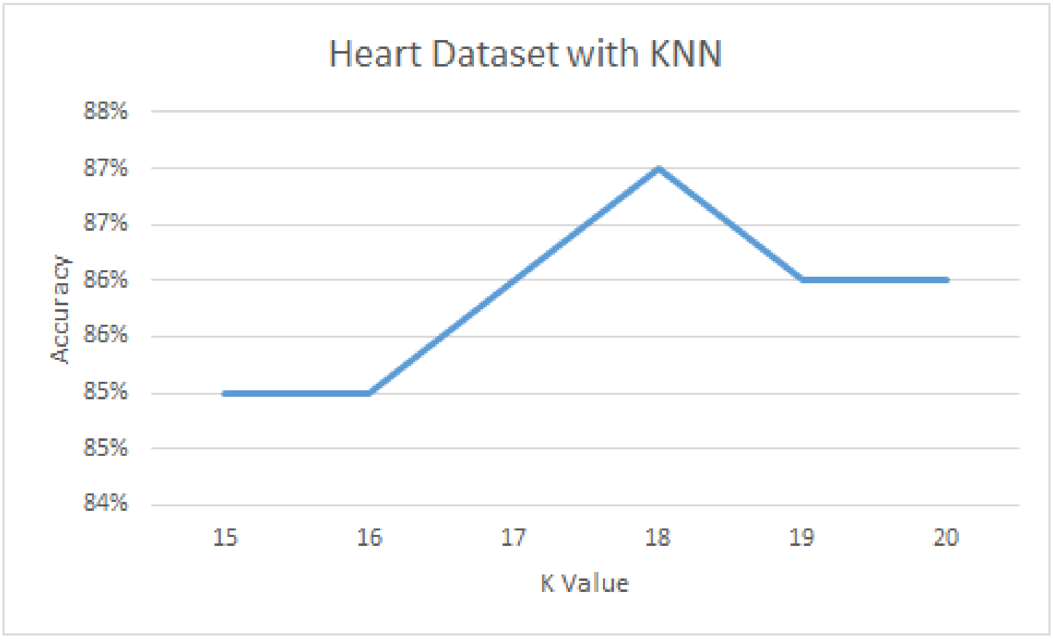


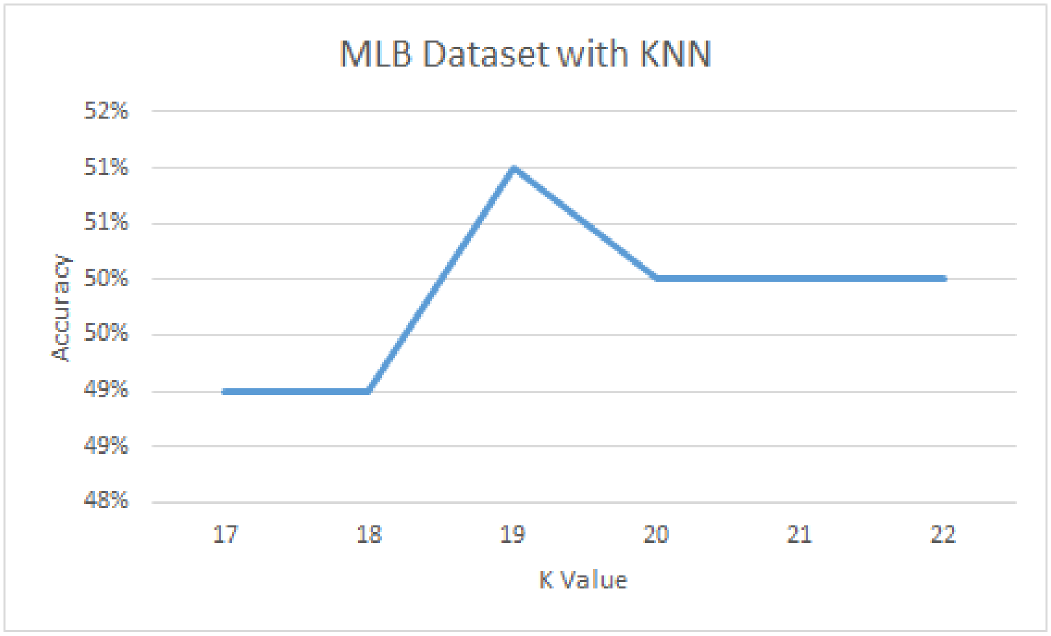
Boosting in particular had an interesting reason to struggle with this. Deciding between classes at a very close border is essentially a coin flip in most situations. Boosting decides to take on those instances *first* because they are more difficult to solve and are more pertinent to be trained. This means that boosting could spend an awful lot of time on those instances, and if not allowed enough time, it will perform very poorly on the entire dataset. We see this illustrated in the graphs. The MLB set needed a lot of iterations to get past the borderline cases of classification whereas the heart dataset didn't have the same issues.



KNN

KNN provided the most interesting result by far of all of the algorithms. When choosing K, I attempted to find a local maxima by simply seeing where accuracy peaked on both of the datasets. As it turned out, K peaked at 19 for the MLB set, and 18 for the heart disease set. I would have actually expected the numbers to be a bit farther apart considering how much more data was in one set compared to another. However, when we consider there are fewer attributes for the MLB set, and thus, fewer similarity measurements, it makes sense that K was lower in the heart disease set.





For the MLB set, it was the worst performing algorithm. At 51% accuracy, it was 4% worse than average and over 13% worse than the multilayered perceptron model. It even performed worse than the maligned decision trees and AdaBoost. Contrasting this directly, KNN was the second *best* algorithm behind support vector machines on the heart dataset. It was able to achieve accuracy of 86%, greatly above average and just short of SVMs.

The inductive bias of KNN is that "Similar Examples are Similar", and thus should be close to each other given the various attributes. This works well under a few assumptions. One, all of the attributes used to classify an example contribute equally to the overall classification. Two, the attributes are actually enough to classify a new example when given. Clearly, these results show us that the MLB set did not follow these assumptions.

Like in the boosting scenario, we see that the MLB set isn't perfect. An MLB team might be really good at pitching and achieve many wins through that, or maybe they do it with hitting, or maybe with defense. Each of these will be represented with different attributes, and each could result in the same classification. For a concrete example, take the 2013 Tampa Bay Rays and the 2013 Boston Red Sox. In terms of attributes, both teams would be classified the same (55%-60% winning percentage). However, the Rays had 150 fewer runs and were 20 points worse in Batting Average and Slugging Percentage[1]. Of course, their pitching statistics are higher than Boston's. These two data points would be classified the same, but in terms of distance, would be very far apart. Since Euclidean distance is a very common similarity measurement to use in KNN (and the one I used), these similarly classified instances are not actually considered similar by the algorithm.

KNN does terribly with this type of data. The statistics I selected to be used to classify winning percentage don't completely tell the story of how a team achieves their winning percentage. While some algorithms are able to make up for this, KNN treats all attributes the same. The heart disease set plays much nicer as there are some uniform symptoms captured in the data that cause heart disease.

The only way to allow the MLB set to work with KNN is to restructure the question or remove attributes to make it play nicer with the algorithm. Maybe we could only look at teams that were good at hitting, and then try and classify them based off of those statistics. Without a common method of "winning", it becomes very difficult for KNN to be a good classifier with this data.

SVM

SVMs were the most consistently high performing algorithms amongst the ones we tested. It was the best performing algorithm on the heart data, with an 88% accuracy rate, well above average, and 2% better than KNN. The MLB set also reacted well to SVM, producing an accuracy of 59%, only behind neural networks.

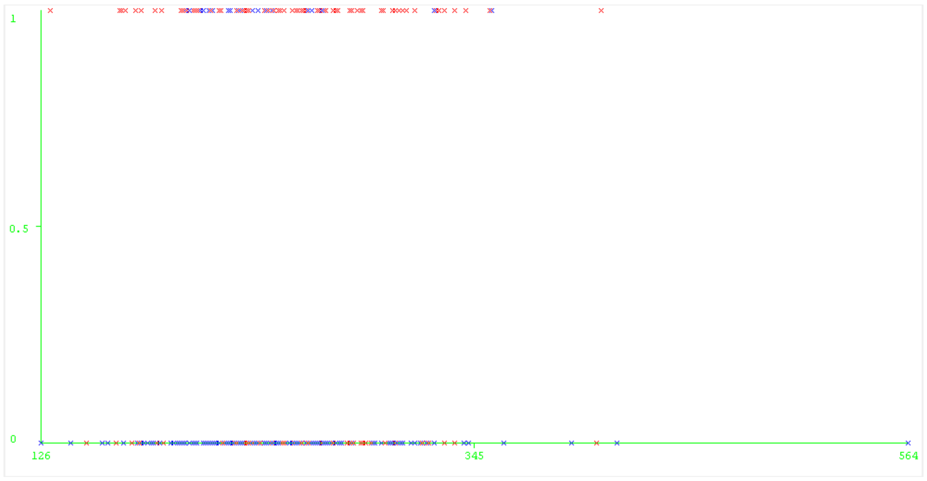
When trying to decide why SVMs were so effective, I thought about how they worked and how my data fit into their model. SVMs attempt to find boundaries along the lines of how the data is classified, and then maximizing the distance between these lines (the margin). If data is well behaved and boundaries can be easily found, then SVMs will perform very well.

My first task was then to try and discover if the boundaries for my data were fairly well defined. Since I can't really graph things in 9 or 14 dimensions, I picked two of what I considered to be the most important attributes and graphed them along with their classification.

In the heart disease dataset, plotting cholesterol against exercise induced angina (heart pain) produced a very clean split of the data. Most positively classified instances (red) had heart pain, and most negatively classified instances (blue) did not. If we think about this from a common sense

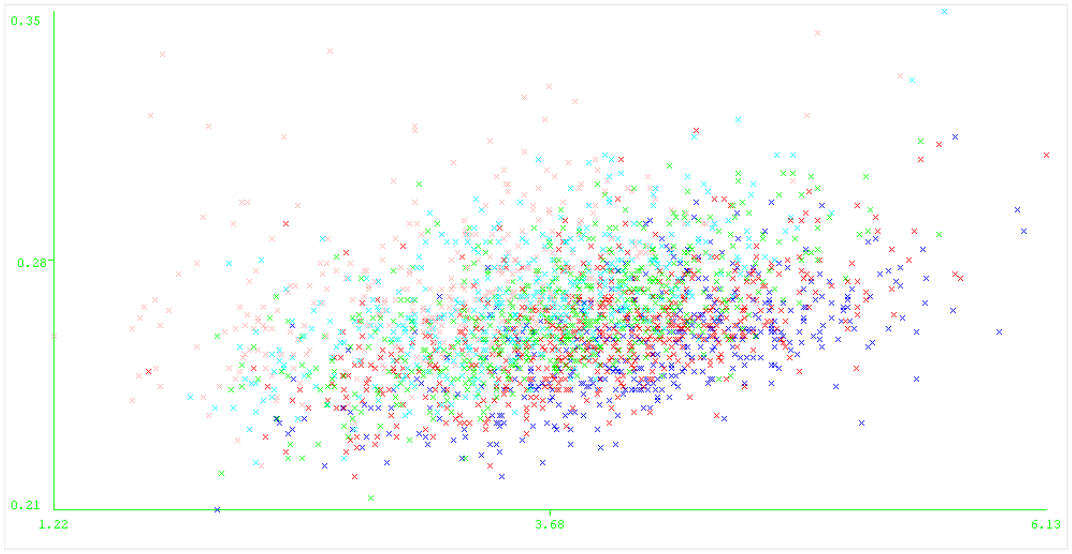
[1] http://espn.go.com/mlb/team/_/name/bos/boston-red-sox
http://espn.go.com/mlb/team/_/name/tb/tampa-bay-rays

standpoint, we know certain symptoms that are very likely to cause heart disease. Those symptoms will generally do a good job of separating the data.



Cholesterol level vs. Exercised Induced Angina with Classifications

The MLB dataset is even more interesting to look at with multi-class data. When we plot AVG against ERA (a crucial statistic in batting and pitching respectively) we see another reasonable split of the data. It's a bit harder to see, but the overall gradient of the classifications show up clear areas where some classes are but not others. This again makes sense when we consider the type of statistics I gathered (and the amount of it) lead to a nice, normal distribution of data as well as a normal distribution of classifications. This distribution naturally tended to a nice gradient of classes like we see in this graph. The best classifications (pink, turquoise) tend to fall nicely with low ERA (X-Axis, lower is better) and high batting AVG (Y-Axis, higher is better). The converse is also true with worse classifications (dark blue and red) having worse statistics.



Batting AVG vs. ERA with Classification

SVMs only tend to worry about data near the borders and discard noise away from them. In looking at these graphs we can see that the noise is very minimal and will have minimal impact on the support vectors.

Choice of kernel made an impact as well, with the PolyKernel doing significantly better than the NormalizedPolyKernel in both time and accuracy.

The fact that my data was so well behaved made it a good candidate for strong SVM classification. Clear borders between classes and lack of significant outside noise provided concrete proof to the common sense arguments we could make about the data.