Assignment 1

Gary Saavedra

gsaavedra3@gatech.edu

Introduction

Choice of Algorithms

J48 was used for the decision tree. Ths was chosen because J48 is closely modeled after the ID3 algorithm presented in class. The J48 uses a variant of rule post-pruning. The MultiLayer Perceptron which uses gradient descent for weight adjustment was used for the Neural Network. SMO was used for the support vector machine. The choice of kernels for the SMO algorithm was PolyKernel and RBFKernel. IBk was used for the nearest neighbors algorithm. AdaBoostM1 was used for the boosting algorithm with J48 as its weak learner. These algorithms were implemented using Weka and the Weka Java API.

**Choice of error metric**

For the learning curves the error metric is both the false positive rate and the false negative rate for each algorithm. This is to make sure no information is missed. If only the percent correctly calculated was used we may experience an inaccurate view of our algorithm’s performance. Using both the false positive and false negative rate shows us which examples our algorithm tends to misclassify. In addition using a simple accuracy measurement such as the overall number of correctly classified instances can be misleading in cases where our class is unbalanced. For example, if the true classification of our instances is largely skewed towards a positive classification type then incorrectly classifying a large portion of the negative class will not have a large effect on our total number of misclassified instances. One data set used here, the Tic Tac Toe set, is an unbalanced class with about twice as many classifications for the positive examples.

**Understanding the curves**

The learning and complexity curves are located along side the analysis. For the learning curves the y axis is the error metric and the x axis is the percentage of the training data used to train the model. Solid lines indicate the false positive rate while dashed lines indicate the false negative rate. For the learning curves the data sets were broken up into 60% training and 40% testing. For the complexity models cross validation is used instead of a testing set to reduce variance due to noise.

Chess Data Set

Why is it Interesting?

The first data set used here is a Chess data set with 3196 instances and 36 attributes. Each instance represents a board configuration in the end stage of a chess game. In this data set the attributes are made up of discrete values. It is a binary classification set with each instance being classified as “win” or nowin”. The data set is fairly balanced with 52% “win” classifications and 48% “nowin” classifications. This data set illuminates some differences in the operating principles of the different algorithms. Using this set we can see how the underlying biases of nearest neighbors and support vector machines affect their ability to classify instances. It also allows for an investigation for why decision trees and neural networks perform well on this type on data set. Analysis of this data can be used to improve a player’s performance during the end stages of a chess match. It will allow a player to determine which moves truly matter when attempting to capture an opponent’s king. For this data set a false postive is characterized as a “nowin” being classified as a “win”. A false negative is a “win’ being classified as a “nowin”.

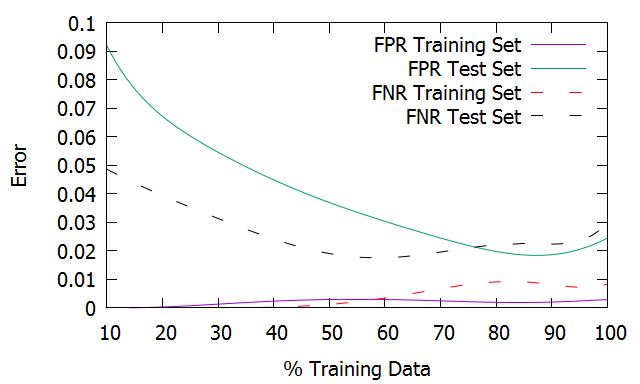
**Artificial Neural Network**

The learning curve for the mutilayer perceptron is depicted in Figure 1. This curve was generated using 100 epochs and 2 hidden layers. This algorithm is one of the best performers out of the 5. Looking at the false positive rate and false negative rate curves we can see at around 90% the model starts to overfit. This is indicated by the the trajectory of the training and testing curves. The training error continues to fall while the testing error begins to rise around 85%. We can see the algorithm needs a significant chuck of the data before it starts performing well. With only 10% of the training data we get a testing error of about 9%. After about 50% of the training data, however, the false positive rate drops below 5%. We can also see from figure 1 that the false positive rate is larger than the false negative rate.

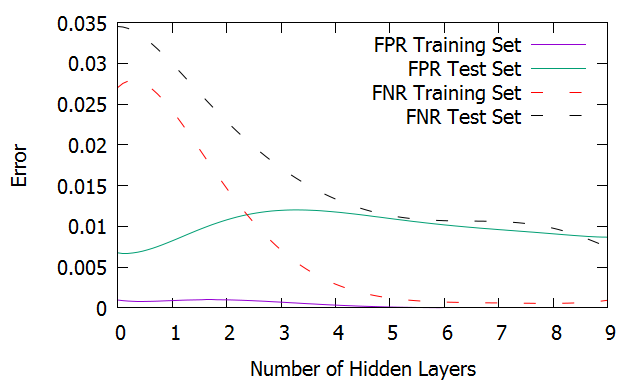
The MultiLayer Perceptron performs well on this type of data set since it can adjust the weights and filter out the less important attributes. We can also see from Figure 2 that even with 0 hidden layers the neural network performs well on this data set. As we add more hidden layers the false negative rate decreases. On the other hand, the false positive rate stays relatively the same.

The MLP also performs well on this data set even with only 1 epoch as shown in figure 3. Increasing number of epochs beyond this does not significantly improve performance. This means gradient descent has found a global minimum error after 1 training iteration. It is also possible that gradient descent only reached a local minimum error. However, given the small error this local minimum would be very close to the global minimum. The small amount of separation between training and test error curves indicates that our error comes from our model’s bias. The examples that the model is misclassifying on the training set also tend to be the examples being misclassified on the test set.

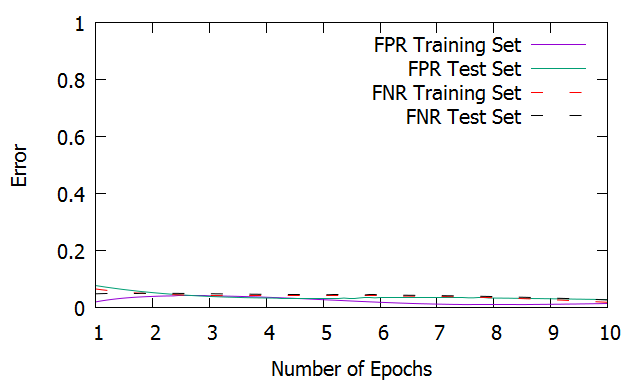
The fact that the neural network performs well after only one iteration is also evidence that our data set is linearly separable. With a learning rate of 0.3 and momentum of 0.2 used to generate these curves our weights will grow very slowly. Typically in order to represent a non linear function that separates the classifications of the data the weights will take many iterations to grow into the proper values. With only 1 iteration it is unlikely our weights had time to adjust to properly represent a non linear function.



*Figure 1: Multilayer Perceptron learning curve for Chess Set*



*Figure 2: ANN, Error vs number of hidden layers for Chess Set*

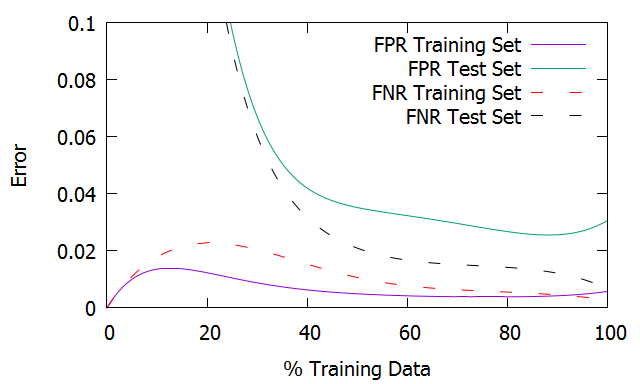


*Figure 3: Error vs Number of Iterations for MultiLayer Perceptron for Chess set*

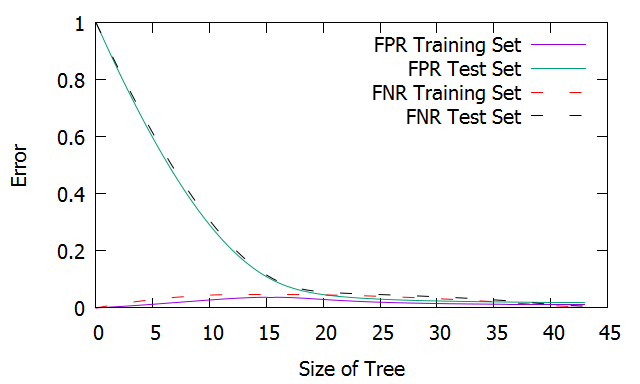
**Decision Tree**

The learning curve is shown in Figure 4. This graph was produced using the subTree Raising and Collapse tree options in Weka. From the learning curve we can see J48 performs well on this data set with about 4% false positive rates and 2% false negative rates for the testing set when about 80% of the training data is used. J48 performs well on this type of data set since it is discrete. The discrete nature allows for easy node splitting. Even with a small amount of training data the J48 performs exceptionally well. In contrast, the multilayer perceptron does not begin performing at this level until about 50% of the training data is used. The gap between the false positive error rates remains steady after about 40% of training data is used. This gap in the error rates is caused by variance.

In figure 14 we can see the complexity model as an error vs tree size graph. Here also the gap between the error rates remains fairly consistent no matter the size of the tree. However, the error rate does decrease as the size of the tree grows. We can also see that the false positive rate and the false negative rate are almost identical based on the size of the tree.



*Figure 4: Decision Tree learning curve for Chess Set*



*Figure 14: J48, Error vs. Size of Tree for Chess Set*

***Nearest Neighbors***

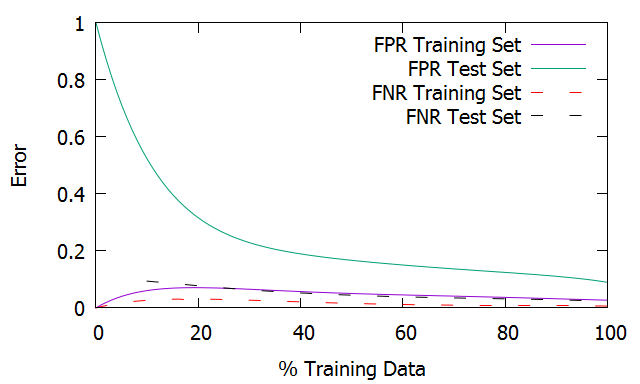
The learning curve is shown in Figure 3. For the curve shown k was set to 3. Euclidean distance was used with no distance weighting. These settings gave the best results. Here the training error remains low and the test error starts out extremely high. The test error never drops below 10% even with all the training data being used. In this case using cross validation actually created about twice as many false positives.

The IBk algorithm does not perform as well as the J48 or Multilayer Perceptron. This has to do with the discrete nature of the data set. A nearest neighbor will tend to perform better on a data set with continuous numeric features which lends itself well to a nearest distance calculation and distance weighting. For example in the case of the Chess data set’s attribute “bkblk” there are only two values that this attribute can take on. This means IBk either gets a perfect match or nothing. The neighbor with the closest distance is the one with the highest number of matching attribute values. On the other hand if the attributes had continuous values the concept of distance would be more applicable as we are not left with a match or no-match approach.

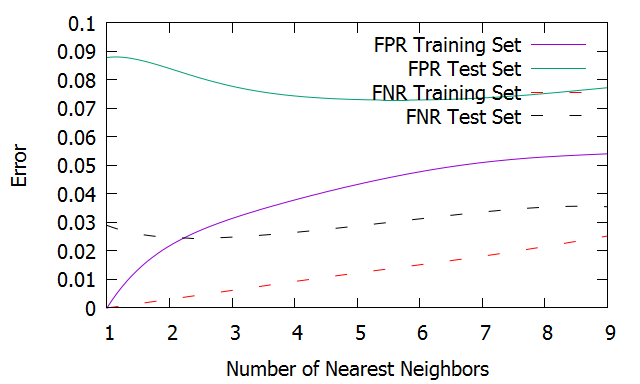
In this data set each feature has a finite set of values. The nearest neighbor algorithm has a bias which treats all attributes equally. Even if a certain set of attributes is not important for classification the IBK may match an instance from the test set to an instance in the training set based on these attributes. Distance weighting does not work well since the features are discrete.

Figure 15 shows the effect of changing k on the error rate. For a small amount of neighbors the variance starts off high and the bias low for the false positive rate. This is shown by the large gap between the test set error and the training set error. Even though the training error is low it is not a very good predictor for the test set thus it has high variance. This is also an indicator of overfitting. The training set is almost perfectly classified but it does not generalize well to the testing set. In the case of k=1 the training set is classified perfectly because when doing a nearest neighbor calculation it will find the instance that is identical. This is an overfitting of the training data and the nearest neighbor with k=1 will not fit the test data well.

As k increases the variance decreases and the bias increases. On the right side of the graph the training error has gone up while the test error has stayed relatively the same. This indicates the model is generating incorrect target values i.e. high bias. Variance is higher for a smaller number of neighbors. This is because you only get a smaller number of data point to compare your test point to. The nearest neighbor algorithm also contains a bias which states that a classification of an instance will be the same as those instances around it. In this case we have 36 attributes. The nearest neighbor algorithm will find an instance with the highest number of matching attributes and call it the closest. However, even though it is the instance with largest number of matching attributes it may have matching attributes that don’t make a significant difference in the classification. This is the curse of dimensionality. With IBk you must see a lot of data before the algorithm figures out which attributes aren’t important.



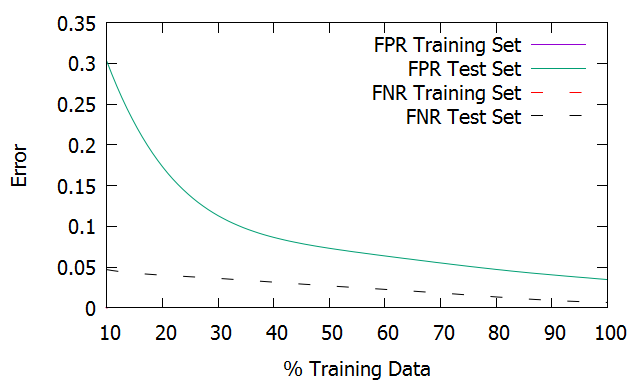
*Figure 3: Nearest Neighbors learning curve for Chess Set*



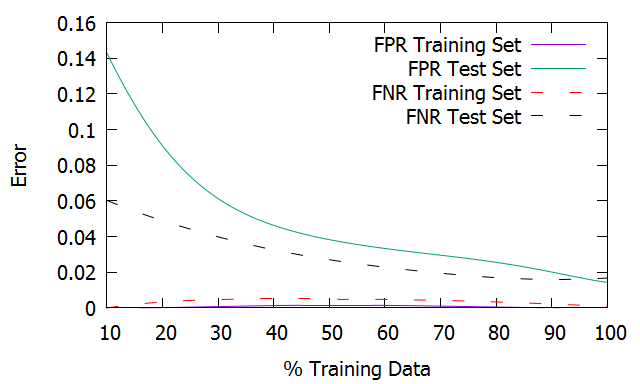
*Figure 15: IBk, error vs. number of nearest neighbors for Chess Set*

**SVM**

Figure 4 shows the learning curve when a PolyKernel is used. Figure 5 shows the learning curve when RBFKernel is used. For the PolyKernel an exponent of 5 was used. For the RBFKernel a gamma of 0.5 was used. These parameter choices gave the best results. Both of these figures show an extremely low error for the training set. The RBFKernel is a linear transformation which indicates our data is linearly separable. For the PolyKernel even though the training error remains at zero the test error continues to decrease as we add more data. For each iteration of the data size the SMO builds a “perfect” model for the training data but the “perfect” model varies with each size of the data. The model still tends to improve its fitting to the test with increasing data set size even though the training error always remains at zero. This same process is happening in Figure 5 with the RBFKernel. The model changes with varying data set sizes because the margin is being optimized differently for each. These curve trends are an example of overfitting. Even though the training data is being fit perfectly by the model when a small percentage of training data is being used the test error is still unacceptably high. The models do not being to perform well until about 50% of the training data is used. At this mark we have a false positive rate less than 10% and a false negative rate of about 5%. The portions of the curve with a small amount of training data represent the variance of the model. The small amount of training data can be used to create a perfect model for training set but the model does not generalize well because it was based on a very limited sample of data. This creates a high variance when trying to classify new examples. As we add more examples the variance from the models decreases.



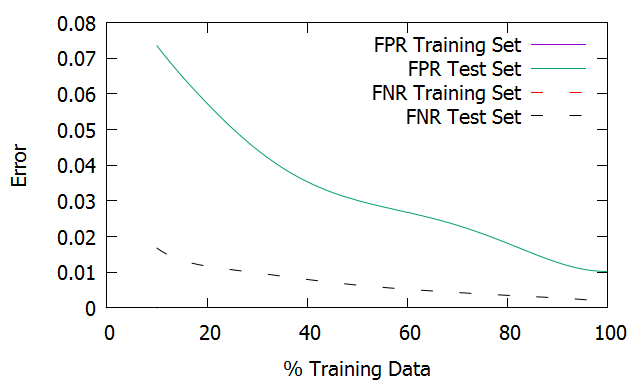
*Figure 4: SVM with PolyKernel learning curve for Chess Set*



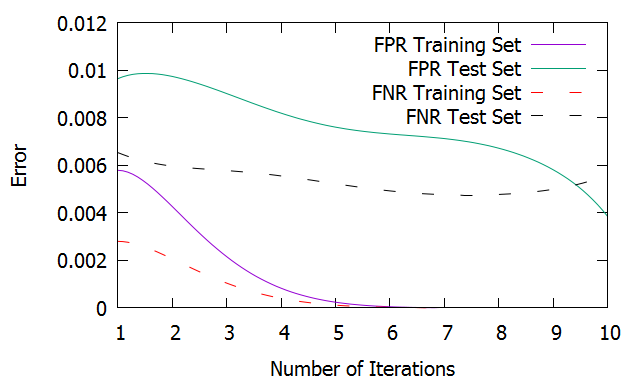
*Figure 5: SVM with RBFKernel learning curve for Chess Set*

**Boosting**

Figure 6 shows the learning curve. In the case of Figure 6 only 10 iterations are used. The boosting algorithm performs well on this data set even with very few iterations. This is due to the fact that J48 already has a strong performance on the data. We do not need to break up the performance of J48 on very many subsets of the attributes in order to create a strong classifier. It is interesting to note that initially at about 20% of the data the boosting algorithm has about twice the testing error as the J48. This may be due to the fact that creating rules for subsets of attributes over such a small data size may be creating incorrect rules. However, as we approach 100% of the data the boosting algorithm has only about 1% testing error compared to the J48’s 4% testing error. The extra error experienced by J48 is most likely due to noise. Boosting does well at removing error due to noise. This is because breaking the data into subsets allows you to average out noisy variance. Increasing the number of iterations beyond 10 does not improve performance. Most likely this last 1% cannot be removed due to noise in the data that cannot be overcome. We can see boosting performs better than any of the other algorithms on the test set. Boosting focuses on creating subsets of the examples it is not good at and then training on those examples. This allows it to improve its performance classifying instances that the other algorithms have trouble on.



*Figure 6: Boosting learning curve for Chess Set*



*Figure 30: Error vs number of iterations for boosting for chess set*

**Tic Tac Toe Data Set**

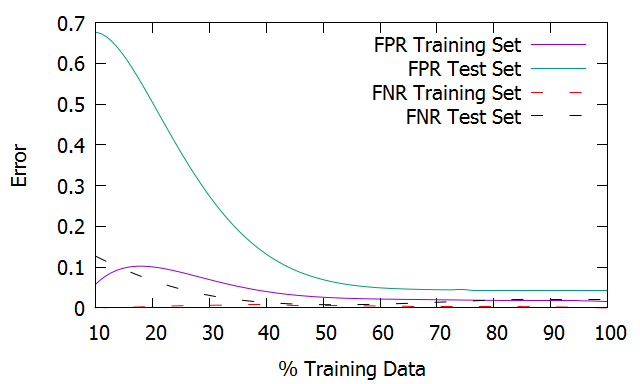
**Why is it interesting?**

The second data set used is a complete compilation of all possibilities of a Tic-Tac-Toe end game board. It is a binary classifier with “positive” or “negative” values. Positive indicates a win and negative indicates a loss. This data set contains 958 instances and 9 attributes. This data set differs from the Chess data set in a few different ways. First it is much smaller at about one third the size. Second it is slightly unbalanced with about 65% “positive” classifications and 35% “negative” classifications. This imbalance will create some differences in the way the algorithms create classifications. This data set will illuminate differences between the algorithms particularly the J48, SVM, and Boosting algorithms.

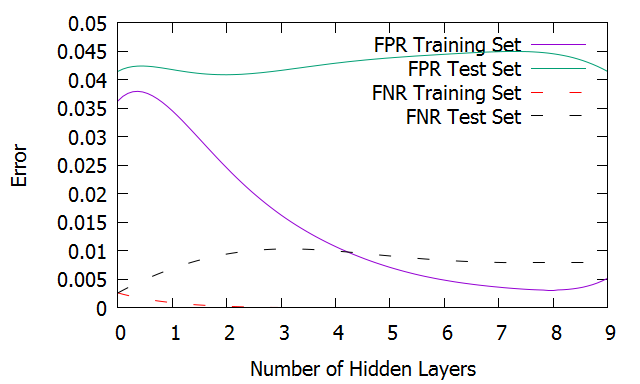
**Artificial Neural Network**

The learning curve is shown in figure 7. This curve was generated using 100 epochs and 2 hidden layers. The Multilayer Perceptron performs very well with this data set after about 60% of the data is used. This algorithm needs far less data to make an accurate prediction compared to the other algorithms. The MultiLayer Perceptron performs well on this type of data set since it can adjust the weights and filter out the less important attributes. In figure 16 we can see a graph of error vs. model complexity. As the number of hidden layers increases the variance increases. Initially at small numbers of hidden layers the training error is close to the test error. This indicates bias in the model. The model is misclassifying data in the testing set based on an inaccurate model produced from the training set. As the model complexity grows, as the number of hidden layers increases, the training error drops significantly while the test error drops only slightly. This large gap between the error rates indicates variance. The model built from the training data is accurate but will have a spread around the target value. The variance is caused by noise in the data. Most likely the test error rate will not decrease any more no matter how many hidden layers we add. In fact, as more hidden layers are added the model tends to do worse. After about 1000 hidden layers the model completely misclassifies the “nowin” target value as “win”. Typically as model complexity increases variance increases while bias decreases.

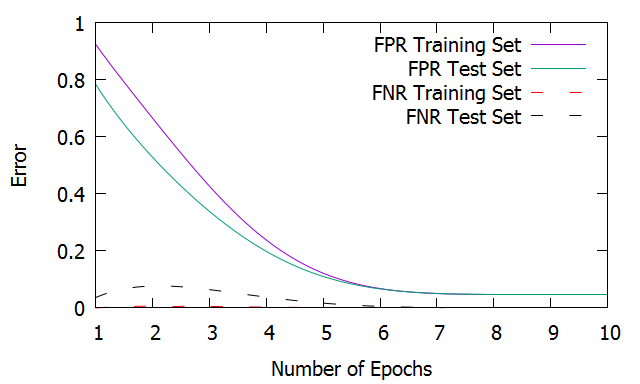
In figure 20 we can after 6 training iterations the false positive rate starts leveling out. Increasing the number of iterations after this does not alter performance. This means most likely a global minimum error has been reached by gradient decent after 6 iterations. It is possible this is a local minimum error and not a global one. However, given the low error of about 5% this local minimum would be very close to the global minimum.



*Figure 7: Multilayer Perceptron learning curve for Tic Tac Toe Set*



*Figure 16: ANN, Error vs number of hidden layers for Tic Tac Toe Set*



*Figure 20: Error vs Number of Iterations for MultiLayer Perceptron for Tic Tac toe set*

**Decision Tree**

The learning curve is shown in in Figure 8. From the learning curve we can see the gap in error rates remains steady but is slowly decreasing. This indicates that more data may be needed before we start to see a closure in the gap. The learning curve also shows that the J48 does very poorly when classifying new instances in the test set with the best performance being an approximate 40% false positive rate. However, this error rate improves if we do not use pruning as can be see by the following confusion matrix:

=== Confusion Matrix ===

a b <-- classified as

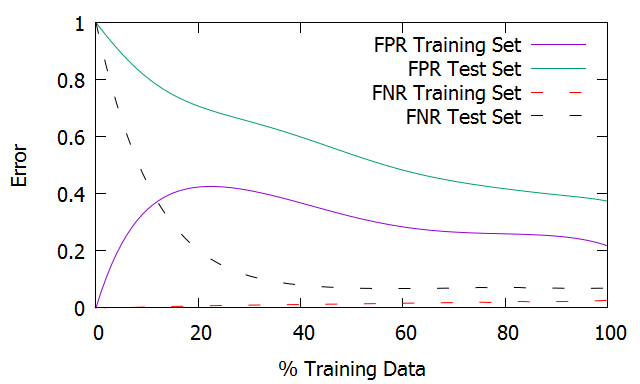
220 25 | a = positive

20 119 | b = negative

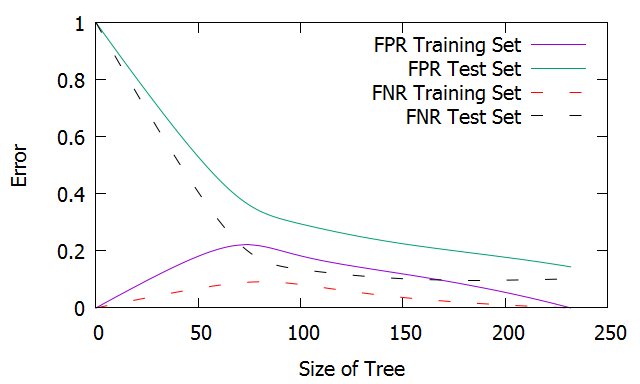
Here we see the false positive rate has been cut to 20/(20+119) = 0.143. This indicates that most likely there is not enough data for the decision tree to create a good model. Pruning the tree only creates a more inaccurate tree by taking away its specificity. In this case the J48’s preference bias for shorter trees has harmed its ability to classify future instances. However, the J48’s performance is still weaker than the MultiLayer Perceptron. The J48 needs more data to identify which attributes are not as important for classification. The MultiLayer Perceptron is able to filter out less important attributes through its choice of weights.

The complexity model in figure 17 shows the bias-variance relationship. The x axis indicates the number of nodes in the tree. As the size of the tree grows the training and test errors initially increase and then decrease for the largest tree sizes. This separation indicates variance. Even though the model can classify the training data with almost zero error it does not do well on the test set. The smaller trees have a large number of misclassifications on both the test and training set. The less complex models will tend to be weaker and misclassify a larger number of examples. This is indicative of the preference bias of J48. In this case the bias harms the classification significantly.

Also notice there is huge discrepancy between the fale positive rate and the false negative rate. This is caused by the imbalance in the classification of the training set. The training set contains about twice as many positive classifications than negative. This means the algorithm will tend to have false positives. This is a case where it helps to have a false positive rate and a false negative rate as our error metrics. Using Weka we can see that the accuracy is measured to be 82% even though the J48 has misclassified almost half the negative examples.



*Figure 8: Decision Tree learning curve for Tic Tac Toe Set*

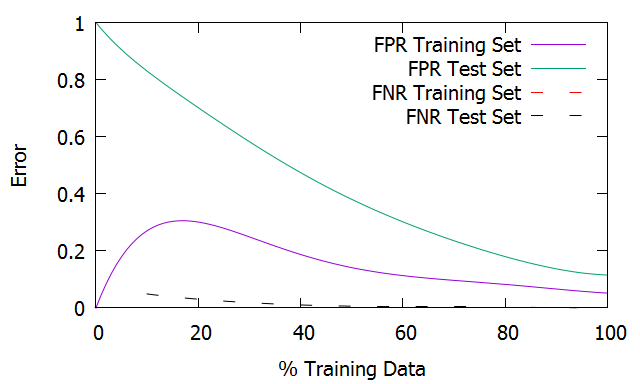


*Figure 17: J48, Error vs. Size of Tree for Tic Tac Toe Set*

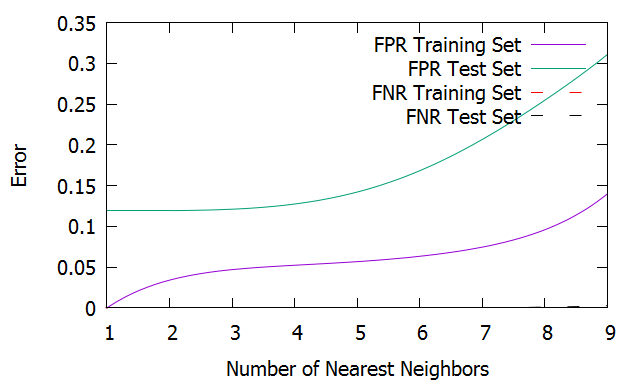
**Nearest Neighbors**

The learning curve is shown in figure 9. Similar to the IBk algorithm for the Chess data set the IBk also performs poorly on this data set. This is because the data set contains attributes with discrete values. This data set does not lend itself well to the nearest neighbors algorithm. This is because a single change in a square of the tic tac toe board is likely to change the outcome of the game. By considering boards that are similar the IBk is classifying new instances on the assumption that a board layout similar to the board layout we are looking at will produce the same results.

Figure 18 shows that the false positive rate increaes with larger k values. The increase in error is caused by the lack of distance weighting. Adding more neighbors without distance weighting will cause the model to be weighted too heavily by very distant instances. Similar to the complexity model for the Chess data set, we see that for small values of k the model has high variance. It is basing its predictions on too few points. Thus, when trying to predict a value in the test set it has a large amount of swing around the data point it is looking for. As the number of neighbors increases we see that the test error also increases. This is caused by a bias in the algorithm which states that every neighbor being considered is equally valid. This bias can cause a hugh discrepency for this data set. The end game of tic tac toe is dictated by all nine squares on the board. If even a few squares are different the end game result is likely to change. When considering many neighbors the algorithm is essentially looking at a completely different game board.



*Figure 9: Nearest Neighbors learning curve for Tic Tac Toe Set*



*Figure 18: IBk, error vs. number of nearest neighbors for Tic Tac Toe Set*

**SVM**

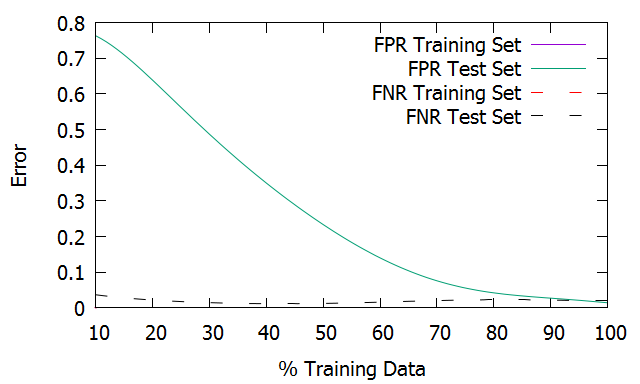
The learning curve for the SMO using Polykernel is shown in figure 10. The learning curve for the SMO using RBFKernel is shown in figure 11. For the PolyKernel an exponent of 5 was used. For the RBFKernel a gamma of 0.5 was used. These parameter choices gave the best results. Comparing the two figures we can see the difference between the Polykernel and the RBFKernel. The PolyKernel performs much better on the test set than the RBFKernel for the false positive rate. For the false

negative rate both algorithms perform well.

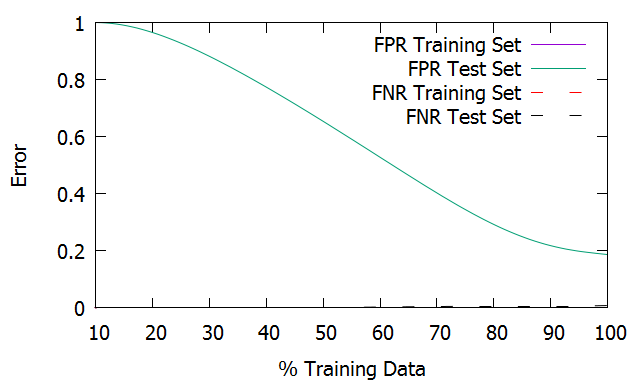
Looking at the figures we see evidence that both kernels are able to create a transformation where the training data is linearly separable. This is evidenced by the fact that there is zero training error in both figures. However, the Polykernel does a better job of creating a linear separation between the two classifications. The PolyKernel does a better job of choosing a linear separator that is halfway between the two sets of classification points. When the model is trained it creates a linear separator between the two different classifications. When it is generalized to the test set the linear separator that is closer to the halfway point between the two sets of classifications will do better. This is indicated by the lower error in Figure 10. When the Polykernel uses its linear separator on the test set it does a better job of dividing the new classifications. The RBFKernel has a higher error overall because the linear seaprator it creates does not separate the test classifications as well. The RBFKernel will tend to create a linear separator that has many negative classifications on the side that should only contain positive classifications.

The small false negative rate compared to the larger false positive rate for the test set indicates that the linear separation the kernels are creating between the classifications will tend to have negative classifications on the side that is only supposed to contain positive classifications. This is likely caused by the linear separator being too close to the negative classifications. We have very few false negatives because the linear separator is close to the negative side. In the case where the data is linearly separable we will see out error is either false positive or false negatives but not usualy both. Our data is linearly separable which means our errors are being created by the linear separation being too close to one side.

With the PolyKernel the error is caused by the bias since the training and testing error are close together. With the RBFKernel the error is caused more by variance since the training error is low but the test error is high. This variance is caused by the spread of classifications around the linear separator.



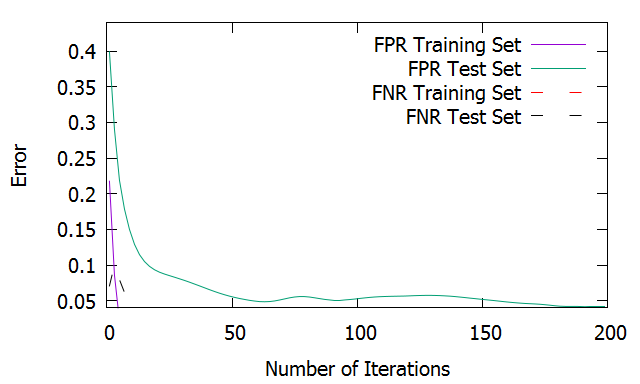
*Figure 10: SVM with PolyKernel learning curve for Tic Tac Toe Set*



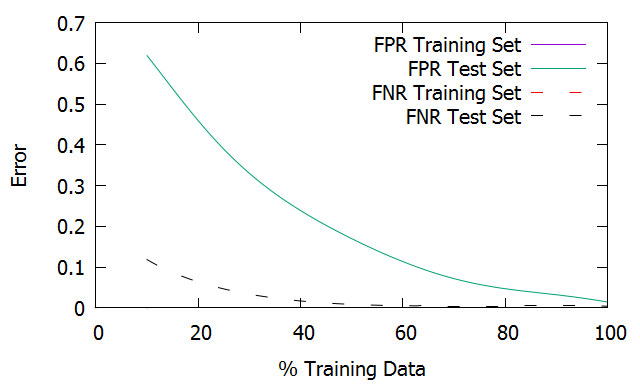
*Figure 11: SVM with RBFKernel learning curve for Tic Tac Toe Set*

**Boosting**

The boosting algorithm does not perform as well on the Tic Tac Toe set with a standard 10 iterations. This can be seen in Figure 19. The test error is high until about 50 iterations and levels out at about 200 iterations. The learning curve using 200 iterations is shown in figure 12. With 200 iterations the boosting algorithm starts performing significantly better at around 80% of the training data. With 100% of the data being used the testing error is about 1%. Despite J48 performing poorly on this data set on its own, with boosting we get the best performance out of all 5 algorithms. With boosting using a weak classifier that initially overfits can be a concern because it can cause the boosting algorithm to overfit. However, we know from our analysis of the J48 algorithm that overfitting is not a problem for our weak classifier. Since AdaBoost applies weights to the rules it creates from applying the weak learners it is able to apply smaller weights to the rules that are creating the bulk of errors for J48.



*Figure 19: Error vs Number of Iterations for AdaBoost for Tic Tac toe set*



*Figure 12: Boosting learning curve for Tic Tac Toe Set*

**Performance**

Figure 21 shows the computation times in seconds for the algorithms. Comparing the time to build a model and apply it between the different algorithms gives us some insight into how each algorithm is behaving. We can see for the nearest neighbors algorithm, IBk, that the time to apply a model is significantly higher than the time to build it. This is because nearest neighbors algorithms do not build a model based on the training data. The significant portion of their time comes from querying the training data and comparing it to the test data. When you apply IBk it performs distance calculations between the training and test instances to find the nearest neighbors. It has no need to have a saved model. We can also see that the amount of memory required to store the model is much lower than most of the other algorithms. This is again because IBk does not build a model but simply performs calculations at runtime. For these times there were 3 neighbors being considered. Thus the query time was log(n) + 3 where n is the number of instances and the space needed is n. In contrast, for all the other algorithms the time required to build a model is higher than the time required to apply a model. This is because these algorithms must actually build a model. Once the model is built, however, the model is simple to apply to new data. Thus, the appication time is lower than the build time.

The ANN training time is low because of the small number of hidden layers and training iterations. We can see here that AdaBoost and J48 are the fastest algorithms. Boosting is known as a fast algorithm. Even though it acting over many iterations, the data is broken into small subsets so there is less data to operate on. If boosting performs quickly it makes sense that J48 would perform quicker since it is only being run once compared to being run multiple times with boosting. SMO takes the longest of all the algorithms. This is because when we are trying to maximize the margin we are trying to solve a quadratic programming problem. In the best case a quadratic programming problem can be solved in polynomial time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Chess** | | **Tic Tac Toe** | |
|  | Time to build model | Time to apply | Time to build model | Time to apply |
| **J48** | 0.03 | 0 | 0 | 0 s |
| **MultiLayer Perceptron** | 0.5 | 0.01 | 0.12 | 0.01 |
| **SMO PolyKernel** | 1.4 | 0.18 | 0.35 | 0.1 |
| **SMO RBFKernel** | 3.44 | 0.43 | 0.24 | 0.06 |
| **ADABoostM1** | 0.25 | 0.01 | 0.03 | 0.01 |
| **IBk** | 0 | 1.1 | 0 | 0.04 |

*Figure : Time to build and apply model in seconds*

|  |  |  |
| --- | --- | --- |
|  | **Chess** | **Tic Tac Toe** |
| **J48** | 21 kB | 21 kB |
| **MultiLayer Perceptron** | 35 kB | 20 kB |
| **SMO PolyKernel** | 776 kB | 169 kB |
| **SMO RBFKernel** | 792 kB | 173 kB |
| **ADABoostM1** | 129 kB | 173 kB |
| **IBk** | 162 kB | 67 kB |

*Figure : Space required for model*

**Conclusion**

For most algorithms the false positive rate was of more concern than the false negative rate. For the Tic Tac Toe set this makes sense because there are twice as many positive classifications as there negative classifications. The algorithms well tend to bias in the direction of the majority class.

References

Engelmore, R., and Morgan, A. eds. 1986. *Blackboard Sys­tems.* Reading, Mass.: Addison-Wesley.