Assignment #1: Supervised Learning Techniques

Student: Jeff Hughes
GT Account: jhughes63
GTID: 903102859
Class: Georgia Tech Machine Learning 7641
Due: 2/2/15

Introduction

The purpose of this assignment was to analyze the characteristics of the five supervised learning algorithms shown in the table below. To achieve this objective, I defined two classification problems and then analyzed the key characteristics of each learning algorithm when applied to these classification problems.

Type of Algorithm	Specific Module Used	Author	Source
Decision Tree	C5.0 Decision Tree with Pruning.	Quinlan	C50 Package for R[i]
Neural network	NNET	Fritsch & Guenther	NNET Package for R[ii]
Boosting	C5.0 Decision Tree with Adaboost	Quinlan	C50 Package for R
Support Vector Machine	Kernlab Support Vector Model	Karatzoglou, Smola, Hornik, Zeileis	Kernlab Package for R[iii]
K-nearest neighbors	Class Package version of KNN	Ripley & Venables	Class Package for R[iv]

This paper has 3 main sections. The first section describes the Classification Problems and the data associated with those problems. The second section covers my Analysis, including a brief examination of each of the learning algorithms. The final section is the conclusion. I've used a question and answer format in the analysis section so the TA grading this paper can quickly see the key questions asked and answered.

Classification Problems

I chose 2 classification problems as summarized in the table below.

Problem	Description	Data Source
"Credit" A Loan Classification Problem	Feed data from a German credit agency to machine learning algorithms and see how effective they are at predicting which loans will default. The data consists of 1000 samples, each containing 14 attributes and one target variable. Some attributes are integer valued. Others are nominal valued. The target variable is a binary value indicating whether the loan defaulted (yes=default, no=did not default).	Statlog (German Credit Data)[v]
"Letters" An OCR Classification Problem	Feed Optical Character Recognition (OCR) data to machine learning algorithms and see how effective they are at identifying the actual letter of the alphabet that was originally scanned. The dataset contains 20,000 samples, each having 16 integer-valued attributes and one multi-valued output variable with 26 possible values – one for each letter of the English alphabet.	Letter Recognition Dataset[vi]

Why are these problems interesting?

I selected these problems primarily to highlight the strengths and weaknesses of the different algorithms tested. Different machine learning algorithms perform well on different types of problems. A Decision Tree algorithm (or boosted Tree algorithm) might be expected to perform well against the "Credit" dataset, because Trees are somewhat robust to missing data and they work well for problems with discrete outputs. Conversely, Neural Networks and Support Vector Machines might be expected to perform well against the "Letters" dataset, because these algorithms are suited to problems with continuous-valued inputs; and they have performed well against complex problems such as facial recognition or language recognition.

Other factors also make these classification problems *interesting*:

- Each of these problems has output values that can be objectively measured as 'correct' or 'incorrect', unlike other classification problems where subjective judgment is involved (e.g. 'wine quality').
- Each of the datasets are large enough to be divided into a training dataset and a cross-validation test dataset, where the training dataset will be further partitioned for analysis purposes.
- Each dataset contains quite a few attributes that are used as inputs, which facilitates a variety of testing (including the ability to winnow some data items and still obtain meaningful results)

How did I prepare the data?

I started with the data described in the table on page 1. After running the Credit dataset through all of the learning algorithms, I discovered that 39.4% of the checking_balance data was 'unknown' and this had a significant negative impact on the predictive accuracy of all of the algorithms. I learned a lot by exploring that issue and will discuss that in the analysis section. Ultimately I decided that I wanted slightly higher quality data for this assignment, so I modified the checking_balance data in the Credit dataset. I did so using the random number generator in Excel to generate one of 4 possible values for the checking account balance (<0, 0-200, 201-500, and 500+). I biased the random number generator to ensure that "good" loans were slightly more correlated with higher checking account balances. The original data had this same bias, but my new data does not have any 'unknown' values. I also decided to discard the 'housing' and 'phone' attributes because they did not appear to be useful to the models. Finally, I created a second version of the "credit.csv" file for use with the NNet and SVM algorithms, which require numeric input. Some variables in the original data were nominal values that clearly represented "ordered" attributes (e.g. Employment duration, credit history, etc.). I mapped these attributes to ordinal values that made sense (e.g. credit history ranged from critical=1 to perfect=5). One nominal attribute ("loan purpose") could not be treated in this manner. An expert banker might be able to make a value judgment that one 'purpose' is better than another, but I cannot. They all seem arbitrary. For this reason, I mapped the different values of the 'purpose' attribute to binary encoded dummy variables. Aside from these changes just described, my data is consistent with the original data pulled from the sources previously identified. Finally, it is worth noting that all numeric data is normalized when it is read into my programs. This is not necessary for some algorithms, but it prevents large number bias for other algorithms.

How did I perform the tests?

I used the R Programming Language[vii] and wrote scripts to load the machine learning packages identified earlier in this document and then execute the machine learning algorithms. These scripts randomized the data file and then set aside 10% of the data for cross-validation. The remaining data was used for training. During each training run, I varied the size of the training dataset such that it ranged from 50% to 90% of the total data pool (I did this so I could examine the effect of larger training datasets on the performance of the learning algorithms). I then started the cross-validation phase, running the test dataset through the trained model. I recorded the results and created the graphs and tables you'll see throughout this report.

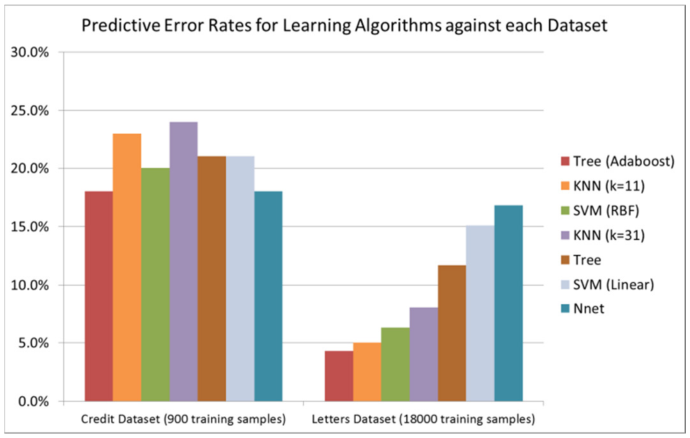
Note that when using 90% of the data for testing, the Credit dataset had 900 training samples and 100 test samples and the Letters dataset had 18000 training samples and 2000 test samples.

Analysis Section

Which algorithms performed best?

There are many ways to characterize the performance of machine learning algorithms, including training time, training samples required, execution time (after training), resource utilization, etc. For this assignment, we are working with small datasets so all algorithms have very fast test execution times as shown in the table below (with SVM being the one outlier). Training time will be examined later in this report. The performance attribute of greatest interest is the predictive accuracy of the learning algorithm as measured through cross-validation. The graph below illustrates the predictive *error rate* for each of the algorithms when trained over 90% of the dataset and cross-validated with the remaining 10% of the dataset. Error rate is the inverse of accuracy in this case, so smaller values are better. For the Credit dataset, all algorithms had relatively high error rates. For the Letters dataset, there were significant differences in the predictive performance of the algorithms. The Adaboosted Tree algorithm was the best performer overall.

	Machine Learning Algorithm Performance	Cross-validation Test Error Rate		Execution Time (secs) against Test Data	
	Machine Learning Algorithm	Credit Dataset	Letters Dataset	Credit Dataset	Letters Dataset
🟥	Tree (Adaboost)	18.0%	4.3%	0.0	0.7
🟧	KNN (k=11)	23.0%	5.0%	0.0	0.6
🟩	SVM (RBF)	20.0%	6.3%	0.0	6.4
🟪	KNN (k=31)	24.0%	8.0%	0.0	0.6
🟫	Tree	21.0%	11.7%	0.0	0.1
⬜	SVM (Linear)	21.0%	15.1%	0.0	0.2
🟦	Nnet	18.0%	16.8%	0.0	0.0



Why did some learning algorithms have higher predictive performance than others?

Let's focus on the results from the Letters classifier where the difference is obvious. It's not surprising that Adaboost performs well. Boosting algorithms focus on the training data points with the highest error and force the model to fit those data points better, which often results in better accuracy against the test data.

KNN is known to perform well on problems where the target function is complex, and that appears to be the case for the Letters classifier. The KNN package I used employs the standard Euclidean distance metric to match each new test sample to previously observed samples. That approach is very effective here, suggesting that there must be a lot of similarity between data samples representing the same alphabetic character. The results in the graph show that the KNN algorithm performs better with a value of K=11 than with a larger value of K=31. This suggests that the difference in attribute values between dissimilar letters is relatively small, so we only want to look for other samples that are *very* similar to the one we are trying to classify.

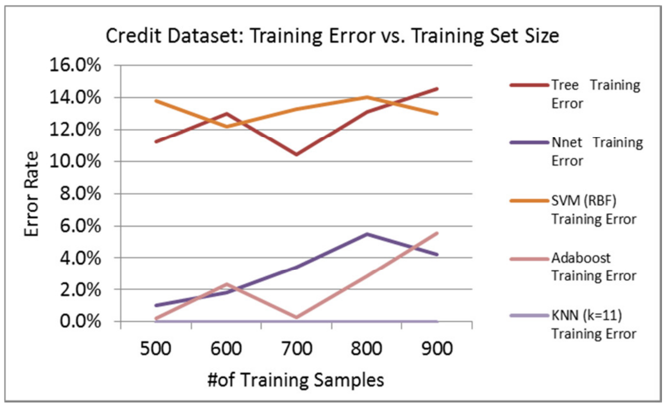
SVM is also very accurate (<7.5% error rate). This means it is able to find hyperplanes that mathematically separate the different characters we are trying to classify. As shown in the graph, however, the linear “vanilladot” kernel does not perform as well as the RBF kernel. This suggests that the characters are not exactly linearly separable. A more sophisticated function is required, which is why the RBF kernel excels.

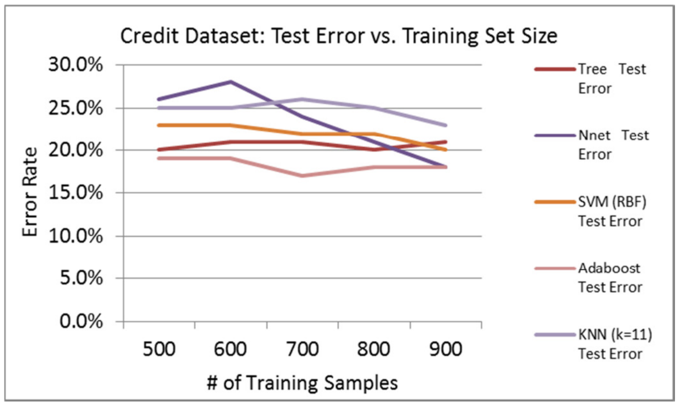
What is most surprising is that the Neural Network algorithm performs relatively poorly. Given sufficient time to tune the parameters of this model, I believe the Neural Network could be made to perform better than shown here. Additional training iterations would also improve performance to some degree. I’ll examine that aspect later in the analysis section.

Why didn’t the algorithms perform better on the Credit Classification problem?

I believe the problem was that the dataset was noisy – by which I mean there were missing data points and contradictory data points where similar inputs produce opposite output values. My testing against the original (unmodified) version of the Credit dataset resulted in a predictive accuracy of 73% for my best performing learning algorithm (Adaboost). Other researchers[viii] reported similar results when working with the same German Credit dataset, with predictive accuracy in the range of 75%. Researchers O’Dea, Griffith and O’ Riordan specifically mention noise being a factor with this dataset[ix].

The actual Training and Test error rates for each of the learning algorithms over the Credit dataset are shown below. With the exception of the Neural Network, none of the learning algorithms noticeably improve as we increase the number of training samples, suggesting they have reached steady state. Regarding the improved test performance of the Neural Network shown below, I do not believe the trend continues past 900. In fact, I ran a k-fold validation test that calculated the actual error rate over 900 samples is around 20%.

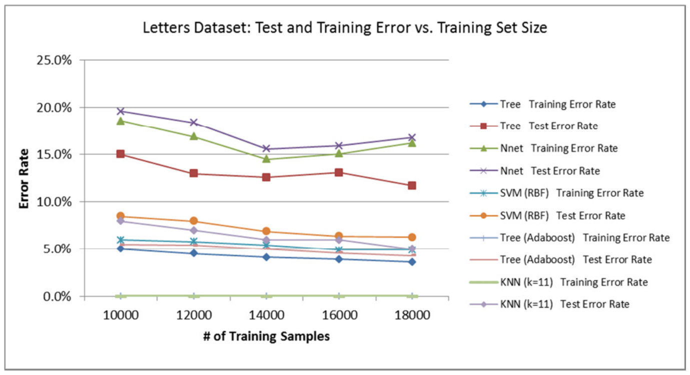


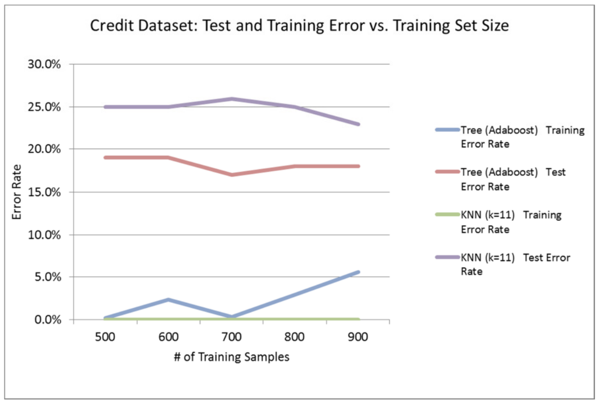


The behavior of the learning algorithms as more data is received, coupled with the experience of other researchers leads me to conclude that data quality issues with the Credit dataset limit the performance of the learning algorithms. My belief is reinforced by the fact that the test accuracy improves markedly for some algorithms when I replace the checking_balance data in the dataset with cleaner data. Despite this noisy data problem, I decided to continue working with the Credit dataset because I felt I could learn a lot from it.

How many samples do we need to train the models? How do we know we haven't over-fit the data?

For the letters dataset, the chart below (left) shows that test error over the test data set continues to decrease as we train with larger and larger training sets for all but one of the algorithms. Since both test error and training error seem to decrease with additional training samples, I concluded that more training data improves the predictive performance of the models. The one exception is the Neural Network, which sees an increase in training error and test error once we exceed 14,000 training samples. This is not a sign of over-fitting, however, because the training error and the test error both increase.

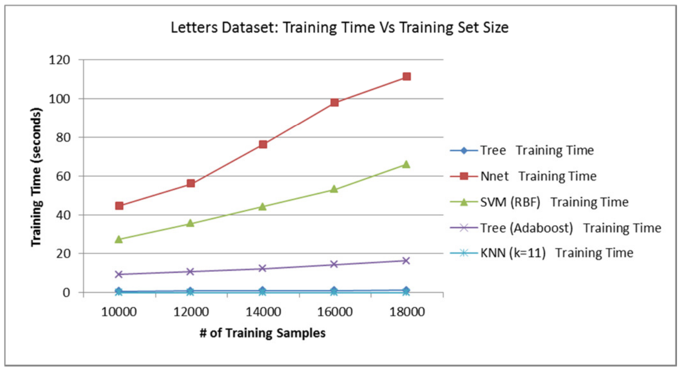


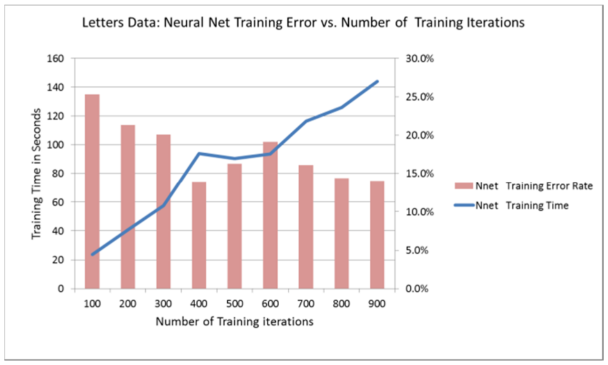


The chart on the right is a subset of data recorded for the credit classifier. There are two interesting things to note in this chart. First, there *are* signs that the KNN model is over-fitting the data as we exceed 700 training samples. The chart shows that test error continues to increase in this case, even though the KNN training error remains at zero. The Adaboosted Tree algorithm exhibits the opposite behavior. Training error remains flat as we exceed 700 training samples, but the test error continues to decrease. This is because boosting algorithms are able to reduce their margin of error on the training data even when there is no overall improvement in training classification. Continuing to reduce the margin of error on the training set, however, results in improved predictive performance against the test data, as can be seen in the figure on the right.

How long does it take to train the models?

I looked at this a couple of different ways. The chart below on the left shows that training time increases linearly as we increase the number of training samples. This is true for all algorithms. The Neural network takes the longest amount of time to train because it has to iterate over the data samples many times in order to establish proper weights for each node. In this case, with a network of 26 outputs and 13 hidden nodes the Neural Network takes almost 120 seconds to train with a dataset of 18,000 training samples (see left graph).





Of course, the training time for the Neural Network is impacted by the number of iterations performed over the training samples. The iterations are required so the Neural Network can establish the correct weighting factor for each node. The neural network on the left was capped at 500 iterations. The chart on the right shows how training time increases (and *training error* decreases) as we increase the number of training iterations. Once we exceed 500 iterations, there is only a small reduction in training error. For this Neural

Network, 500 training iterations seem adequate. Although not shown here, the Adaboost tree algorithm also iterates over its dataset multiple times. A similar analysis shows diminishing returns above 50 iterations.

How does data quality impact the predictive performance of the learning algorithms?

During my testing with the original (unmodified) version of the Credit dataset, I observed high test error rates for every learning algorithm. The same algorithms performed well against the Letters dataset, so this suggested that there was a problem with the Credit data. Thinking that the 'curse of dimensionality' might be impacting my results, I decided to *winnow* some of the data attributes from the input file. A brief search of published research papers suggests that others have successfully employed winnowing to improve performance[x]. I eliminated the 'purpose' attribute and the 'other credit' attribute. I chose these attributes because the output of the C5.0 Tree Model (and Adaboosted Tree) showed these variables were not really being used in their models. I tested against a subset of the other learning algorithms (KNN, SVM and NNET) and noted a reduction in predictive error that ranged from 3% to 10% of the original error.

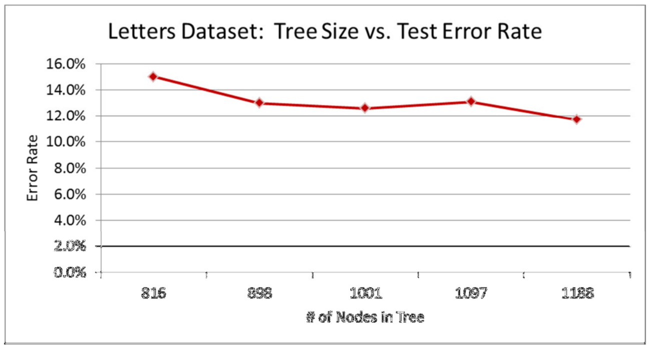
Then I modified the data for the checking_balance attribute, as noted in the data preparation section of this report. This yielded a significant improvement in test accuracy across all learning algorithms. Results can be seen in the table below. It is not meaningful to examine the improvement in error rate for any one algorithm because that improvement depends on the type of changes made to the data. It is interesting to note the relative improvement in performance between the various algorithms, however. The Neural Network algorithm and the SVM algorithm both saw very large performance improvements relative to other algorithms. This suggests that these algorithms may be more sensitive to missing or inaccurate data.

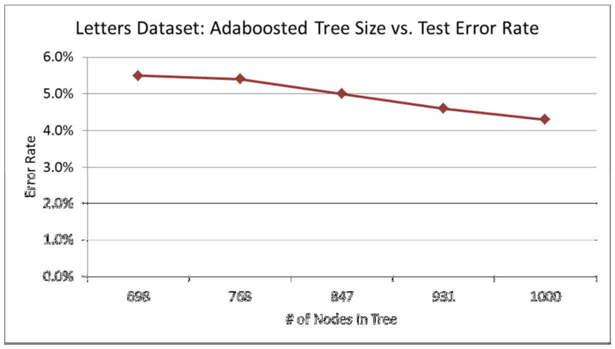
Difference in Predictive Error	Credit Dataset		
	Original	New	Improvement
Tree	30.0%	21.0%	30%
Nnet	43.0%	18.0%	58%
SVM (RBF)	40.0%	20.0%	50%
Tree (Adaboost)	27.0%	18.0%	33%
KNN (k=11)	37.0%	23.0%	38%

How did I assess the performance of the Tree algorithm?

The C5.0 tree algorithm attempts to split nodes in a way that will maximize the normalized information gain. This means that at each level in the tree it looks for the attribute that it can use to divide the tree in a way that will most effectively partition the output values, which should help it determine the correct solution with the minimum amount of tree traversal. C5.0 is a self-pruning tree. For these reasons, my expectation was that little or no tuning would be required to optimize tree performance. One thing I did look at was the performance of the tree relative to tree size.

When trained with larger and larger datasets, Trees tend to grow in size. Initially, this improves the predictive performance of the algorithm when run against the test set. Eventually the tree over-fits the data, however, and pruning may be required. I did not expect to see an over-fitting problem because the C5.0 Tree algorithm has pruning automatically enabled. Looking at a plot of test error versus tree size confirms that over-fitting is not a problem. If it were, we would see the accuracy continue to increase on the training dataset while the accuracy decreases on the test dataset. I concluded that the tree was performing within expectations.



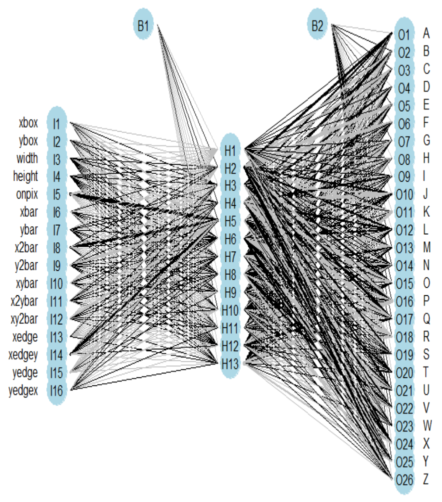


What did I do to improve the performance of the Boosting algorithm?

For this project, I used the C5.0 Tree algorithm[xi], which has an optional boosting feature that implements the Adaboost algorithm. The results at the beginning of the Analysis section showed that the AdaBoosted C5.0 Tree was the best performing algorithm. Its predictive error rate of 4.3% was just over 1/3 of the error rate of the standard C5.0 Tree algorithm which had an error rate of 11.7%. To achieve this performance, I tested with different values for the number boosting iterations, finally settling on 50 iterations. I later learned that performance would be just slightly better if I increased to 90 iterations.

What did I do to improve the performance of the Neural Network? Could anything else be done?

Based on what I had read about Neural Networks, I expected this to be one of the best performing algorithms on the letters classifier problem. That problem involves multiple integer-valued inputs and a multi-level output, so it seems like a good match for a Neural Network. I started out using default parameters for the Neural Network, specifying a value of “1” for the number of hidden nodes. The test error was approximately 80%, which was absolutely terrible. I switched to the NNET package and then began playing with the number of hidden nodes in the network. I quickly settled on 13, because that gave a 2-to-1 ratio between the number of outputs and the number of internal nodes. I used the default ***logistic*** activation function for NNET. See figure to the right for an illustration of the resulting neural network[xii]. Given the huge improvement in performance after increasing the number of hidden nodes, it seems possible that other network architectures could have yielded even more improvement. It’s also possible that different activation functions would improve performance. Time permitting I would examine those options.



What did I do to improve the performance of the SVM model? Could anything else be done?

As shown in the graph at the beginning of the Analysis section, I tried two different kernels in the SVM model. First I tried the vanilladot kernel which is a linear kernel. Next I tried the Radial Basis Kernel (RBF). The latter kernel performed significantly better, which suggests that the datapoints are not exactly linearly separable, but can be separated using more sophisticated mathematical models. The RBF Kernel exhibited less than half the error rate of the Linear kernel, making it the 3rd-best performing algorithm. Given the huge improvement in performance after changing kernels, it seems possible that we could get even better performance out of a different kernel model, or by tuning kernel parameters.

What did I do to improve the performance of the KNN model? What else could I have tried?

For the KNN algorithm, the obvious variable to change is K. I chose a couple of different values for K as shown in the results section, and ran all of my tests using those values. After seeing how well the algorithm performed, I went back and tried a couple more values of K. It turns out that the model performs even better with smaller values of K. In fact it achieved a 4.1% error rate on the letters dataset with a value of K=3. The package I used only supported Euclidean distances. Otherwise it would have been interesting to try different distance metrics to see if further improvement was possible. Trying a weighted distance metric would also have been interesting.

Conclusion

Most of the machine learning algorithms did an adequate job against the Credit Classification problem but performed extremely well against the Letters Classification problem. This was a surprise because the Letters Classification problem seems like a much more difficult problem. It relies on 16 integer-valued inputs and produces a discrete output with 26 levels. By contrast, the Credit classifier problem relies on 14 nominal valued inputs with only one binary output.

It seems clear that the quality of the data had a significant impact on the performance of the learning algorithms in this experiment. The Credit dataset had missing data and it included some data that did not seem very likely to predict whether a loan would default (e.g. purpose, age, dependents, etc.). More importantly, the data in the Credit dataset appeared to be very noisy, which limits the predictive accuracy we can realistically expect to see from any machine learning algorithms.

Replacing the missing data in the checking_balance attribute with a new dataset improved the performance of all learning algorithms against the Credit Classification problem, but some algorithms (i.e. Neural Network and SVM) saw much larger gains than others, suggesting that they were more sensitive to missing data. Other tuning steps – such as SVM kernel model changes, additional training cycles for the Neural Network and Adaboost – were very effective at improving the performance of the learning algorithms on the Letters dataset but had little or no impact on the performance of these algorithms against the Credit dataset. This is further evidence that the noise in the Credit dataset limited the performance of the learning algorithms.

The Adaboost algorithm performed extremely well against the Letters dataset. As predicted by Robert Shapire in his 'Boosting the Margin' paper[xiii], letting the Adaboost algorithm continue to iterate over the training data improved the algorithm's performance against the test dataset long after the point where training error had gone to zero. KNN also performed extremely well against the Letters dataset, especially with very small values of K. The SVM algorithm performed well, as expected. The Neural Network algorithm did not perform well against the Letters Dataset, but it was one of the best performing algorithms against the Credit dataset after I eliminated missing data from that dataset.

---

[i] Max Kuhn, Steve Weston and Nathan Coulter. C code for C5.0 by R. Quinlan (2014). C50: C5.0 Decision Trees and Rule-Based Models. R package version 0.1.0-21. http://CRAN.R-project.org/package=C50

[ii] Venables, W. N. & Ripley, B. D. (2002) Modern Applied Statistics with S. Fourth Edition. Springer, New York. ISBN 0-387-95457-0

[iii] Alexandros Karatzoglou, Alex Smola, Kurt Hornik, Achim Zeileis (2004). kernlab - An S4 Package for Kernel Methods in R. Journal of Statistical Software 11(9), 1-20. URL http://www.jstatsoft.org/v11/i09/

[iv] Venables, W. N. & Ripley, B. D. (2002) Modern Applied Statistics with S. Fourth Edition. Springer, New York. ISBN 0-387-95457-0

[v] Data files from Lantz, Brett. *Machine Learning with R: Learn How to Use R to Apply Powerful Machine Learning Methods and Gain an Insight into Real-world Applications.* Birmingham, UK: Packt Publishing, 2013.Original data from *Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.*

[vi] Data files from Lantz, Brett. *Machine Learning with R: Learn How to Use R to Apply Powerful Machine Learning Methods and Gain an Insight into Real-world Applications.* Birmingham, UK: Packt Publishing, 2013.Original data from *Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.*

[vii] R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

[viii] Bart Baesens, Rudy Setiono, Christophe Mues, Stijn Viaene, and Jan Vanthienen. "Building Credit-Risk Evaluation Expert Systems Using Neural Network Rule Extraction and Decision Tables" *ICIS 2001 Proceedings* (2001). Available at: http://works.bepress.com/stijn_viaene1/3

[ix] O'Dea, Paul, Josephine Griffith, and Colm O'Riordan. "Combining feature selection and neural networks for solving classification problems." Proc. 12th Irish Conf. Artificial Intell. Cognitive Sci. 2001.

[x] Ratanamahatana, Chotirat, Gunopulos, Dimitrios. "Feature Selection for the Naïve Bayesian Classifier Using Decision Trees", Applied Artificial Intelligence, 17:475–487, 2003,

[xi] C5.0 algorithm by Robert Quinlan. C5.0 Tree Package itself comes from CRAN repository.

[xii] Plot produced using devtools. Hadley Wickham and Winston Chang (2015). devtools: Tools to Make Developing R Packages Easier. R package version 1.7.0. http://CRAN.R-project.org/package=devtools

[xiii] Schapire, Robert E., et al. "Boosting the margin: A new explanation for the effectiveness of voting methods." Annals of statistics (1998): 1651-1686.