Final Project Report

Daniel Rogers
Stanford ID # 05321389

Abstract

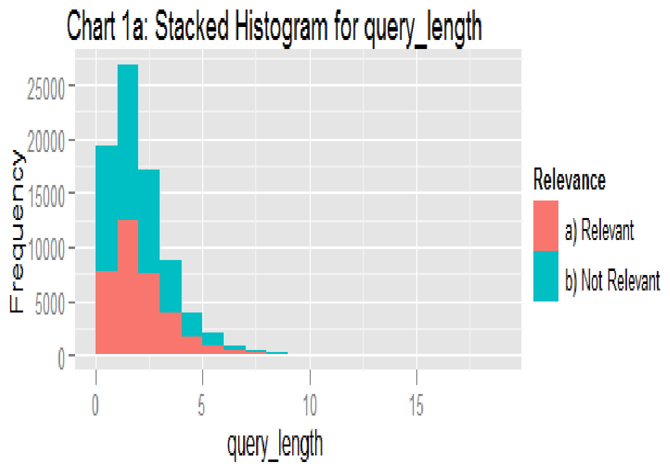
The object of this paper is to explore various data mining tools while finding the best method to make relevant predictions over a set of sample query data. I will consider five different classifiers: k-nearest neighbor, the AdaBoost variant of boosting using decision trees, random forest, naïve Bayes, and support vector machine.

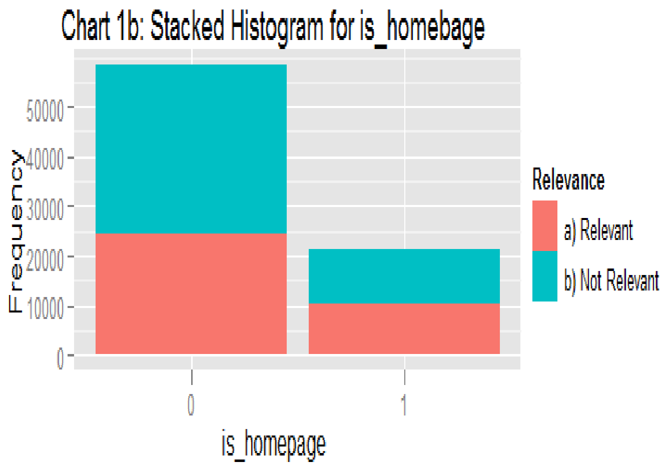
1. Introduction

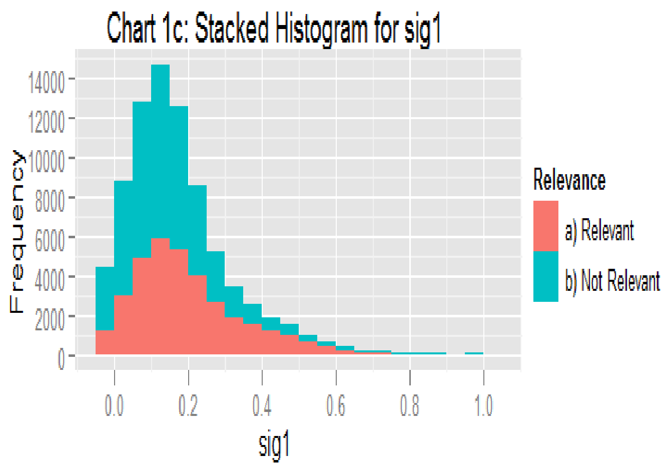
The goal of this project is to build a model to predict whether a search engine query is relevant. The data provided is from a search engine query and contains URL information. The training set contains 80,046 observations while the testing set contains 30,001 observations. Ten attributes are provided on both datasets, along with a query_id and url_id. Query_id and url_id are additional attributes that uniquely identify an observation, but have not been deemed with any significance at the outset. The training set also contains a variable, relevance, which will be the attribute that the data mining model will try to predict.

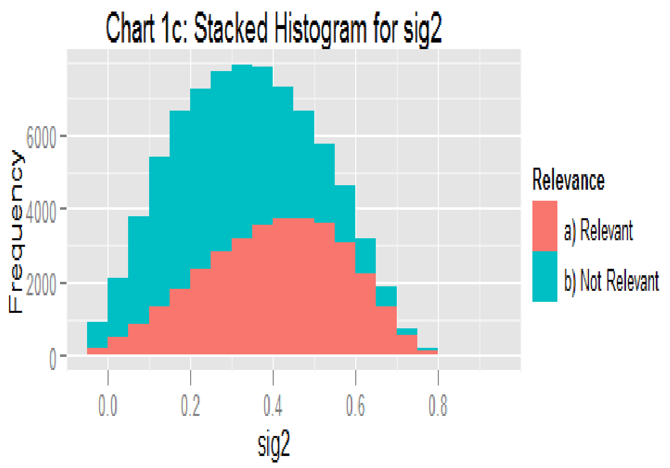
2. Data

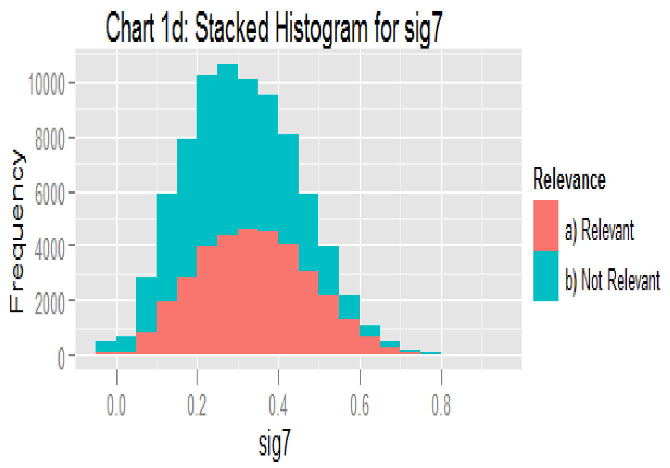
There are ten attributes that have been provided to predict relevance. In both datasets, all attributes in all observations are not missing and are not negative. The variable is_homepage appears to be a nominal variable as the only values it takes are 0 or 1. Query_length consists of integers, and the distribution is right skewed with most observations binning to the value of 1 or 2. This suggests the variable is a quantitative discrete attribute. The variables sig1, sig2, sig7, and sig8 all take values between 0 and 1. These variables appear to be quantitative continuous variables, at least to the precision of two decimal places. I have produced a stacked histogram for these attributes in the Chart 1 series below using the ggplot2 package. Each of the numeric attributes shows roughly normal distributions. The stacked histograms help show which variables may be stronger predictors. For instance, sig2 seems to be one of the better variables for predictive purposes. The variables sig3, sig4, sig5, and sig6 appear at first to be somewhat unmanageable, skewing very far to the right. The Chart 2 series illustrates this. Almost all the data is concentrated in the first bar, but extends a long distance out. By transforming all values with the function f(x) = log(1+x), the distribution approaches something much closer to a normal distribution. The Chart 3 series show the transformation. Sig6 still doesn’t seem to have been helped in terms of transforming to a normal distribution, but the distribution seems less skewed than without it.











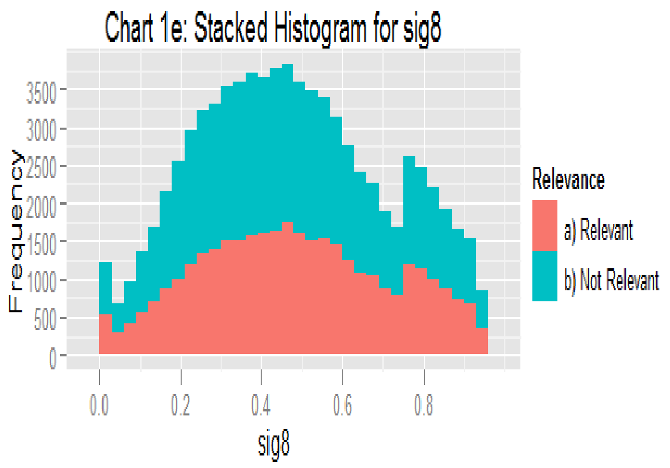


Chart 2a: Histogram for sig3

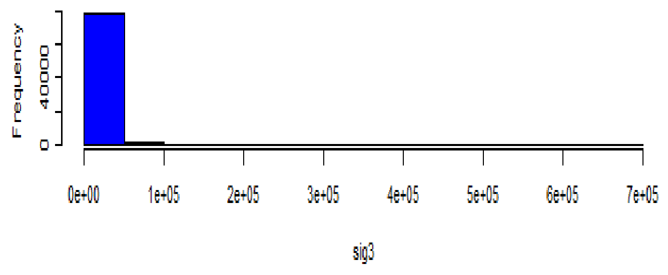


Chart 2b: Histogram for sig4

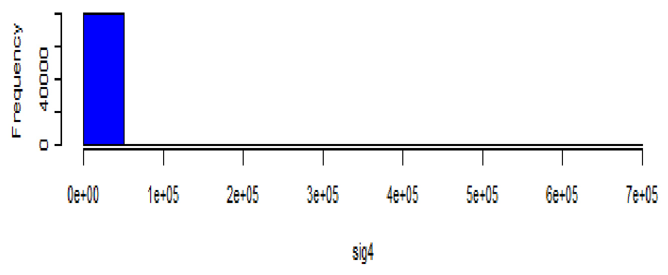


Chart 2c: Histogram for sig5

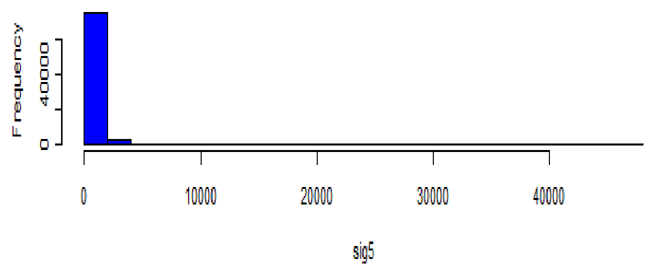


Chart 2d: Histogram for sig6

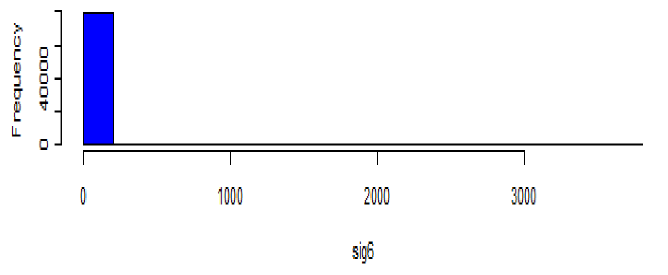


Chart 3a: Histogram for ln(sig3+1)

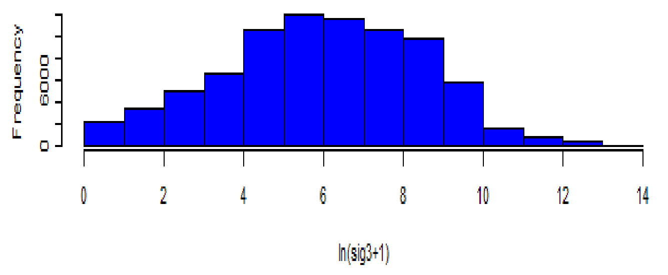


Chart 3b: Histogram for ln(sig4+1)

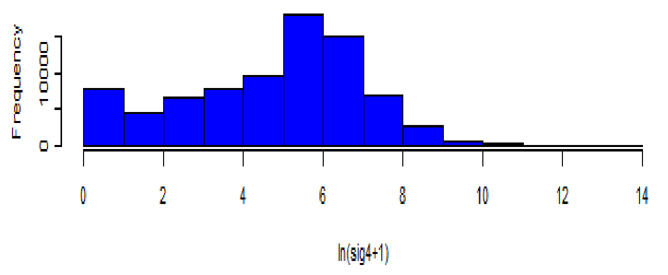


Chart 3c: Histogram for ln(sig5+1)

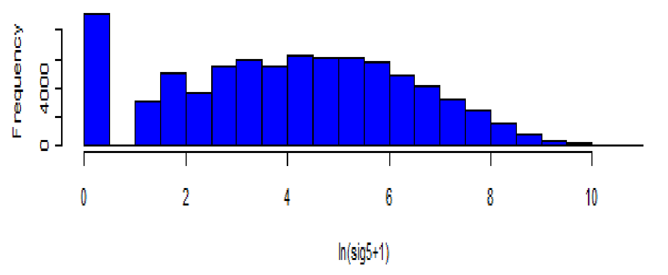
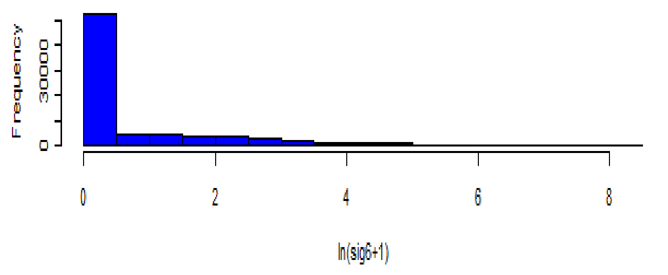
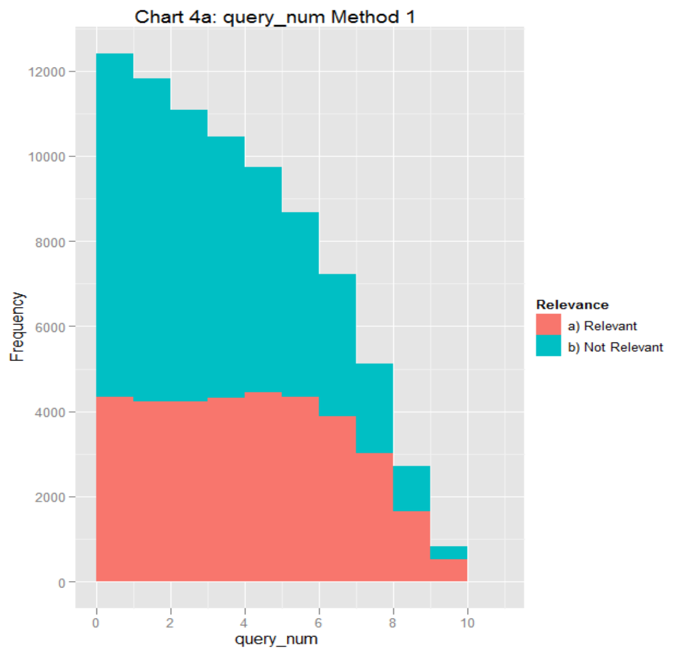
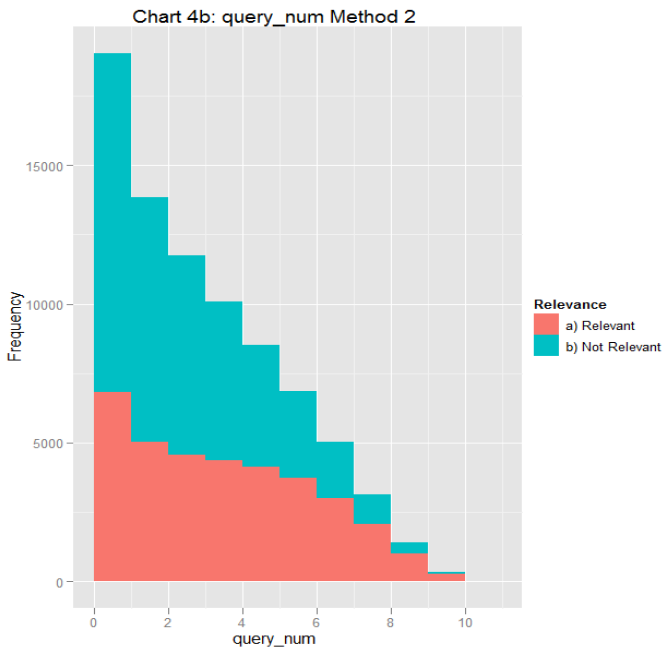


Chart 3d: Histogram for ln(sig6+1)



As mentioned above, the data provided is unique by query_id and url_id. The url_id tends to have sequential sequences within a query_id, which may indicate that those are sequential queries, thus may have some predictive value. I attempted two methods of leveraging url_id and query_id. First, I ordered the data by query_id and url_id, then I created a new variable that that is a sequential count, resetting back to 1 at each new url_id. Second, I ordered the data in the same manner, but reset the value to 1 whenever the previous observation of the url_id was more than one value away from the current observation or if it was a new query_id. The Chart 4a and 4b show a stacked histogram of the two methods just described above respectively. Chart 4b presents much more normal looking distributions, but both seem to have some predictive power. It isn't necessarily clear which method is better just from the graph. Testing the two variables of a few models, both variables produced improved results, but the second method produced lower testing and training error so the second method was used.





3. Evaluation

I have considered five models for trying to predict whether a query is relevant. To validate my models, I have split up the initial training set into a training set and a testing set, not to be confused with the final testing set provided. I assigned a random number with a uniform distribution over 0 and 1 to each query_id. If the random number was below .75, all observations associated with that query_id were binned to the training set, otherwise the observations were binned to the testing set. The data was split at the query_id level so that the full set of url_ids could be leveraged when building a model. Table 1 illustrates the partitioning of the training set into the working training and testing set, and illustrate that their make-up of relevance is similar.

Table 1: Make Up of Relevance

	Total		Train		Test	
Relevance	**Count**	**Percent**	**Count**	**Percent**	**Count**	**Percent**
Total	80,046	100.0%	60,027	100.0%	20,019	100.0%
1	34,987	43.7%	26,328	43.9%	8,659	43.3%
0	45,059	56.3%	33,699	56.1%	11,360	56.7%

4. Model Selection

I tested five models to determine which one had the best fit. I consider the ten attributes provided as well as the constructed variable query_num. Additional variables could be explored, such as the max value across a query_id. Also ideally models should be explored where variables are dropped, as correlated values may just create noise in the model. For instance the Naïve Bayes model will suffer significantly if using correlated variables. I did not explore these steps due to time constraints. Table 2 gives the error rates for each of the models after determining the best fit for that model.

Table 2: Model Error

Model	Training Error	Testing Error
Naïve Bayes Model	36.9%	36.5%
K-Nearest Neighbor (k=105)	32.7%	33.2%
Random Forest	0.0%	32.8%
AdaBoost	33.9%	33.7%
Support Vector Machine (cost=2)	31.9%	32.6%

a) Naïve Bayes

Naïve Bayes is a simple and quick model, but carries with it the unlikely assumption that all attributes are conditionally independent given the class variable. I wrote my own implementation of the naïve Bayes model. I only treated the variable is_homepage as a factor, the rest of the attributes I considered numeric. For is_homepage, I calculated the conditional probability that is_homepage is 1 or 0 given relevance. For the rest of the attributes I assumed that the conditional probability of the variable in relation to relevance is a normal distribution. To find the prediction of relevance, I calculated the probability that the combination of values would be relevant or not relevant. I then classified the observation based on whichever value was larger. As can be seen above, many of the attributes have nearly normal distributions. Interestingly, the test error is lower than the training error. This would seem to imply that this model is robust, but not necessarily capable of representing the full complexity of the model. Regardless the testing and training error is much higher than the other models considered.

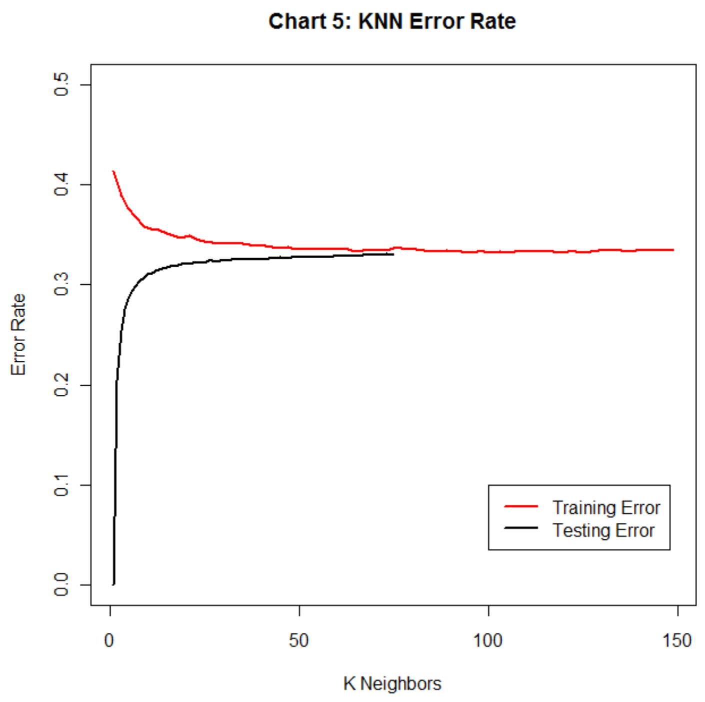
b) K-Nearest Neighbor

K-nearest neighbor, at a given point, counts the number of cases where relevance is 1 and the cases where relevance is 0. It then assigns the value with a higher frequency to that point. In cases where there are ties, multiple methods can be considered to break the tie. For instance, the model can consider the nearest point to break the tie. The knn() model from the class package, which is the model I used, will randomly pick a value for ties. Since it is classifying values based on proximity to other points, K-nearest neighbor is very sensitive to distance. For example, consider two variables, x_1 and x_2. Both have normal distribution and their means are 0, but x_2's standard deviation is twice that of x_1. Then k-nearest neighbor will essentially weight the distance between two values in x_2 more than values in x_1, even if their scale relative to each other were the same. To get around this, I normalized all the attributes using the transformation f(x)=(x-min(x))/max(x) based on the values in the training set. This will transform the maximum value to 1, the minimum value to 0, and the rest of the values in between. Running a few test models, this did seem to improve the error in both the training and testing set. Table 3 presents the error rate with and without scaling all the variables at k=105.

Table 3: KNN with\without Normalization of Attributes - K=105

Description	Training Error	Testing Error
Attributes Normalized	32.7%	33.2%
Attributes Not Normalized	35.1%	35.6%

It isn't clear before running the model what value k should take. I ran the model for all odd values between 1 and 149. As expected, the training error for k=1 is 0. The testing error for k=1 is fairly high at 41.3%, illustrating a clear case of over fit. We see the testing and training error converge fairly quickly. The minimum testing error occurs at k=105, with an error rate of 33.2%. Chart 5 presents a graph showing the trends of the training and testing error as the value of k increases.



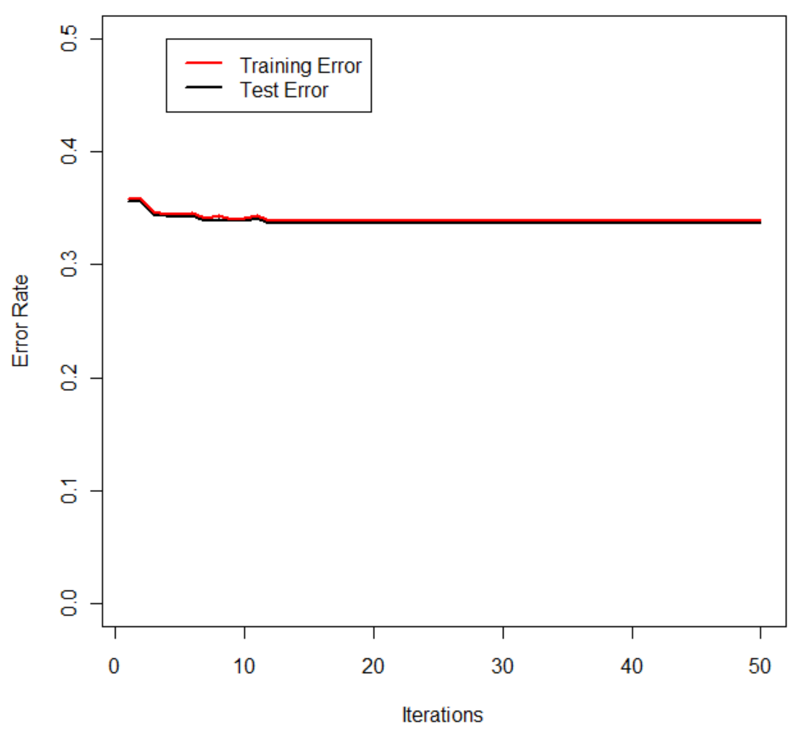
c) Random Forest

Random forest is a popular ensemble classification method. It randomly chooses a subset of the available attributes, and runs a decision tree classifier. This is done over some predetermined set of iterations, and the combinations of all the decision tree classifiers are considered when determining the predicted value. I used the randomForest() model from the randomForest package. It defaulted to using 500 decision trees and considering three variables at every split. The training error is 0, and the testing error is a decent 33.0% error rate.

d) AdaBoost

AdoBoost is a popular implementation of the boosting ensemble classifier. Boosting in general works by applying a classification method to the data, then reweighting the training data to emphasize incorrect predictions so that the model can emphasize those observations on the following iterations. Decision trees are the most popular weak classifier to use with boosting, and it is what I used when applying the AdaBoost model. Specifically I used the rpart() function as used in class. My AdaBoost model did not behave as expected. The training error never dropped below 33%, and exponential gain did not see any decrease. Chart 6 shows the error rate over fifty iterations. The testing error is actually slightly lower than the training error, but doesn't fluctuate. If I apply the condition that if the individual classifier error rate is greater than 50%, then reset the weights, the training error will have bumps every few iterations, but essentially stays the same.

Chart 6: AdaBoost Error Rate



e) SVM

Support vector machine (SVM) attempts to choose a hyperplane with the largest margin between the prediction classifiers in the training set, margin being defined as the distance between two parallel lines where each line partitions off one set of the prediction value. I used the svm() function which is part of the e1071 package in R to run this model. Running the default model, which sets scale to TRUE and cost to 1, my testing error performed better than the rest of the models. Setting scale to FALSE made the error rates worse. Support vector machine, like k-nearest neighbor, is sensitive to distance between the attributes. An attribute that spans a large distance may overly contribute the model. I tried scaling the data as I did in the knn model, but that failed to improve the model. It would seem the scale option considers other transformations of the data besides normalization. I also tried tweaking the cost parameter. If the svm model cannot strictly partition the predictor attribute in the model, then a slack variable can be introduced. The cost parameter tells the model how lenient it should be in allowing mixing between the classes of the attributes when determining the boundary. Increasing the cost should lower the training error, but at the cost of possibly over fitting. Table 4 shows the different cost values I tested and the model error. The model with a cost value of 2 performed the best. This was the model chosen for the final testing set submission. I also tried using the ksvm() function from the kernlab package. I tested the default model and a model where I set the cost parameter C to 2. They performed essentially the same, but had slightly higher error. Given more time, I would have liked to explore some of the kernel methods and some of the other parameters to see if I could improve my model.

Table 4: SVM Error Rates

Cost	Training Error	Testing Error
5	31.4%	32.8%
4	31.6%	32.7%
3	31.7%	32.7%
2	31.9%	32.6%
1	32.2%	32.7%
0.5	32.5%	32.8%