David Lin and Ben Wagle

Professor David Sontag

Machine Learning

15 December 2013

Sporting Event Predictor

**Introduction**

People love watching sports, and they love trying to predict the outcome of a competition between two teams. The records and statistics for sporting events are routinely kept and it is common practice for people to attempt to predict the victor of a competition based on these records of past performances. However, when people make predictions, they often introduce other factors like personal bias and intuition, which are not necessarily correct or quantifiable.

Using machine learning algorithms, we aim to find methods for accurately guessing outcomes of sports events. Using purely statistical data without introducing any outside bias, we hope to correctly classify games as home team wins or losses. Success in this project could revolutionize not just the world of bookkeeping and sports gambling, but also the study of sports in a more general way. We could realize the more broad nature of sport and the similarities between different athletic competitions humans have invented.

Before we began, we expected a correct classification rate between 60 and 70 percent. As a baseline, we found that always predicting a home team win resulted in roughly 59 percent success and only classifying based on win percentage resulted in 60 - 61 percent success. Using two linear and two nonlinear classification methods, we achieved results between 59.6 and 65.1 percent correctly classified games when tested on 1230 previously unseen data points.

We believe these results can be improved and expanded to other sports and possibly cross-sport predictions, but ultimately, sports have an inherent unpredictability that cannot be circumvented.

**Related Work**

We looked at a project written by Stanford University students Richard Harris, Allan Joshua, and Justin Sirignano. The project aimed to predict the outcome of baseball games.

While we did not reference this paper when working on our project, we find it worth mentioning as explanation for decisions we made when formalizing the problem we decided to solve. In their paper, the authors state that baseball was a good sport to classify because it has the most statistical data, as well as the most number of games per season for any major sport.

While this is true, we found baseball to be too specific of a sport to use. What we hope to ultimately achieve is a generalizable methodology that can be applied to a range of sports, as well as mixing of sports. However, baseball does not have a traditional offense, defense, or scoring structure. Thus, we found that other sports which all have similarities (basketball, football, soccer), would not translate well when including baseball statistics. While there is no reason that feature vectors of baseball statistics would not be applicable to our algorithms, they would not be useful in cross-sport classification. Furthermore, what we have developed is capable of predicting any sport given a feature vector of statistics, but the referenced baseball predictor is only applicable to baseball due to the specificity of the types of statistics that exist in the sport.

**Task**

Our goal is to utilize averaged statistical data of NBA teams to predict the outcome of games with respect to the home team. Using data for NBA games taken from ESPN, we compute teams’ average game statistics, such as steals, points scored, and assists. These averaged statistics are used to construct our feature vectors, which we use as the input to our algorithms. The algorithms output a prediction, which designates either a home team win or loss. Each game is considered as a single datapoint and its feature vector consists of the concatenation of the home team’s and away team’s average statistics up to, but not including, the current game.

Gamei = [AvgStealsHome, AvgPointsHome, … , AvgFoulsHome, AvgStealsAway, AvgPointsAawy, … , AvgFoulsAway]

These values are normalized between 0 and 1, and then the away team’s statistics are made negative. The label for each datapoint is either 1 for a home team win or -1 for a home team loss.

**Algorithms**

Our approach to this problem was to test variations of both linear and nonlinear classifiers. We began with the linear classifiers to build upon our initial baseline. We planned on seeing marginal improvements in prediction, but were mainly interested in the specific values of the weight vector used in prediction. After optimization of the linear classifiers, we moved to nonlinear classifiers in an attempt to improve our prediction success even more. After optimization of the kernel and decision tree, we saw an average of up to 5 percent increase in the prediction success when compared to the baseline.

With regards to assumptions about the data, we were sure that the data would not be linearly separable, and therefore the linear classifiers would quickly converge at some level or error. Below, we go into more detail regarding each classification algorithm and the process of optimization it entailed.

Perceptron

Using a weight vector with values initialized to 1, each feature vector (game) is dotted with the weight vector. Because the away team’s stats are negative, this dot product results in the difference between the home team stats and away team stats. Using the sign of this dot product, we predict home win if positive, and home loss if negative. If the prediction does not match the actual result, the weight vector is updated using the feature vector with its values flipped according to a scale by the correct label. The labels are 1 or -1, so this scale just affects the sign of the feature vector values. This new feature vector is then added to the current weight vector to update the classifier.

dotProd=0

for i in range(len(feat)):

dotProd+= float(feat[i])\*weights[i]

#set prediction

if dotProd >=0:

predict=1

else:

predict=-1

#check prediction

if(predict != int(labels[k])):

hasError=True

errors+=1

for j in range(len(feat)):

weights[j]= weights[j]+(float(feat[j])\*int(labels[k]))

For testing, we use the classifier obtained from training in the same manner, dotting the weight vector with the feature vectors in our validation and test sets.

errors=0

for k in range(len(data)):

feat=data[k]

dotProd=0

for i in range(len(feat)):

dotProd+= float(feat[i])\*w[i]

if dotProd >=0:

predict=1

else:

predict=-1

if(predict != int(labels[k])):

errors+=1

return float(errors)

The parameters of the model that we chose to modify were the size of the training set and the number of iterations through the data. The training set size was useful in studying the effect of VC dimension on our ability to effectively train and predict. Altering the number of iterations allowed for us to prevent overfitting, as well as preventing the program from run indefinitely.

PEGASOS

For PEGASOS, weights are initialized to 0. For several values of lambda, we optimize the SVM objective through several iterations, using a decreasing step variable, eta, to use stochastic gradient descent to find the optimal weight vector. If the current weight does not correctly classify a datapoint correctly, it is updated to a weight which gives better performance in the next pass through data.

for lambda\_exp = lambda\_exp\_min:lambda\_exp\_max

w = zeros(size(x, 2), 1);

u = zeros(size(x, 2), 1);

t = 0;

lambda = 2^lambda\_exp;

for iter = 1:20

for j = 1:size(train\_features, 1)

t = t+1;

eta = 1/(t\*lambda);

if train\_labels(j) \* dot(w,train\_features(j,:)) < 1

u = ((1 - (eta\*lambda))\*w) + transpose(eta\*train\_labels(j)\*train\_features(j,:));

else

u = (1 - (eta\*lambda))\*w;

term = ((1/sqrt(lambda))/norm(u));

if term < 1

w = term\*u;

else

w = u;

Like with the Perceptron algorithm, we varied the size of the training set to show that with our low number of features, the VC dimension hinders our ability to take advantage of a larger training set.

Gaussian Kernel

Using LIBSVM and its MATLAB interface, we used a Gaussian (radial basis function) kernel:



The kernel gives a notion of similarity between two data points as a measure of distance between 0 (very distant) and 1 (very close).

Using our training set, we varied the gamma and cost parameters to find optimum values which resulted in a model which gave the maximum classification rate on our validation set.

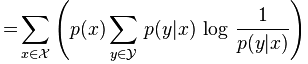
Decision Trees

Rather than compute the information gain by hand at each level of the tree, we utilized the scikit-learn Python module. Not only did this allow us to quickly fit a decision tree, but also output a visualization of the resulting tree classifier. The main calculations inherent in using this module were computing which feature to split on at each node in the tree. The two methods available as a parameter to the model were Gini Impurity and Entropy (information gain).

Gini Impurity: computed by summing the probability of selecting each game multiplied with the probability of incorrectly classifying that game:



Entropy: computed by summing the probability of each feature value multiplied with the sum of the probability of selecting a game given that feature value multiplied with the log of that probability.



We also varied the depth of the tree and the size of the training set in order to obtain an optimal classifier. Varying the tree depth allowed us to improve the generalization of our resulting tree. Increasing the amount of training data used to construct the tree resulted in a general improvement in the prediction score on our validation and train data.

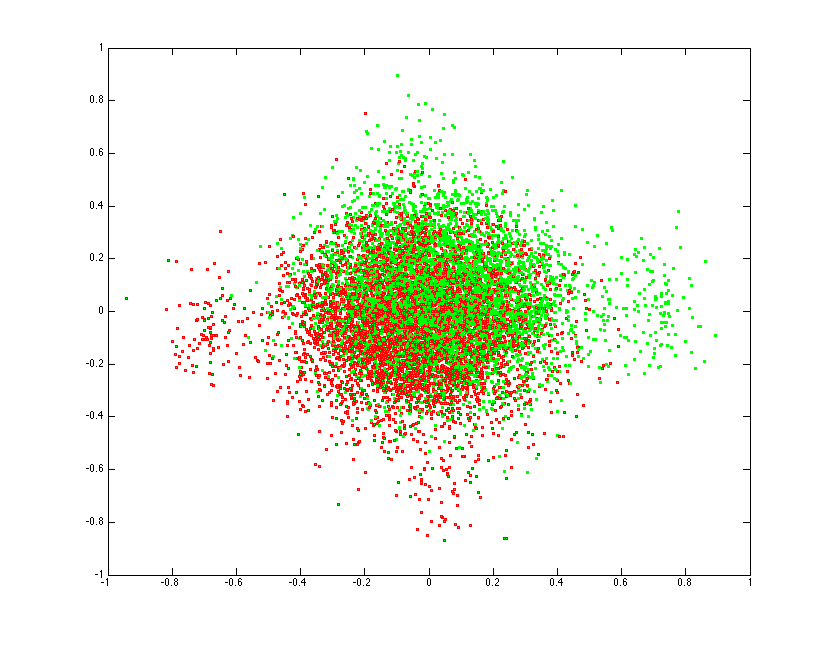
**Data**

ESPN has data for ten previous NBA seasons with the statistics for each game recorded. First we compiled all the raw statistical data available for regular season games from 2002 to 2013. For each season, for each team, for each game, we computed the teams’ average statistics up to, but not including the current game. Team A’s fifth game statistics were the average of the statistics from games 1, 2, 3, and 4 of the same season. Then to predict game 5, we used those average statistics from the first four games for team A, as well as the opponent's averages in the same manner, to build that game’s feature vector.

To preprocess data, we computed all the averages before each game for every team. We then normalized all of the averaged statistical values to values between 0 and 1. We then concatenated the home team’s averages with the negatives of the away team’s averages to create our feature vector for a game.

Each season’s averages were computed separately because we assumed a team’s performance between seasons could change based on factors like personnel changes and injuries, and therefore we believed a new season should be given a new record.

We also adjusted the construction of feature vectors in an attempt to improve the prediction success of the non-linear classifiers. Because dot products are not utilized in the fitting of the non-linear classifiers, we converted our 32-dimensional feature vectors into 16-dimensional vectors with each entry being the difference between the corresponding home team statistic and away team statistic. This however, did not yield impressive results and was much worse when compared to the results achieved using the 32-dimensional vectors. We attribute this partly to the reduced size of the feature vectors, as well as the fact that considering the actual teams’ statistics is more useful than considering the difference in statistics (i.e. knowing that the home team has a win percentage of 0.567 and the away team has a win percentage of 0.781 is more useful than just using the value -0.214).

To visualize the data points, we used singular value decomposition and plotted the first two principal components from our training points.

In the middle of the plot, green points (home team wins) are well-mixed with red points (home team losses) and we can see that they must be difficult to separate. However, further away at the edges we can see that the red and green are much more separated, indicating to us that certain extreme combinations of statistics may always lead to a more confident prediction.

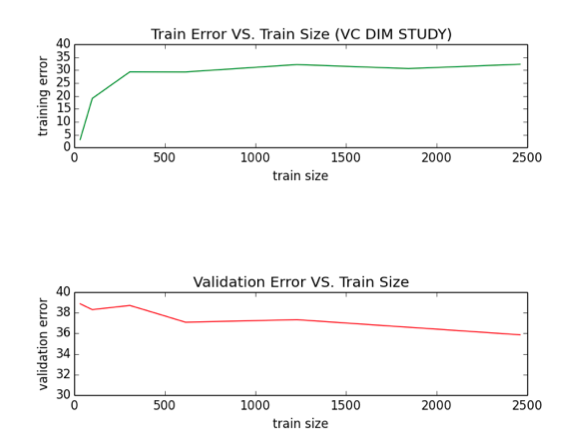
**Methodology**

We hypothesized that we would be able to predict 60-70 percent of the unseen data correctly. A correct classification is considered to be a correct 1 or -1 output, corresponding to a win or loss for the home team of a game. These classifications were then compared with the true values taken from ESPN. Error is calculated by finding the fraction of predictions that do not match the true outcome of the respective game. With ten season available, we used the first eight seasons as our training set, the ninth as our validation set, and the tenth as our unseen test set.

**Results**

Perceptron

The major takeaway from our Perceptron results is the firsthand insight into VC dimension. Our results show that the training error jumps to oddly high values as the size of the training set grows. Additionally, the training error and validation error become increasingly similar. We attribute this to VC dimension of the hypothesis class being utilized. As a linear classifier, Perceptron has a VC dimension of d+1 where d is the dimension of the feature vector. Our feature vector is only 32 dimensional, thus, as our training set grew on the order of thousands, it made complete sense for the error to increase and approach the validation error. When validating, we always used the season following the final season in our training set as our validation set.

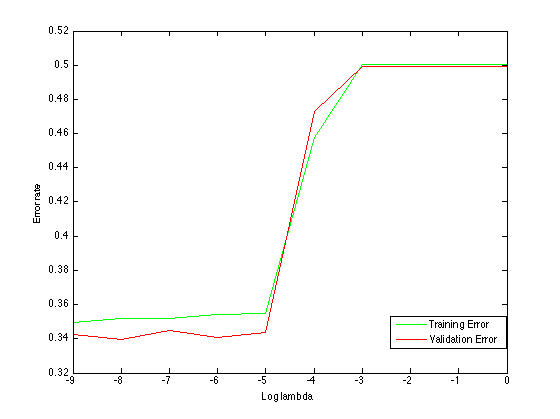


Eventually, we expanded our data set from just the 2007-2011 seasons to include all seasons from 2002-2013 (excluding the shortened 2011-12 season). Using cross validation, we found the optimal season size to be 9,731 games and the optimal number of iterations through the data to be 7. At these values, Perceptron yielded the lowest validation error (33.7%). Again, the VC dimension comes into play, causing our training error to become worse than the validation error.

At the optimized values of 9,731 games and 7 iterations, Perceptron had 59.6% success in predicting games, just barely better than our baseline, although the visualization of the data (page 5) shows that a linear classifier should work about as well as just guessing. 

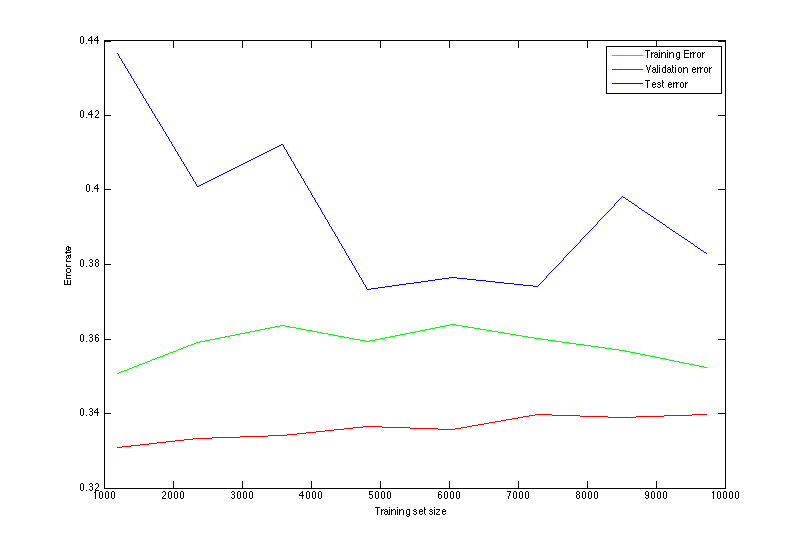
PEGASOS

PEGASOS ran slightly better than Perceptron. Ranging lambda between 2-9 and 20, training error remained around 35% until we chose lambda = 2-4, at which point the error rate converged toward 50%.



With lambda = 2-8 and 30 iterations through the training data, we received a test set correct classification rate of 61.63%.

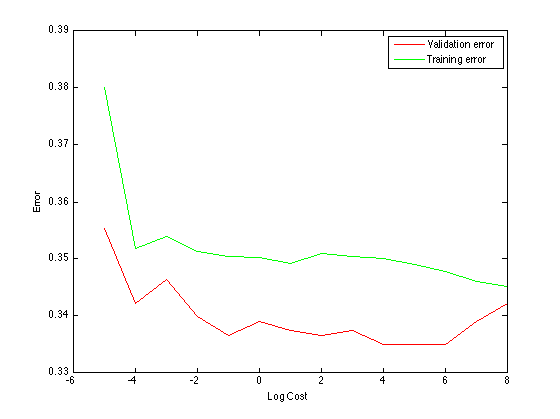
As with Perceptron, we also varied the size of the training set in an attempt to optimize further.



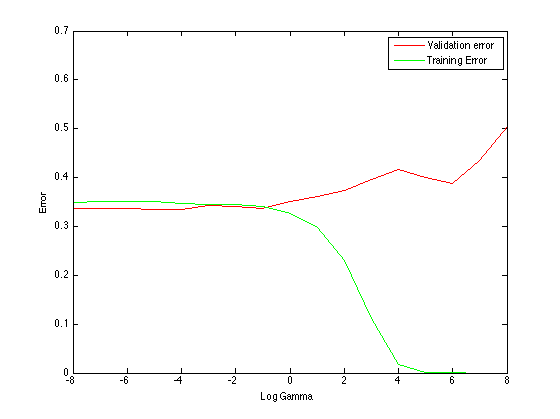
The lowest validation error occurred with a training set size of 1189, however the test set score at that point was not at the same caliber of accuracy. In fact the best test performance occurred with a training set size of 4820, but the validation performance gave no indication of reaching a local optima at that point.

Gaussian Kernel

First, varying the cost variable in the kernel between 2-5 and 28, and using the validation set to evaluate performance, we found the best cost to be 24, at which the validation error was 33.50%.



After setting the cost variable to 24, we also varied gamma to try to optimize further. We varied gamma between 2-8 and 28 to find the best value, again evaluating with our validation set. We found the best gamma to be 2-4, which gave a validation error of 33.42%.

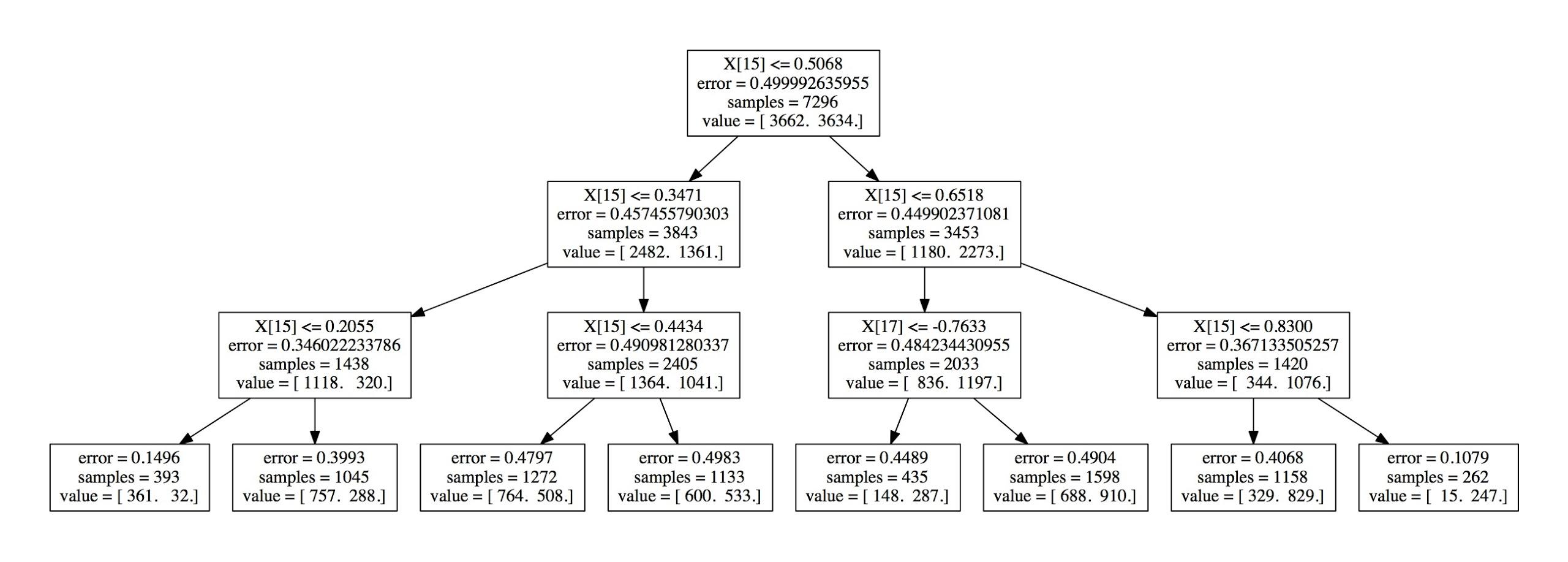


Interestingly, the validation and training errors seemed to be stable around 33% before training error decreased while validation error increased. Using cost = 24 and gamma = 2-4, the model predicted 62.92% of games correctly from the test set.

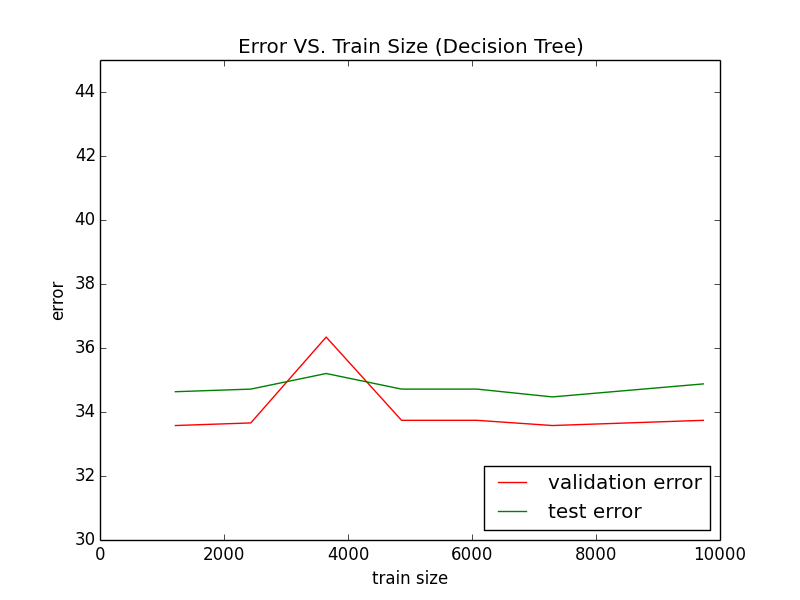
Decision Trees

To obtain optimized results for the decision tree, we varied both the training set size and the maximum depth of the tree. Using cross validation, we found that 7,296 games and a depth of 3 resulted in the lowest validation error (33.6%). In optimizing, we aimed to minimize validation error while also maintaining a generalizable classifier. We found that as the depth of the tree grew, the model began to over fit to our training data. For reference, we include as an attachment a visualization of the tree fit on 7,296 games without restriction on the depth. We also include a visualization of the optimal tree classifier fit on the previously mentioned parameters.

Tree using 7,296 games and max\_depth 3 (enlarged version in attached PDF):



Our overall results when training on sets ranging from 1 season to 8 seasons:

Our final test set classification rates for each algorithm:

Perceptron: 59.6%

PEGASOS: 61.63%

Gaussian kernel: 62.9%

Decision trees: 65.1%

**Discussion**

Our final test set classification rates matched our hypothesis of predicting 60-70 percent of games correctly and increased in the order that we expected according to the algorithm used, though we expected that the nonlinear classifiers would perform better than they did.

As stated previously, the low VC dimension of a hypothesis class of linear classifiers explains the high training error and unimpressive performance of our linear classifiers. Additionally, we acknowledge that the distribution of our data does not lend itself to linear classifiers (not linearly separable). We overcome this problem by using non-linear classifiers, which ultimately yielded better results.

**Conclusion**

While our results were slightly better than the ~60 percent baseline discussed in our introduction, there is still much room for improvement. Our main shortcomings were the amount of data and dimension of the feature vectors we used. We think if we were to add more features, such as the head to head record between teams, player statistics, and statistics kept in other databases (ex. NBA.com), we would be able to achieve even better results. It would not take long to compile and process this extra data.

With regards to the future, our algorithms and methodology should be just as applicable if we replace the NBA feature vectors with ones from the NFL, Barclays Premier League, NHL, etc. This is something we plan to do in the near future, as soon as we can compile and process the necessary data from each of the respective leagues. Ultimately, our algorithms and process should prove to yield an impressive classifier for any sports.

It would also be interesting to take the classifier for one sport and directly apply it to the statistics of another sport. For example, using a linear classifier trained on NBA games, we would take the resulting weight vector and apply it to a soccer team’s average statistics, statistics of the same type as used in the NBA feature vectors (i.e. steals, shots, fouls, etc.), to calculate predictions for soccer matches. Finding the existence of a “universal” classifier for all sports could potentially show that all sports are much more similar than we believe them to be, and give us new insight into the nature of sport and athletic competition.

**Bibliography**

1. Chang, Chih-Chung and Lin Chih-Jen (2013) *LIBSVM* (Version 3.17) [Computer program].

Available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>

2. Harris, Richard, Allan Joshua, and Justin Sirignano. *Predicting the Outcomes of Baseball*

*Games*. Rep. Stanford University, 2011. Web. Available at

<http://cs229.stanford.edu/proj2011/HarrisJoshuaSirignano-Predicting%20the%20Outcomes%20of%20Baseball%20Games.pdf>

3. Pedregosa et al. (2013) *Scikit-learn: Machine Learning in Python* (Version 0.14) [Computer

program]. Available at <http://scikit-learn.org/stable/index.html>

4. Sontag, David (2013) *visualize.m* [Source code]. Available at

<http://cs.nyu.edu/~dsontag/courses/ml13/assignments/ps6_data.zip>