# NCAA Basketball Playoff Prediction:

Introduction to Problem:

The problem we are trying to solve using machine learning is to predict the results of an NCAA playoff bracket using data from playoffs of previous years. Our method of predicting is done one game at a time using features that define the skill of each of the two teams facing each other. More on what features we chose is defined in the “Feature Selection” section. The data we are using is organized into previous seasons labeled as A-R, and it contains data on regular season and playoff wins, losses, teams faced, etc. We observed that the best type of machine learning to use is supervised learning, since we have all the data on previous playoffs and their outcomes (the targets) to train on. To start, we used the basic perceptron as our model for predicting, and then we moved to a regression model using mini-batch gradient descent. In both models, we iteratively added various machine learning techniques in order to achieve better prediction results.

Feature Selection:

Feature selection was done largely by considering what logically would be a predictor of whether a team will win a game. When examining two different teams, the main indicator of skill available to us was their regular season win percentages on a per-season basis. As teams can greatly vary from season to season, there is very little correlation from one season to the next.

The next feature we chose was a team’s seeding in the playoffs. While a team’s seeding is not always an accurate predictor, it is still a useful indicator of how a team will perform in the playoffs.

The order of features was set up so the first couple features are the win percentage and seeding of the team we are predicting the outcome of the game for. Following the first team’s features are the features for the team they are facing. The target when training was either a ‘1’ representing that the first team won, or a ‘0’ representing that the first team lost (second team won).

Later in the project, we added a feature for each team that represented the average win percentage of all the teams a particular team faced in the regular season. The purpose of this feature was to capture how difficult of a schedule a team had, which might explain away why a team did not perform as well in regular season compared to a team that they are better than.

Final feature-target setup:

(win-%, seed, opponent-win-%, win-%, seed, opponent-win-%), (team1’s-outcome)

Team 1 Team 2

Model Creation:

Base Perceptron (Learn rate = 0.1, Training iterations = 1000): The first model we used was a basic perceptron and we ran seasons A-Q of the data through it to train the perceptron. From there, we randomized the remaining season R and used that as the testing data. After several runs of training and testing with this data, the average and median accuracy was 67- 68%. Here is an example output of one run:

Run with season R as the test season

Confusion Matrix:

[[ 17. 9.]

[ 12. 29.]]

Accuracy:

0.686567164179

K-Fold Cross-Validation: Next, we added k-fold cross-validation to the model, where the k-chunks are the season A-R. We take turns using each season as the testing data and the rest of the seasons are used to train the model. After each season has been used as testing data, the accuracy of the perceptron is measured as the average over all the accuracies of each model created on the k-fold cross-validation. A portion of the output from the cross-validation is given below:

Run with season A as the test season

Confusion Matrix:

[[ 17. 3.]

[ 14. 29.]]

Accuracy:

0.730158730159

Run with season B as the test season

Confusion Matrix:

[[ 24. 5.]

[ 13. 21.]]

Accuracy:

0.714285714286

...

Run with season R as the test season

Confusion Matrix:

[[ 33. 34.]

[ 0. 0.]]

Accuracy:

0.492537313433

Average accuracy over the k-fold cross-validation:

0.639195309827

This gives us a true accuracy of 64% which is lower than the 67-68% we got before we implemented cross-validation. Something noteworthy of this output is the run with season R as the test, which has an accuracy of 50%. A closer look at the confusion matrix reveals that the model is over fitting and predicting only one outcome for all games. We concluded that the perceptron isn’t very consistent from run to run, and in conjunction with a fairly low accuracy, is not a very good model to use. This is expected as the perceptron is one of the simplest and least flexible supervised learning models.

Permutation Test: The next feature we implemented was a permutation test. This was done in order to check if the perceptron can perform better than randomized data. This calculation takes a considerable amount of time, so we only run through 1000 iterations of the permutation test. We first train and test the perceptron normally and save the accuracy of that model. Then for each iteration, we permute the target column of the data, and train and test the perceptron. We keep track of every permuted model that performs better than the first one we ran and calculate the p-value. The output with season C as the testing data is given below:

Run with season C as the test season

Confusion Matrix:

[[ 30. 22.]

[ 1. 10.]]

Accuracy:

0.634920634921

Running permutation test and calculating p-value...

p-value = 0.001

For this specific run of the permutation test, we got a p-value of 0.001, which shows that the perceptron is performing better than random. We also checked the accuracy over multiple runs of the permutations, and they were all around 50% which is to be expected with randomized outcomes.

Regression Model (Learn rate = 0.1, Train iterations = 1000): Describe here.

Decreased Learning Rate Over Time (Init Learn rate = 0.1, Train iterations = 10000): Describe here.

Added Opponent Win Percentage Feature: Describe here.

# Loan Default Prediction:

Introduction to Problem:

Enter intro to problem here.

Feature Selection:

For this, given the large number of features given to us in the training data, we went through and found ones that would most likely have no impact (all being the same number or similar) and had gigantic gaps in values. From there, we went through and did some preliminary tests on various features in order to find the features that had the largest possible impact. We did not want to have too many features so we picked ones that seemed to affect the outcome on the loss. Due to this fact some human error may have been added (as we decided which features to use instead of letting a computer decide which features were most important).

Model Creation:

The first draft of the model involved splitting the training data into two subsets: no loss, and total loss. For each sub set the extremes (top and bottom 5%) were removed and the remainder of data was averaged into an average data point. We did not use some of the data in the training data (to be used for a testing set). A data point would be compared to each average, for each of the roughly 800 features; if it was closer to the no loss it received one point in that feature, if it was closer to the complete loss it lost one point for that feature. If the final tally was above zero, predict no loss; if it were below zero predict complete loss.

The final model used a modified version of the perceptron code used by the NCAA prediction. Surprisingly, (after over an hour of running the code) it ended up being more accurate than NCAA prediction.

1. Ignoring the percentage of loss, the algorithm correctly predicted whether the loan would default or not sixty percent of the time. The numbers produced [-400, 400] seemed to be random and could not be used to determine the percentage of default. Using this model also crashed our computers multiple times (and we have a Haswell processor with 16 GBs of RAM).
2. A lot of the features seem random and appear to not have any correlation with the amount of default. Using less features would make the algorithm faster (and easier to run) but (currently) we do not know of a good way of eliminating features with unsupervised learning.