Alex Anderson, Nicholas Stuter, Kristopher Wagner
CMPSC 448

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:

$\vec{x}$(win-%, seed, opponent-win-%, win-%, seed, opponent-win-%), t(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): Due to the complications of the perceptron and how simplistic it is, we moved on to try a regression model with the least squares error. We also chose our weight update method to be mini-batch gradient descent in order to make sure we converge to optimal weights in an efficient manner. The mini-batch gradient descent takes 10 records at a time and performs the gradient over them, and the calculated gradient is used to update the weights. A portion of the regression model output is given below:

```
Run with season A as the test season
Confusion Matrix:
[[ 26.   9.]
 [  5.  23.]]
Confusion Matrix Accuracy:
0.777777777778
Least-Square-Error Accuracy:
0.832415328601

...

Run with season R as the test season
Confusion Matrix:
[[ 27.  10.]
 [  9.  21.]]
Confusion Matrix Accuracy:
0.716417910448
Least-Square-Error Accuracy:
0.796424850628

Average accuracy over the k-fold cross-validation:
0.810340245518
```

We can see a big increase in the accuracy of the regression model by 17% more than the perceptron model. The accuracies of all instances of the regression models were also more consistent with the average accuracy than the instances of the perceptron models we were getting.

A note about the confusion matrix for the regression model: the purpose of it was to visualize what kind of predictions the model would give. Since our targets are either 0 or 1, we chose 0.5 as the threshold for predicting for the confusion matrix. Naturally, the confusion matrix suits the perceptron better and is not intended to be used with regression, so we see the accuracy is slightly off of the true accuracy that the least squares error gives.

Decreased Learning Rate Over Time (Init Learn rate = 0.1, Train iterations = 10,000): Next, we wanted to implement a learning rate that would decrease over time and have more iterations in order to achieve better convergence. The model is first trained for 1000 iterations with a learning rate of 0.1. Then, we switch to a learning rate that is calculated by $\frac{Tuning\ Constant}{Cur\ Iteration\ Count}$. We chose a tuning constant of 100, because when we switch the learning rate at 1000 iterations, the calculation gives a learning rate of 0.1 to start and will get smaller from there. Here is the output of the average accuracy using this additional feature with the regression model:

```
Average accuracy over the k-fold cross-validation:
0.81044139282
```

We see that the accuracy is still at 81%, so the decreasing learning rate does not make a meaningful impact to the convergence of the model. This could be due to the fact that the regression model already does a good job at converging due to the mini-bath gradient descent.

Added Opponent Win Percentage Feature: The last addition we made to the models was to add a feature to the data: the average win percentage of the teams that a specific team faced during regular season. This was discussed further in the “Feature Selection” section. The output of the average accuracy of the regression model after adding this feature is given below:

```
Average accuracy over the k-fold cross-validation:
0.810153002316
```

This addition also did not improve the accuracy of the model. Because it is the average of the opponents’ win percentage, all these values end up being within 10% of 50%. This small range in values could be making this feature insignificant to the model. Something to try in the future is to normalize this feature and see if that makes the data more meaningful.

Regularization Note: We considered implementing regularization into our regression model, but we saw that the weights were already very small and regularization would not be necessary. Here is an example of the weights for an instance of the regression model:

```
[[ 0.52807226]
 [-0.02832871]
 [ 0.36148606]
 [-0.02264244]
 [ 0.02464491]
 [-0.18858922]]
```

Alex Anderson, Nicholas Stuter, Kristopher Wagner 4/17/2014
CMPSC 448 Final Project Report

Loan Default Prediction:

Introduction to Problem:

The problem that we tried to solve, using machine learning, dealt with loan prediction. A bank supplied over a hundred-thousand individual summaries of the results of loans. These results were enumerated, with roughly eight hundred features and then the amount that the bank lost because the customer defaulted on their loan. The bank wanted an algorithm that given these features of someone applying for a loan, the algorithm would predict if they would default and how much they would default. The major challenges we faced involved feature selection, model selection, and the most important data cleaning. The data had missing information and had no labels. Cleaning the data and knowing what to normalize seemed impossible. There was also an error in the logic of the bank. The information they supplied was for people that were already approved to get a loan. They want to use this algorithm to decide if they should give someone a loan. The sample provided (people approved for a loan) does not match the population they want (people who may or may not be approved for a loan). Unless they combine this method with their previous method it will not work correctly.

Feature Selection:

For this, given the large number of features given to us in the training data, we went through and eliminated 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). Given more time we would have tested more features to see which ones increase accuracy.

Model Creation:

Initial Custom Model: 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.

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).

Perceptron: Because this problem requires supervised learning, we chose to use the perceptron created for the NCAA playoff predictions. Due to feature selection being so complicated and taking most of the work effort allocated to solving this problem, we simply used the final version of the NCAA perceptron and do not investigate further into model prediction. Here is a portion of the output for a run using the perceptron on the data:

```
Run with k[0] as the test group
Confusion Matrix:
[[ 8673.   834.]
 [   71.    11.]]
Accuracy:
0.90562102409

Run with k[1] as the test group
Confusion Matrix:
[[ 8675.  913.]
 [    0.    0.]]
Accuracy:
0.904776804339

...

Run with k[10] as the test group
Confusion Matrix:
[[ 8742.   837.]
 [    9.     1.]]
Accuracy:
0.911773907602
```

This output shows the accuracy at ~90% which is an excellent accuracy, but most likely this is not as good as it appears to be. A majority of the targets for the records fall under one classification, which causes the perceptron to overfit to that classification. This is demonstrated best in the run with k[1] as the test group, because the model predicts all records under one classification and still achieves a 90% accuracy. As always, the perceptron is not the most reliable model, which leaves a lot of room for further investigation into better models in the future like our implementation of the regression model.