**Final Project Report**

**Problem Overview**

In this project, we try to explore different Data Mining techniques to predict a winning entry for the National Basketball Association (NBA) Playoffs Bracket Challenge through analyzing data that includes player and team stats from different sources such as publicly available datasets and the NBA Stats official website.

The NBA Bracket Challenge is an annual challenge where players fill out their predictions for the NBA playoffs bracket for a chance to win 1 million US dollars and other prizes. For all rounds of the Playoffs, and for each game series in a round, players fill out their predictions for the series winner, including how long would the series go. A player who gets a perfect prediction wins $1m.

Our project aims to build a group of programs that would try to predict a Playoff bracket that wins this challenge based on data from previous NBA seasons.

**Objective**

The objective is clear; win $1m! To achieve that, the programs we build should be as accurate as possible, therefore, our objective is to develop a predictive model that is very accurate. We set an initial objective of developing a model with at least 80% accuracy.

**Plan**

We developed an iterative plan to understand how different parts of the data affect our model, the plan has three phases, each phase will have one model as a result:

Our first model will try to predict the outcome of a playoff game based on the regular season stats of the two teams playing in the game. The output will be whether the home team wins the game or not.

Since playoffs are played in a best of seven format, a shortcoming of this model is that predicting a game’s outcome for any number of times will always have the same result. We try to address this issue in the second model.

The second model introduces data that is external to the games. This is usually data that varies between games in a series. Examples of external data would be: time of game, or distance travelled before game. This is the model that will be used throughout this project.

An issue with this model is that it does not take into account player contribution to the game. Some players are really game changers, and their presence on the team can affect the team’s performance, which is why our third and final model will take that into account.

The third model will add player’s regular season stats for all players in a game to the input. This should add a lot of new features that might prove to be important to improve the accuracy of the model.

**Data**

We gathered data from multiple sources to account for all the different features that we want to implement in our models.

The first set we collected was the set of regular season team stats for all NBA teams since 1997. We wrote a Ruby script that queries the NBA Stats public API to collect the team stats. Unfortunately, the api only provided stats dating back to 1997.

The second set we needed was a set of all playoff games played since 1997. We wrote a Ruby script that scraped the website (Basketball Reference) to collect all playoff games since then. The set contains results of games as well as some external data about the games.

We integrated the first two sets by adding both home and away team stats to each playoff game. Each row in the resulting set contained columns relating to the playoff game itself, and columns that hold team stats for the home and away teams of that game.

The next step was to remove columns that we were not useful, e.g., team name, number of games played in the season, team score, and more. After removing such columns, we tried to infer useful features from existing columns. Examples of inferred columns are Days Since Last Game, and Opponent Wins Needed To Advance, which were inferred from Game Date, and the results of previous games in the series, respectively. Finally, we converted categorical features to dummy variables. For example, the feature Round had 4 values like “First Round”, and “Conference Finals”. We replaced the feature with 4 new boolean features that indicate whether that game belonged to that round or not (1 or 0).

The third dataset is a public Kaggle dataset that has all players’ season stats dating back to 1950. Stats like Points, Field Goals Made/Missed, 3Pts Made/Missed, Steals, Rebounds, Turnovers, and many more are available in this set. The one problem this set had, though, was that many of these stats had null values for the early seasons of the NBA, as not all stats were being tracked at the time.

Since the previous sets we collected contained data dating back to 1997, we had to eliminate a big part of this set.

One final set that we are yet to collect, is the team rosters for all playoff games since 1997. The website Basketball Reference has this data.

Combining the first two sets for our second model, we 1850 records had 108 features that describe each match. The label for each match was either a 1 or a 0; whether the home team won the game or not.

**Performance Metric**

Since maximizing the number of correct predictions is critical in our problem, we have used Accuracy as our performance metric for all the models. Accuracy is a very simple metric that allows us to understand the performance of our classification model by looking at the proportion of examples it has correctly predicted. The accuracy is always between 0 and 1, and a better performance is achieved for higher accuracy.

Where TP is True Positives, TN is True Negatives, FP is False Positives, and FN is False Negatives.

In simple terms; this is the number of correct predictions, divided by the total number of predictions made.

**Methods**

**Decision Tree and Random Forests**

The first model we implemented was a simple decision tree. To measure splits we used the entropy measure as it produced better accuracy than gini. To increase the model’s accuracy and avoid known issues with decision trees like overfitting and feature bias, we built a random forest. The random forest also used entropy to measure splits. We played around with different parameters. Our current model uses maximum leaf nodes as 5, minimum number of data points allowed in a leaf node is determined by the min\_samples\_leaf as 11, and finally we used 100 trees in our model (n\_estimators). Using these parameters we were able to achieve an accuracy of about 60-64%.

The following plots show the effects of modifying some parameters for our Random Forest model:

|  |  |
| --- | --- |
|  |  |
| Maximum number of leaf nodes. A value of 4 appears to achieve the best accuracy. | Maximum tree depth allowed. A value of 2 appears to achieve the best accuracy. |
|  |  |
| Minimum number of samples in leaf node. A value of 11 appears to achieve the best accuracy. | Number of trees in the forest. It appears that the larger the number of trees, the better accuracy. |

**Neural Networks**

After building the Random Forest, we built an artificial neural network model. We had a hunch that this model would outperform all other models because of its non-linearity. The first and simplest model we built had only one hidden layer. All layers were Dense (fully connected), the input and hidden layers used relu as an activation function, and the output layer used the sigmoid function because we needed a binary output (1 or 0 - win or lose). The number of nodes in the layers was 12, 8, and 1, respectively. This model achieved nearly 59% accuracy on testing data (70% on training data).

Per usual, we modified the hyperparameters of the model to look for better accuracy. A better model we were able to construct used two hidden layers instead of one. Like before, relu was used for input and hidden layers activation, and sigmoid was used for the output layer. We used 64, 32, 16, and 1 node(s) in each layer, respectively. This model’s performance jumped into around 62% (75% on training data).

**K Nearest Neighbors**

We tried different KNN models with different values for K. We have achieved the best accuracy (60%) when K = 13.

**Bootstrap Aggregating**

In an attempt to improve the performance of the KNN model. We used bagging to run a couple of KNN classifiers and vote for the output. We experimented with the number of estimators and the value of K for estimators. We achieved around 62% accuracy when the number of estimators = 10, and K = 13.

A future step would be to try to run the model with different K value for each estimator.

The following plots show the effects of modifying different parameters for our Bagging model:

|  |  |
| --- | --- |
|  |  |
| Number of nearest neighbors used to classify. A value of around 37 appears to achieve the best accuracy. | Number of Nearest Neighbor classifiers used. Values of 20 and below appear to achieve the best accuracy. |

**Improving our NN Model**

Most of our time was spent on trying to make the neural network model better. We thought that introducing non-linearity might produce good performance in a complex problem like ours.

**Scaling**

We wanted to explore the potential of our NN model further. We started by scaling all our features to be between 0-1 using the following formula of Min-Max Scaling:

, where , and .

Unscaled data “can degrade the predictive performance of models, and can also slow down or even prevent the convergence of many gradient-based estimators.” [1]

**PCA**

Furthermore, we ran Principle Component Analysis on our features to understand how much variance does each feature cover, and therefore select fewer features that account for most of the variance of our data. We had 108 features at the time, the following figure shows the cumulative variance as we consider a larger number of features.

|  |
| --- |
|  |
| Cumulative variance as the number of considered components grows. |

We can see from the figure that somewhere around 55-70 components explain most of the variance. We chose to continue with 65 components. This explains around 99.54 of the variance of our data.

**Number of hidden layers**

We cut the number of hidden layers down to 1. Our previous model had 2 hidden layers and produced about the same accuracy after scaling and feature selection as a model with 1 hidden layer. Occam’s Razor!

**Number of nodes in layers**

Using the 65 new features, our baseline model had accuracy around 63%, but we noticed a huge difference in training VS test accuracy. Training accuracy was around 90%. This is a case of overfitting. We modelled the training VS validation accuracies for 150 epochs, the following figure shows that while training accuracy was increasing, validation accuracy was decreasing.

|  |
| --- |
|  |
| Training VS Validation accuracy for the baseline model. |

We tried to explore the effects of increasing/decreasing the number of nodes in each layer. Since the number of features is not large, we suspected that a smaller number of nodes (and therefore a less complex network) will reduce overfitting. The following figure shows the accuracies of different models with different numbers of nodes. The baseline model had 32, 16, and 1 nodes in the input, hidden, and output layer, respectively. The small model had 8, 4, and 1, while the large model had 512, 256, and 1. We ran the models for 250 epochs, and added a callback to allow us to early-stop training when validation accuracy stops improving (increasing). We set a value of patience of 75, meaning that the model will early-stop if it does not see improvement in validation accuracy for 75 consecutive epochs. We also added a callback that records the best seen model in terms of maximum validation accuracy.

The figure clearly shows that increasing the number of nodes results in better training accuracy, but it also increases the gap between training and validation accuracy, i.e., overfitting. You can also notice that training stopped around the 90 epoch on average for all models.

|  |
| --- |
|  |
| Accuracies for models with different numbers of layer nodes. |

**Regularization**

The next step was adding regularization techniques to limit the complexity of the model. We first started by adding to the loss during training by penalizing large training weights. The penalty of each layer was the sum of squared weight values of that layer multiplied by a hyperparameter l2 = 0.001. Then we added dropout layers after the input and hidden layers with dropout rate of 0.2. Then we tried a mixture of both techniques. The following figures show the effects of these changes on all 3 models (baseline, small, and large).

The figures show that regularization decreased the gap between training and validation accuracies. They also show that the small model yielded the smallest gap. The accuracies fluctuated between 60%-64% (70%-80% for training) for the best models (using early stopping and model checkpoints to keep track of the best models).

|  |
| --- |
|  |
|  |
|  |
| Different model accuracies with different regularization techniques. |

Based on the results, we chose to continue with the small model, as it reduces overfitting the most. We further added regularization to the output of the activations of the input and hidden layers. Our final model consisted of 3 layers, 8, 4, and 1. It accepted 65 features as input and produces one boolean output, whether the home team wins or loses the match. The learning rate of the model was 0.005, using an Adam optimizer. This model had reduced overfitting as the following figure shows, and had an accuracy of around 68% (72% for training).

|  |
| --- |
|  |
| Final model training VS validation accuracy. |

**Improving our Random Forest**

Evaluating the performance of a model is not always a straightforward task. As our initial random forest suffered from overfitting as well 97.5% training accuracy in comparison to 64% testing accuracy, we spent more time tuning the forest’s hyperparameters.

This time, instead of tuning and observing by hand, we used a hyperparameter optimizer that runs an exhaustive search over a group of selected parameters for an estimator. The optimizer runs cross validations using all combinations of the specified parameters and reports the best combination based on a provided score metric. The best parameters achieved around the same accuracy 64%, but training accuracy dropped to around 74%. We used 3-cross validation. The parameters are [2]:

max\_depth = 6, The maximum allowed depth for the tree.

min\_samples\_leaf = 13, The minimum number of samples required to be at a leaf node.

min\_samples\_split = 10, The minimum number of samples required to split an internal node.

max\_features = 15, The number of features to consider when looking for the best split.

**Model Summary**

Our two models yielded around the same accuracy, with the NN having accuracy ranging between 64%-69% between trials, and the random forest with about 63%. The following figures show accuracy for training and testing data. We had progress on overcoming the overfitting issue, which resulted in lower training accuracy compared to earlier models. The RF suffers from overfitting more than the NN. Another notable observation is that both models produced more true positives than true negatives, i.e., the models have higher sensitivity than specificity. With Sensitivity rates of 95% and 85% for the Random forest and the Neural network, respectively, and Specificity of 12% and 25% in that order. This means that our models can tell when the home team is going to win but are very bad at telling if the home team would lose. This might be explained by the fact that the models did actually learn to give some advantage to home teams, but cannot yet see cases when that advantage is voided.

In terms of model complexity, the Random Forest although complex due to the high number of features, is still considered less complex than the neural network, due to the neural network’s non-linearity. Tuning the random forest was simpler than trying to come up with a good neural network architecture.

|  |
| --- |
|  |
|  |
| Training and Testing data accuracy for different sample sizes. |

**Discussion**

There are many interesting ways this project could be improved with more time and resources. Firstly, we are quite aware that we do not have enough data. This has been the bottleneck throughout all of our work. We have only considered playoff matches throughout the previous 22 years, but that was not enough. Perhaps we should try to add matches from the regular season as well. But this introduces new problems, such as having to collect team performances before each of these matches, then normalizing this data, since, for example, points scored in 82 matches (what we have now) will definitely be larger than those scored before the 40th match of the regular season. But in general, collecting more data will definitely be our number one priority if we had to start over again. Another thing we might want to consider is trying to represent our data in different ways such that more significance is given to home and away status. It is known in most sports that teams playing at home have the advantage, mentally and physically. We wanted to try to capture some of this advantage through adding features like distance travelled before match, but we were not able to collect such data.

We also wanted to integrate player data into each match, but this remains a very tricky task. Because optimally, we would want the network to consider a player’s contribution to the game as a whole, and regardless of their position in the feature matrix. But placing player data side by side in the features matrix will result in players being compared with players in the same positions in other records, feature by feature, and that does not reflect the true values of players as wholes. For example, if each player was described using 10 features, then we would add 50 new columns for the game starters for each team, if the first 10 features in record 1 were describing Kawhi Leonard and in record 2 they were describing Kevin Durant, then the model will compare the two players in isolation of other players in the team. Changing the position of any player will lead to different results. A solution to the ordering problem might be to order players by playing position (i.e., Point Guard, Power Forward, etc), but then again, this still suffers from players not being treated as wholes. Further work needs to be done to figure out the best way to represent players.

One thing we also wish we have done is actually give more chances to other models and algorithms. We were very focused on the neural network because it looked very promising when we completed our progress report, but now that we were not able to get to the 80% goal we set, we started to think that we might have been able to get better results with other classifiers. That is definitely something to consider next time.

Overall, we are glad that we were able to implement and play around with different models, and although we are disappointed that our biggest issue was not having enough data, we are glad we were able to work with what we have and come up with an improved NN model that reduced overfitting, and improved performance by around 10%, all while learning about different techniques that can improve models, like scaling, regularization through weight decay and dropout, applying principal component analysis as learned in class, and hyperparameter tuning.

References:

1. <https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html>
2. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>