Data Intensive Computing - Final Project

Group 5
Adrian Campoy Rodríguez - Fernando García Sanz

October 24, 2020

Abstract

The aim of this project is to generate a custom NBA dataset in order to employ it for predicting the outcomes of the different matches played during a season. By means of API requests performed with python and data processing and cleaning using scala and spark, a dataset with information from the seasons 1990 to 2018 has been created. The results obtained later on with the predictions are quite close to other state-of-the-art outcomes of tasks with the same goal.

1 Introduction

The application of data analysis to sports is currently a hot topic, specially given the new possibilities arising from the application of Machine Learning techniques to them. In particular, using data analysis for forecasting of team-based sporting games has seen a drastic increase in the last decades [1]. This project in particular focuses on predicting the outcome of a NBA basketball considering the data from seasons 1990 to 2018.

2 Tools

spark and scala were used for data processing and data cleaning. Moreover, python programming language was employed in order to perform the API requests, as well as for the later machine learning tasks. More specifically, the library *sci-kit learn* was used in order to apply the Machine Learning models used in this project.

3 Data

The API balldontlie was used to obtain the required data. The following data was obtained:

- **Games** information: Matches played from 1990 to 2018 containing, among other information, home team score, visitor team score and the teams ids.
- **Player** information: This file contained the player id, other additional player information which was not used like height and weight as well as the most relevant team they played in.
- **Team** information: These files contain team id, team name, city, conference, etc. However mainly team id was used from these files.
- **Stats**: These files contain all the stats for each player in each game (e.g. rebounds, blocks, minutes played, etc). These files were mainly used to obtain the rosters of each team and each season.
- **Season Average**s: From here it is possible to access the season average stats for each player and during each season. This information is the one used in the Machine Learning section as features for the model.

4 Methodology

4.1 Data Processing

First of all, we performed the corresponding requests to the API using the python code in the file `ApiRequest.py`. This code allows to get all the necessary information and to save it into different `.json` files for convenience. This way, we downloaded Games Information, Player Information, Teams Information and Stats. However, as it might have been noticed, the information of the rosters of each team and each season was not available in any of these data blocks. Therefore, spark and scala were used in order to process the to aforementioned `.json` files as well as generate the corresponding rosters for each team and each season. This was done as follows:

First, the file RosterGenerator.ipynb was used to process the stats and players .json files, which were loaded as DataFrames. Stats contained, among other information, the player id and team id for each game of each season. Therefore these data needed to be matched with the player id available in players.json. For each season, in the stats file, we retrieved all the player and team ids in that season by filtering per year. Later on, given that one player can play in several teams in the same season, we decided to assign each player to the team with more occurrences (i.e. the team in which the player played more games). In order to do this, the team ids were grouped by player id, aggregated and the team id with more occurrences was chosen. After doing this, we end up with a dataframe containing two columns: player id and its corresponding team id for that season. Then, this dataframe is filtered iteratively by team id, selecting the corresponding player ids that will make up the roster of that team and that season. Once this is done, the set of player ids is is saved as a .json file (one for every team and season).

Then, once all the rosters are saved in .json files, the code in RosterLoader.py is employed for requesting the Season Averages of each player in the team for that season saving them in new .json files. This step was necessary given that the requests to obtain Season Averages required specifying the player ids and season of interest in the request url. As a consequence, it was first necessary to obtain the rosters and then perform requests again to the API using python.

Once this was done, Pre_process_rosters.ipynb was used along with the Season Averages in order to perform two operations:

- First, selecting the 12 most relevant players of each team and season (in a regular basketball game only 12 different players are allowed to play for each team). The criterion used for selecting these players was the amount of time they played, which was recorded in Season Averages. Therefore, these files were loaded as dataframes, ordered by minutes played and then the 12 first elements were chosen.
- Second, the averages were scaled to 36 minutes, which is a common way of normalizing in NBA metrics.

Finally, both the roster with the 12 most relevant players and the one with statistics scaled to 36 minutes were saved into .json files.

Afterwards, a similar process needs to be performed over the Games information in order to store all the played games during each season in different .json files. This is achieved by means of the code contained in Pre_process_games.ipynb. With this code, the Games .json files are loaded into DataFrames and then iterated over season in order to gather all the games corresponding to that year. This information is then saved in a .json file.

4.2 Predictions

Once all the data processing has been done, the following files are obtained:

- A set of .json files containing the season averages of the 12 most relevant players of each team during each season are obtained. This set can be further divided into two subsets:
  - Statistics scaled up to 36 minutes per game.
  - Statistics without scaling.
- A set of files containing all the information of the different games played all along each season (date of the game, home team id, home team score, season, postseason, visitor team id, and visitor team score).

This information was gathered and used in the file NbaPredictor.py. The techniques applied over the data were: Multi-Layer Perceptron (three layers of 100 nodes each), Naive Bayes classifier, Random Forest and Logistic Regression. These methods were selected after some research on different approaches for prediction of basketball games [1, 2]. Also different features were tested according to how meaningful they could be for a basketball game as well as considering the information regarding feature selection provided by [1]. Finally, the selected features were: assists, blocks, defensive rebounds, three pointer field goal attempts, three pointer field goals made, field goal attempts, field goals made, free throw attempts, free throws made, games played, offensive rebounds, personal faults, points, steals, and turnovers[1].

[1] Note that all these statistics are a season average per player obtained from Season Averages

Each sample input in the models represents a match. A data point consists of a vector given by:

$$24 \times (number\ of\ features\ elements + Team_{id}) + b$$

where the 24 corresponds to the number of players in a match (12 local, 12 visitor), *number of features elements* corresponds to each of the season averages listed before for each player, $Team_{id}$ corresponds to the team id of the player (normalized between 0 and 1), and b corresponds to a bias term. A K-fold cross validation with $K = 10$ was used to train and test the models.

The accuracy and ROC-AUC results are shown below:

Method	Accuracy (original DS)	Accuracy (balanced DS)	Accuracy [1]	Accuracy [2]
MLP	64.28% ± 1.26	61.97% ± 1.36	66.60%	-
Naive Bayes	61.21% ± 1.84	60.53% ± 1.72	-	67.6%
Random Forest	61.80% ± 1.16	59.06% ± 1.65	-	-
Logistic Reg.	66.41% ± 1.67	64.88% ± 1.66	67.72%	68.3%

Table 1: Accuracy of models with balanced and unbalanced data sets (first and second column). Accuracy of Perricone et al. [1] and R. Fang [2] (third and fourth column).

Although the results of the accuracy shown in table 1 are close to those from [1, 2], the true positive rate and false positive rate did not provide good results as it can be seen in the ROC-AUC figures 1,2, 3, 4. After further analysis of the data, it was observed that the reason is that approximately 65% of the matches were won by the local team, therefore the data set is skewed. As a consequence, the models were predicting local team as winner (label 0) approximately 65% to 85% of the times. In order to tackle this, the data set was balanced in such a way that 50% of the games used as input had local team as winner and the other 50% visitor team as winner. This way, the model was forced to focus more on learning the relevant features that may lead a team to win or lose rather than simply predicting most labels as 0. Finally, after this modification the accuracy got slightly worse. However a clear change in the behaviour of the ROC-AUC can be observed specially for MLP, Random Forest and Logistic Regression (see figures 5, 6, 7, 8). The ROC-AUC results obtained after balancing the data were closer to those obtained by [1], where AUC of 0.63 with MLP (the only AUC shown) was obtained whereas in our results, although the area under the curve results show high variance, several folds surpass or are close to 0.63, as shown in figure 5.

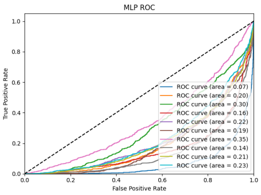

Figure 1: MLP ROC with unbalanced data set

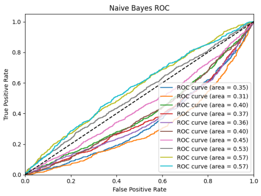

Figure 2: Naive Bayes ROC with unbalanced data set

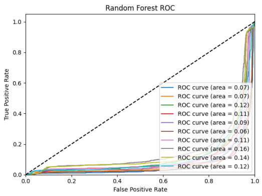

Figure 3: Random Forest ROC with unbalanced data set

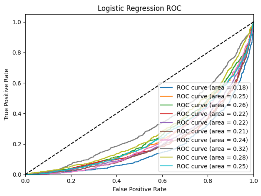

Figure 4: Logistic Regression ROC with unbalanced data set

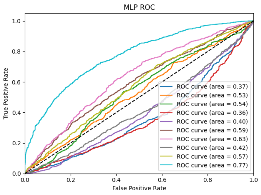

Figure 5: MLP ROC with balanced data set

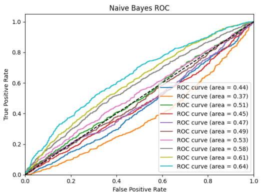

Figure 6: Naive Bayes ROC with balanced data set

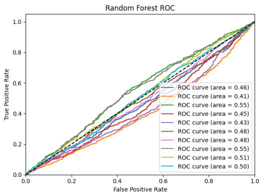

Figure 7: Random Forest ROC with balanced data set

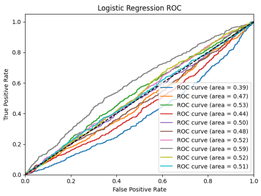

Figure 8: Logistic Regression ROC with balanced data set

5 How To Run

In order to execute the code, the following is needed.

- Execute the `ApiRequest.py` writing the proper *url* in order to get a `.json` file with the wanted data.
- Once all data is recorded, execute the `RosterGenerator.ipynb` notebook in order to build the pseudo-rosters of each team per season.
- Execute the `RosterLoader.py` file in order to get the averages of all players of a team for each team and season, saving them in a new folder.
- Run the `Pre_process_rosters.ipynb` notebook in order to get the team rosters statistics with and without scaling and including the twelve most relevant players.
- Execute the `CleanRosterAvg.py` file in order to place the generated final rosters in the proper paths.
- Employ the `Pre_process_games.ipynb` in order to save all the different games played per season in single `.json` files.
- Finally, use the `NbaPredictor.py`, customizing the features to consider and the classifier parameters, in order to predict the outcomes for the test seasons.

References

[1] J. Perricone, I. Shaw, and W. Swiechowicz, "Predicting results for professional basketball using nba api data,"

[2] "Nba predictions." `http://dionny.github.io/NBAPredictions/website/`. Accessed: 2020-10-22.

[3] "balldontlie api." `https://www.balldontlie.io`. Accessed: 2020-09-24.