Predicting the chance of NBA players scoring

João C. L. Folhadela and Nuno M.F. Capitão

Abstract

The rise of data science and machine learning (ML) applications in sports-analytics has proven to enhance team and athletes performance, and it's becoming more prevalent in the industry. This article aims to explore and shape strategies for offensive basketball plays by computing the probability of the players from NBA teams scoring based on past historical data. To do so, three machine learning models were used and compared: Logistic Regression (LR), Support Vector Machine (SVM) and Feed Forward Neural Network (FFNN). For the 5 players with most entries in the data set we obtained an average accuracy of $\sim 62\%$ for the LR and $\sim 61\%$ with the SVM. The FFNN attained an average accuracy of $\sim 70\%$, a significant increase compared to other models although its hyper-parameter tuning for each player was more time consuming. To illustrate a practical application, an iterative web-app was designed, resorting to the Streamlit app framework, which presents a few case studies making use of the FFNN model.

Key words: Machine Learning, Data Science, Support Vector Machine, Feed Forward Neural Network, NBA, data app, Sports-analytics

Introduction

Project goal

The goal of the project is to delve into some of the common offensive basketball plays and predict the best possible outcome of hitting the basket on an NBA court. Standard basketball features such as team composition, game time, distance to the closest defender and others were looked at in order to assess the best trajectory to move the ball and which player within the attacking positions should shoot the ball.

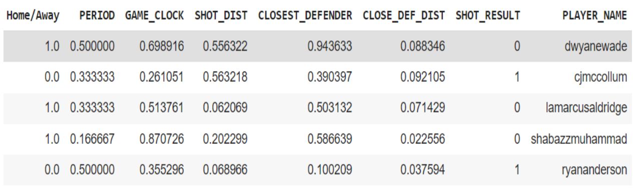
Since we are working with a bi-dimensional classification problem (predicting whether a player scores or not) the algorithms chosen were based on the LR, SVM and FFNN models. The algorithm with the best performance was then connected to an web-app to help represent some practical examples graphically and allow an user to simulate an offensive play of his choice and select inputs from any NBA team and set of players, as well as the other features. The probabilities of each player scoring are then displayed alongside an animation representing to whom should the ball be passed to.

Data collection & preparation

The data set is obtained through `https://www.kaggle.com/dansbecker/nba-shot-logs`. It consists of NBA games throughout the 2014-2015 regular season. After cleaning the data, encoding names and removing irrelevant data points, we're left with 281 attacking players (players that took a shot, column PLAYER_NAME), some having over 900 entries, and the shot taken (column SHOT_RESULT) will be the models prediction (hit = 1, miss = 0) and the information related to this prediction, that will be taken into consideration and work as features for our models, will be:

- Home/Away - the shot was taken in a Home/Away game for the team's player (HOME = 1, AWAY = 0)
- Period - period of the game at which the shot was taken (PERIOD = [1,4])
- Game clock - at what game time was the shot was taken (GAME_CLOCK = [0,12])
- Shot distance - distance from the basket the shot was taken (SHOT_DISTANCE)
- closest defender - which defending player was closest to the player taking the shot (CLOSEST_DEFENDER = number associated to the defending player)
- closest defender distance - the distance of the attacking player from the closest defender (CLOSE_DEF_DIST)

Data preparation consisted of replacing names that were misspelled, removing NaN values, normalizing the entire data and encoding the features that had string values into numbers (Defending player, for example) below is the head of the data frame ready to be worked on.



Home/Away	PERIOD	GAME_CLOCK	SHOT_DIST	CLOSEST_DEFENDER	CLOSE_DEF_DIST	SHOT_RESULT	PLAYER_NAME
1.0	0.500000	0.698916	0.556322	0.943633	0.088346	0	dwyanewade
0.0	0.333333	0.261051	0.563218	0.390397	0.092105	1	cjmccollum
1.0	0.333333	0.513761	0.062069	0.503132	0.071429	0	lamarcusaldridge
1.0	0.166667	0.870726	0.202299	0.586639	0.022556	0	shabazzmuhammad
0.0	0.500000	0.355296	0.068966	0.100209	0.037594	1	ryananderson

Fig. 1. Normalized Data Frame head() after data preparation.

Model Training

Every model was implemented creating two functions, one to fit the model with given parameters and a train set, and another to tune its parameters, validate it, test it and return the model's predictions. A problem that arises is that every player must have its own data (i.e. we don't want a less experienced player skewing LeBron James data, therefore getting less accurate predictions for him) so the model must be trained for each player data set. The second function has as input the player to be evaluated, it picks his data and trains, validates, and tests accordingly.

All models were trained using libraries that already have built-in functions of the models ready to use, i.e.:

- LR - scikit-learn linear model LogisticRegression model;
- SVM - scikit-learn SVM.SVC model;
- FFNN - tensorflow keras sequential model;

Therefore the training part is substantially made easier, for each model the required is only to apply a ".fit(x_train,y_train, hyper-parameters)" where the the inputs are the already split data for given player and the hyper-parameters corresponding to the model.

Below we can see the functions made that train the models, having the proper imports been made, of course.

```python
def LR(x,y,C):
  return LogisticRegression(max_iter=1000,
                                   C=C).fit(x,y)
```

Code 2 - LR model training function - takes as inputs train data and parameter C; returns the trained model

```python
def SVM_fit(C,d,x,y):
  return svm.SVC(kernel='poly',degree=d,
                      C=C,coef0=0.1,
                      probability=True).fit(x,y)
```

Code 1 - SVM model training function - takes as inputs train data and parameters C and D; returns the trained model

```python
def model_fit(nodes1,nodes2,x_train,y_train,
                x_val,y_val):
  model = Sequential()
  model.add(Dense(nodes1,input_dim=6,
                activation='relu'))
  model.add(Dense(nodes2,activation='relu'))
  model.add(Dense(1,activation='sigmoid'))
  model.compile(loss='binary_crossentropy',
                    optimizer='adam',
                    metrics=['accuracy'])

  return model.fit(x_train,y_train,epochs=25,
                     batch_size=15,
                     validation_data=(x_val,
                     y_val),verbose=2)
```

Code 3 - FFNN model training function - takes as inputs train and validation data, #nodes per layer; returns the trained model; returns the trained model. **Note**: the code used in the project, for this last model, only returns the compiled network, the fitting comes later.

The tricky part now becomes tuning the hyper-parameters for each model, for each one has its own unique parameters to tune and trying to find the best ones is what requires most computing processing, therefore making the code run much slower. Below is a table showing the parameters corresponding to each classification model.

Model	Hyper-parameters
LR	C - Inverse of regularization strength
SVM	Kernel; C - Same as LR; ; D - degree of poly kernel G - gamma of RBF kernel;
FFNN	# layers; # nodes per layer; # epochs; batch_size

Table 1. Hyper-parameters used for each model

More parameters from each model's library documentation can be used to increase the accuracy but for this project the ones mentioned above were the only ones that impacted the performance.

The LR and SVM models were tuned using a cross validation score from sklearn, it takes the fitted model, using the training set, and compares scores for the validation set. This is calculated for each parameter value. For LR this is done through a routine which tries out several C values, from 0.1 to 10 in steps of 0.5, the validation score is therefore stored in a list and the maximum gotten corresponds to the best parameter. The LR model is therefore calculated with this new best parameter, an implementation of this algorithm is seen below:

```
Cs = np.arange(0.1, 10, step, dtype = float)
means = []
accs = []

for C in Cs:
    scores = cross_val_score(
        LR(LR_train_x, LR_train_y, C),
        LR_val_x, LR_val_y, cv=10)
    means.append(np.mean(scores))

max_index = np.where(means==max(means))[0][0]
best_C = Cs[max_index]

validated_LR = LR(LR_train_x,
                  LR_train_y, best_C)
```

Code 4 - Parameter tuning algorithm example using LR model function.

The same algorithm is implemented for the SVM model, with an extra cycle being added to tune parameter D, in the same way C parameter was before, so now both are tuned at the same time to find the best combination possible. The implemented code went through D = [1,2,3] and C from 0.1 to 10 in steps of 0.5, which makes a total of 60 possible combinations of C and D.

For the FFNN model we once again use the parameter tuning algorithm, but for cycles corresponding to the number of nodes, so two cycles, one for each node. The algorithm will try combinations of nodes from 1 to 23 nodes in steps of 3, which results in a total of 49 combinations. As for the number of layers, epochs and batch_size, those were previously tuned by trial and error, trying to have an equilibrium between computation time, accuracy and convergence.

Model evaluation

To evaluate the models, a fixed set of features was chosen (features = [1,1,7.56,3.6,210,13.7], each value corresponding, respectively, to each feature seen in the Data section) also, 5 players with the most data entries were selected and tested as they will have the data sets less prone to error and miss-classification. The model performance will be evaluated based on the accuracy score obtained, for the LR and SVM models, and val_accuracy, for the FFNN (this value is returned by the fitted neural network, it is similar to how accuracy score is calculated) and also on how much time it took to get those accuracies. The implemented functions calculate the accuracies for the best hyper-parameters got and the following tables summarize the results gotten.

LR Model

Player name	Accuracy	Avg. Accuracy	Time
lebronjames	67.9 %	62%	10s
klaythompson	62.1%		
montaellis	61.5%		
jamesharden	56.5%		
lamarcusaldridge	61.3%		

Table 2. LR model evaluation table

SVM Model

Player name	Accuracy	Avg. Accuracy	Time
lebronjames	66.7 %	61%	2mins.
klaythompson	61.1%		
montaellis	64.8%		
jamesharden	51.1%		
lamarcusaldridge	60.4%		

Table 3. SVM model evaluation table

FFNN Model

Player name	Accuracy	Avg. Accuracy	Time
lebronjames	68.93%	70%	10mins.
klaythompson	69.90%		
montaellis	70.87%		
jamesharden	70.87%		
lamarcusaldridge	68.93%		

Table 4. SVM model evaluation table

What we can conclude from these tables is that the LR and SVM models have a very close Avg. Accuracy but the SVM takes much more time to compute the best parameters, as it should since there are more parameters being tuned. As for that reason we can say that, for this case, both models perform very similarly. Once we reduce that amount of parameters, for the SVM case, we get similar executing times.

For the FFNN we see an abysmal executing time increase as it would be expected: every single test to each number of different nodes has itself a validation of the network hidden parameters as well, nevertheless, this is a crucial step to have good predictions for every single player. One could use the same number of nodes for all 5 players and could've seen an increase in accuracy for some players but a reduction for others. So this is the most correct approach to get better results even though too time costly. Even though picking a number of nodes for all players is bad, this must be done as to effectively implement the function into the web app, otherwise one would have to wait for an eternity to get a result.

Next we analyze the accuracies behaviour for a single player based on the parameters used to see how much they impact the model.

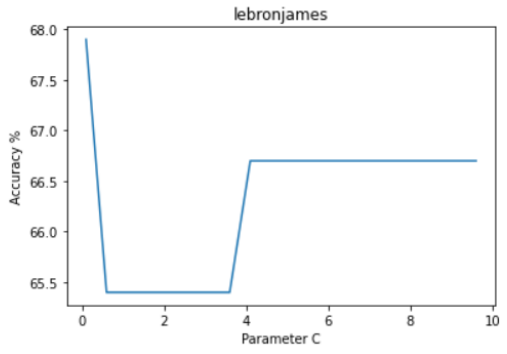
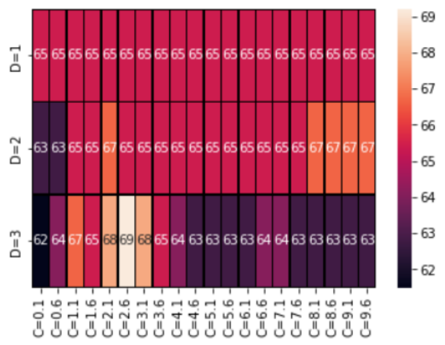


Fig. 2. LR model for Lebron james data - accuracy vs. parameter C



	C=0.1	C=0.6	C=1.1	C=1.6	C=2.1	C=2.6	C=3.1	C=3.6	C=4.1	C=4.6	C=5.1	C=5.6	C=6.1	C=6.6	C=7.1	C=7.6	C=8.1	C=8.6	C=9.1	C=9.6
D=1	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65
D=2	63	63	65	65	67	65	65	65	65	65	65	65	65	65	65	65	67	67	67	67
D=3	62	64	67	65	68	69	68	65	64	63	63	63	63	64	64	63	63	63	63	63

Fig. 3. SVM model for Lebron james data - accuracies for given D and C

We can notice that indeed the parameters values do have some impact in the accuracy, so it is rather important to always try getting the best possible ones.

We can also check that the FFNN model training and testing loss converge meaning there's no over/under fitting. For some cases the convergence takes more epochs to reach, on others it is almost instantaneous. Below is a graph showing a model trained on Lebron's data converging after only 4 epochs.

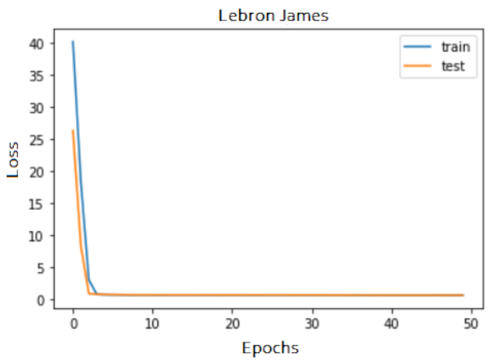
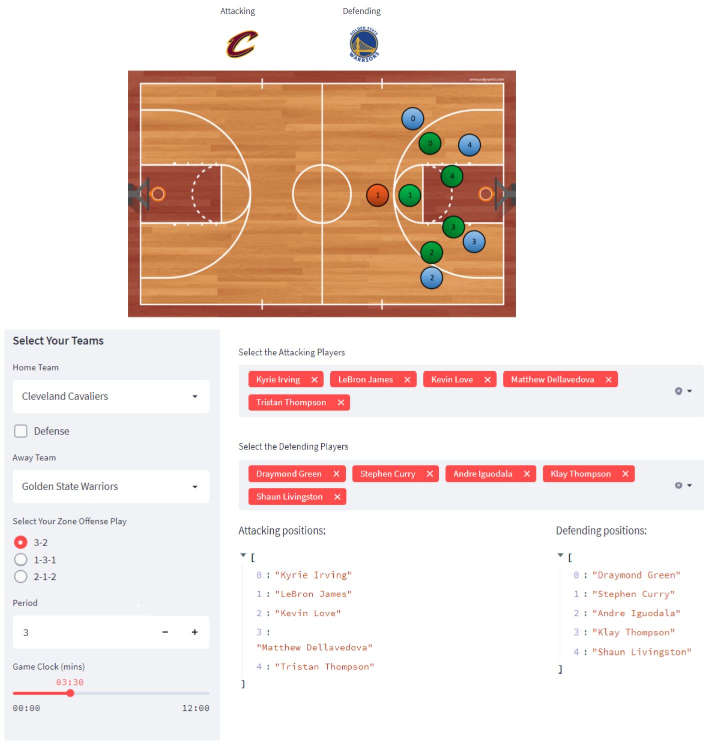


Fig. 4. FFNN model training on Lebron James data - Loss vs #epochs; the model does converge

Model Deployment

In order to bridge the project with a practical application, we built a simple web-app using the streamlit framework with selectable various types of case studies. The different input features consists of data acquired from the nba_api.stats and basketball_reference_scraper APIs. Since the data-set covers the 2014/2015 regular season, basketball_reference_scraper recovers every NBA team roster from that year and the nba_api.stats provides detailed information about the team and the players.



Select Your Teams

Home Team

Cleveland Cavaliers

☐ Defense

Away Team

Golden State Warriors

Select Your Zone Offense Play

[✓ 3-2]
1-3-1
2-1-2

Period

3

Game Clock (mins)

03:30

00:00 12:00

Select the Attacking Players

Kyrie Irving ✕ LeBron James ✕ Kevin Love ✕ Matthew Dellavedova ✕ Tristan Thompson ✕

Select the Defending Players

Draymond Green ✕ Stephen Curry ✕ Andre Iguodala ✕ Klay Thompson ✕ Shaun Livingston ✕

Attacking positions:

```
[
  0 : "Kyrie Irving"
  1 : "LeBron James"
  2 : "Kevin Love"
  3 :
  "Matthew Dellavedova"
  4 : "Tristan Thompson"
]
```

Defending positions:

```
[
  0 : "Draymond Green"
  1 : "Stephen Curry"
  2 : "Andre Iguodala"
  3 : "Klay Thompson"
  4 : "Shaun Livingston"
]
```

Fig. 5. Different available input features for the user

There are several offense open plays at disposal from which an user can select any preferable NBA team and assign players from its 2015 roster to each position (as long as it exists in our data set). Other parameters such as the period and the game clock are also selectable. All these features are then turned into a likelihood of scoring using the FFNN model.

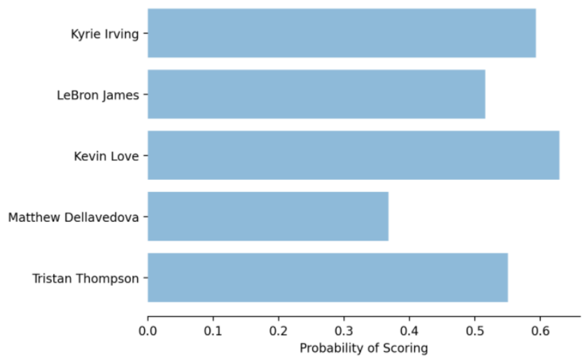


Fig. 6. Output of the probabilities obtained by the FFNN model for the selected features

The output returns which player is more likely to score from his position. Most of the backend development from the web-app is linked with the caching mechanism *@streamlit.cache* to optimize its performance. However, it's important to mention that the level of fluidity and otimization desired was not achieved, hence there's still some room for improvements with better structuring and using other libraries.

Conclusions

The models trained in this project don't retrieve an optimal accuracy for real world implementation. The one that always outperformed was the Neural Network (with an +8% increase in accuracy compared to other models) which is to be expected. It's important to notice that our data set is relatively small from an individual player duels point-of-view. Since it's only data from a regular season, players could have only played each other twice if not less in case of injuries. So a good way to improve the results would be by training the models with data acquired from several seasons, including the play-offs and also increasing the number of features feeding the training as there are many more variants that may point out to a success full hit (ex. angle at which ball was thrown, hand used, etc.). If we shift the practical example studied into a real world application, it's possible to train any play for a game beforehand and see where the ball should be moved taken into account past historical duels, which can lead to a better game performance. Although we're working with a single frame from an open play, it can be upgraded to real-time information with adequate technology, data and model training.

References

1. Medium article - Data Science: The science of moving dots in Basketball and shot value
2. Basketball Court Dimensions
3. The Math Behind Basketball's Wildest Moves — Rajiv Maheswaran — TED Talks
4. NBA shot EDA
5. Machine learning mastery article - How to use Learning Curves to Diagnose Machine Learning Model Performance
6. How to Make Predictions with Keras
7. Basketball playbook
8. Rim height and court dimensions
9. Burkov, A., Machine Learning Engineering, True Positive Incorporated, 2020.