CIS 5200 Fall 2022, Project Report

Predicting NBA Player-Prop Lines and Outcomes

Krish Shah and Avi Upadhyayula

Abstract

Given the heightened interest in player-prop markets across the sports betting/trading industry, our paper seeks to analyse the implementation of machine-learning models to predict NBA player-specific single-game scoring. We use a rolling-window feature engineering approach across a variety of machine learning techniques, exploring the impact of using different loss functions on various performance metrics. Ultimately we find that while sophisticated ML models outperform a naive baseline, the improvement is minimal and a more specific model is most likely necessary.

1 Motivation

The sports betting industry has taken off in the last few years and is expected to reach close to 180 billion USD by 2030. This is compounded by the emergence of sports trading exchanges, where individuals can buy/sell positions in bets that would have otherwise been static (held until the game ends).
With this demand for liquidity, there is significant opportunity for firms that can provide strong pricing. In particular, player-prop markets have been known to be poorly priced (both because the problem is difficult to model correctly and because sportsbooks and fantasy apps often neglect order flow information), opening up an even larger need for efficient pricing.
With the increasing amount of data being tracked and generated from sports games, machine learning models may be able to gain insight into performance that human intuition is unable to comprehend.
In order to keep the problem manageable, we work to show a proof-of-concept for machine learning models in predicting NBA player-specific single-game scoring.

2 Related Work

While somewhat extensive work has been done to apply machine/statistical learning models to predict game-outcomes, there has been extremely limited research into applying models to player-prop markets.
We found one blog post that had done some work, but the bulk of the work was scraping betting data (and NBA data) and then implementing a few regressions with season-average windows. While this approach is initially informative, it is incomplete and doesn't take into a few important considerations:

- The season average statistics may be computed over windows of differing lengths.
- It uses the market closing line as part of X when in reality this information is only available immediately before the start of the game.

We aim to replicate the model success shown here without using market information, and time/resource permitting, improve upon the results when we do apply market information.
You can view the entire post here:
`https://python.plainenglish.io/nba-betting-using-linear-regression-beba050a50fb`

3 Data Set

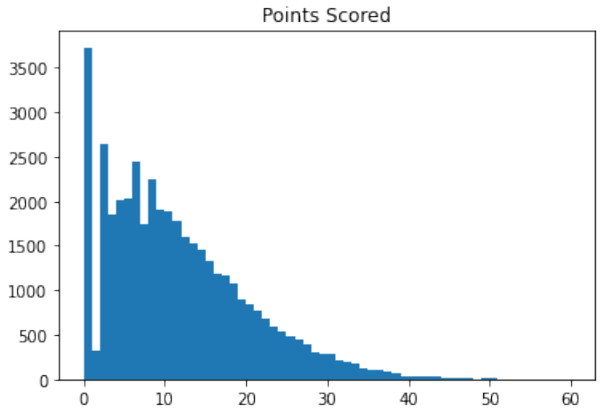
3.1 Datset Description

Our dataset consists of 56,000+ observations of NBA player box-score information over the course of the last 2+ NBA seasons. We also have information for the team-wide statistics in each of these games. We discuss the feature engineering process in the next section, but pre-processing was required in order to set up our X so that for every row we have information from games played prior to that game as well as the outcome (number of points scored) in that game by the specific player.

Additionally, we have market-line information for 20,000+ of these observations. This information consists of the player, the game and the market over/under price on the number of points that player will score in the game.
For the majority of our analysis, we consider a subset of the 56,000+ observations, either filtering so that we have sufficient prior information to construct X or taking the intersection of our raw and market datasets. *Note that many of our models could be run using a much larger training dataset, but we believe that the number of observations we have is a good balance between making our models computationally efficient while still providing enough information to gain signal.*

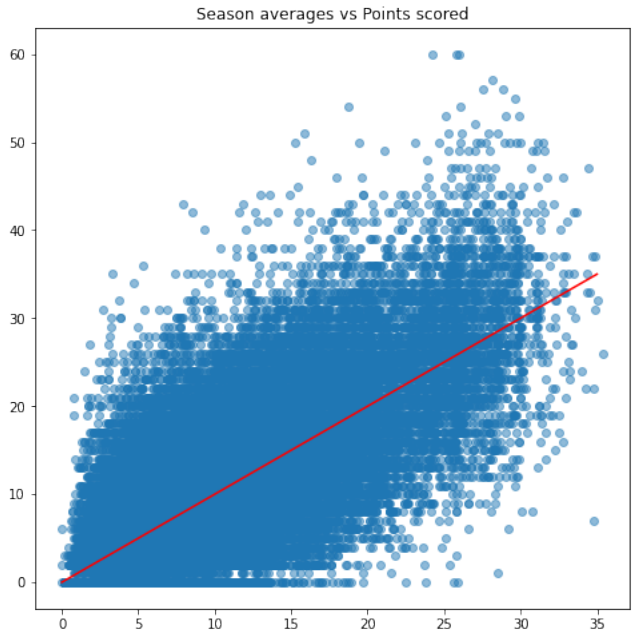
3.2 Exploratory Data Analysis

Exploring feature-specific distributions within the dataset allows us to make domain-informed decisions when implementing various methods. We first explored the distribution of points scored across all rows:



We noticed some interesting observations about this distribution:

- It is clearly zero-inflated, most likely because our distribution includes players who do not play at all in a game. Noting this, we should probably exclude players under a certain average MPG threshold.
- The distribution is evidently one-deflated. In the context of how NBA games are played, this makes sense. Scoring a single point in a game is extremely unlikely (you'd have to make no field goals and only a single free throw).
- The distribution may follow some sort of count process. Possibilities include a Poisson distribution or some kind of negative binomial distribution.



Something else we noticed while exploring the data was the prevalence of outliers. Players seemed to score significantly above or below their season average more frequently than expected. This supports our concern for variance within the dataset: individuals may go on "hot" or "cold" streaks within a single game, producing single-game point totals that are far off what we'd expect. Moreover, there isn't a good way to eliminate these outliers from our analysis (nor is it necessarily principled to do so). Players getting injured or being unexpectedly thrust into large roles (or simply having a really good game) makes individual game scoring rather unstable. We discuss in Section 7 about potential alternatives.

Finally, the following table contains a statistical description of season averages in important player metrics.

	Usage Rate	Points	MPG	FT%	2PFG%	3PFG%
Mean	0.188	10.521	23.021	0.755	0.522	0.321
SD	0.055	6.379	8.309	0.139	0.091	0.124
Min	0.043	0	1.8	0	0	0
Max	0.396	35.384	39.4	1	1	1

4 Problem Formulation

4.1 Machine Learning Formulation

Consider an NBA player P playing in a game G for team A against team B. We want to predict how many points y this P will score in G. This is a regression task.

4.2 Feature Engineering

We can think about our various information in terms of each of these entities:

- P: Information about a player's performance.
- A: Information about the team the player plays for
- B: Information about the team the player is playing against

The number of points scored by a player, y, is determined by

$$y = 3 * 3\text{PA}_G * 3\text{P}\%_G + 2 * 2\text{PA}_G * 2\text{P}\%_G + 1 * \text{FTA}_G * \text{FT}\%_G$$

.
We can see there are two main there are two main types of information we are interested in:

- Usage: How often do certain types of actions (eg. three point attempts) occur? (i.e. what is the value of 3PA_G)
- Efficiency: When a certain action/attempt occurs, how often is it successful? (i.e. what is the value of $3\text{P}\%_G$)

Note that when predicting y we can only use information for games that happened *prior* to G. Obviously, we cannot directly observe each of the variables in the equation, so we have to provide information about each of them to the model so that it may predict the score. We argue that each of these values is a combination of a player's individual usage/efficiency, structured within the scoring environment created by their team's offense against the other team's defense.
This leads us to believe that the following information (for each of the entities) may be important:

- For the player P:

  Usage: usage rate, minutes/game, free throw rate, three point rate, two point rate, turnover rate
  Efficiency: three point percentage, two point percentage, free throw percentage

- For the teams A, B (where for B this would be usage/efficiency they allow defensively):

  Usage: possessions, free throw rate, three point rate, two point rate, turnover rate
  Efficiency: three point percentage, two point percentage, free throw percentage

We also need to balance two key concerns in the model:

- Non-stationarity: Each of these three entities may be changing over time, a team may be trying something new on offense or defense, a player may be shaking off rust or becoming more comfortable, etc.
- Variance: Individual performances may exhibit high variance, and it is unclear whether "hot streaks" (where a player may be more likely to perform well given they have recently been performing well) exist.

We hypothesise that recent performance is more important when considering usage metrics and historical performance is a better predictor of efficiency.

In order to allow our model to at least entertain these possibilities, we will use rolling windows to compute the relevant statistics. That is, for a window of length x, we will compute the relevant statistics by looking at the last x games played by that player or team. Note that what values of rolling windows we use are hyperparameters that theoretically should be trained, but for ease we will just start with $5, 10$ and year-to-date.

In addition to these terms, we also included interaction terms between the rate and efficiency statistics within each time window.

Further note that hypothetically, you could get a rolling window for every value of x (which would essentially be the same as feeding the entire raw data). This might be more appropriate for models that can handle $p >> n$, but because of time and complexity constraints we did not implement it.

4.3 Loss Functions

As this is a regression problem, there are two natural loss functions we could apply: mean squared error or mean absolute error. While using MSE (L2 norm) can often be more computationally practical, it is not as robust to outliers. We found in our data a few significant outliers (simply due to the inherent variance in the number of points an NBA player can score). It is certainly reasonable that a model with a lower MSE might be practically worse because it does better on a few outliers but is generally worse. MAE (L1 norm) is more robust to outliers but can result in significantly longer training times. In certain models, we look at alternative loss functions to try and balance between the two.

5 Methods

5.1 Baseline Method

Our baseline method is to simply naively predict that a player will score their season average. This method could correspond to a simple MLE prediction of how many points a player will score if the only information we had was the previous number of points they had scored.

5.2 Linear Regressions

We fit a standard linear regression, ridge regression (L2 penalty) and LASSO regression (L1 penalty) using the methods from the sklearn.linear_model library.
For both Ridge and LASSO regression we used cross-validation to find the optimal hyperparameter for α, which controls the size of the penalty.

5.3 Stochastic Gradient Descent Regression

We can use stochastic gradient descent regression to more quickly train and evaluate linear regression models, allowing us to do a complete Grid Search over a larger hyper-parameter space.
SGD Regression can implement an elastic net (combination of L1 and L2 regression), which enables our model to do implicit feature selection/regularization. We will use the Huber loss function. This loss function is quadratic for smaller errors, but after a certain threshold (determined by epsilon) it becomes linear. This limits the impact of outliers on our overall loss (while still giving them some credit).
Because of its speed, we can do 10-fold cross validation to find the optimal hyperparameters for alpha, learning rate, epsilon and the L1 ratio.
Our implementaiton uses sklearn.linear_model.SGDRegressor().

5.4 Support Vector Regression

Due to the size of the dataset, support vector regression can become expensive. However, we still wanted to see how a model using radial basis functions would perform, so we fit a support vector regression using sklearn.svm.SVR().
We chose the kernel to be 'rbf' because we had already fit linear models with the SGD Regression. Due to the size of the data, tuning hyperparameters using a grid search was infeasible, so we did some light hyper-parameter tuning by observing whether our regressor was potentially overfitting and reducing the C parameter in order to combat this.

5.5 Random Forest Regression

We also used a random forest model to try to take advantage of an ensemble learning method. The Random Forest trains many weak decision trees and then uses boosting to improve the overall predictions. We use bootstrapping in order to further reduce the risk of overfitting and limit the variance of our estimates.
Because fitting a random forest can be computationally expensive, we used a randomised 2-fold cross validation to help select hyperparameters for min_samples_split, max_features and max_depth, which are all designed to control the strength of each decision tree.

Additionally, we wanted to fit the random forest using both a squared error and absolute error loss function, using 300 estimators for both. We looked at using Poisson deviance as well, but decided that since our data exhibits over-dispersion naturally this would not be a suitable loss function. Fitting with absolute error was also extremely slow, so we reduced the number of estimators (making it practically unhelpful but still wanted to include it in a theoretical discussion of which models might be effective).

5.6 Neural Net

Although typically deployed in classification tasks, we chose to apply a Neural Network (`https://keras.io/api/models/sequential/`) to this regression problem. Neural Networks simulate a multi-step regression by modeling observations as a nonlinear combination of initial features, and then setting weights for each neuron in hidden layers via backpropagation. To that end, we chose to use a He distribution to select initial weights. (Keras's standard method uses the Normalized Xavier Distribution, which is suboptimal for networks that use the ReLU activation function.)
We structured the network with an input layer and 4 hidden layers, with a dropout layer before the output. This was done to prevent overfitting. We chose to have 4 layers after realizing poor performance with less complex models. We suspect this phenomenon to occur as a result of the dataset's complexity, and a high degree of interaction between each feature.
We also experimented with different loss functions: MSE and MAE. Both are useful when performing regression tasks, but MSE produced worse results on the test set. We suspect this to arise from outliers in the dataset: datapoints that perform poorly are weighted too highly.

5.7 CNN

We apply a 1D convolution layer (`https://keras.io/api/layers/convolution_layers/convolution1d/`) to create a Convolutional Neural Network. Convolutional neural network models (dubbed CNNs) were initially developed for image classification problems. In standard 2D CNNs, the model learns an internal representation of an input with two dimensions.
1D CNNs harness this process and apply it to one-dimensional sequences of data. Typically used in tasks with active observations (for example, sensor data), 1D CNNs can be useful for regression on time series data. Our data did not exactly conform to this standard; but we believed our feature engineering produced features, like rolling windows, that were comparable.

5.8 xGBoost Regression

xGBoost is a library for implementing gradient boosting models. In particular, we are interested in using it for this regression task. xGBoost implements gradient boosting extremely quickly, which allows us to use cross-validation to find good hyperparameters. We implemented xGBoost Regression using both a mean squared error loss and a pseudo-huber loss function.
xGBoost also allows us to easily visualize feature importance, which makes the model easier to interpret than an sklearn Random Forest or other boosting techniques.

6 Experiments and Results

6.1 Hyperparameters

Ridge Regression:

```
{alpha = 8, fit_intercept = True}
```

LASSO Regression:

```
{alpha = 0.01, fit_intercept = True}
```

Support Vector Regression:

```
{C = 0.4, gamma = 'scale', kernel = 'rbf'}
```

SGD Regression:

```
{alpha = 1e-06, average = True, epsilon = 0.6, l1_ratio = 0.01, learning_rate = 'optimal',
loss = 'huber', penalty = 'elasticnet'}
```

Random Forest Regression:

```
{n_estimators = 300, min_samples_split = 50, max_features = 'sqrt', max_depth = 7,
criterion = 'squared_error', bootstrap = True}
```

xGBoost Regression:

```
{n_estimators = 500, max_depth = 6, learning_rate = 0.01, booster = 'gb_tree'}
```

6.2 Performance Metrics

As discussed when looking at loss functions, it may not be completely appropriate to just look at squared error due to the presence of outlier performances. Thus while we were primarily interested in mean squared error (for computational efficiency), we were also interested in the mean absolute error and median absolute error. While it would be potentially interesting to look at mean absolute percentage error, because some of the true values are 0, we cannot compute the percentage error for every observation, making this unfortunately infeasible.

6.3 Performance Results

We ran each of our above models using the same training and testing set, implemented with sklearn.model_selection.train_test_split() using test_size = 0.25 and random_state = 21. We used standard scaler to fit and transform the training data and then fit the testing data. For our models that were not scale invariant, this made sure we did not impose a relative importance on variables beforehand. It also made discussing coefficients/importance of variables possible after the fact.

Model	Train RMSE	Train MAE	Train MdAE	Test RMSE	Test MAE	Test MdAE
Baseline	6.15	4.71	3.74	6.29	4.81	3.82
Linear Regression	5.99	4.64	3.78	6.16	4.76	3.86
Ridge	5.99	4.64	3.79	6.16	4.77	3.87
LASSO	5.99	4.64	3.79	6.16	4.77	3.85
SVR	6.03	4.55	3.58	6.25	4.78	3.73
RF Regression (MSE)	5.86	4.57	3.81	**6.15**	4.79	3.97
SGD Regression	6	4.62	3.73	6.16	**4.75**	3.81
Neural Net (MSE)	**4.28**	**3.29**	2.65	6.95	5.37	4.28
Neural Net (MAE)	5.07	3.58	**2.41**	6.74	5.17	4.08
CNN (MSE)	5.91	4.61	3.83	6.18	4.82	3.99
CNN (MAE)	6.01	4.51	3.49	6.3	4.77	**3.71**
xGBoost (MSE)	5.71	4.45	3.65	6.16	4.76	3.86
xGBoost (Pseudo-Huber)	5.85	4.37	3.34	6.27	4.76	3.72

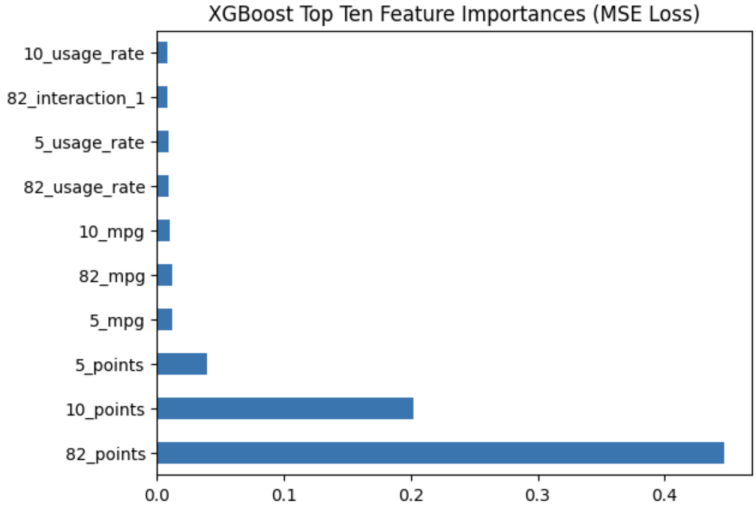
Table 1: Summary of Performance Results

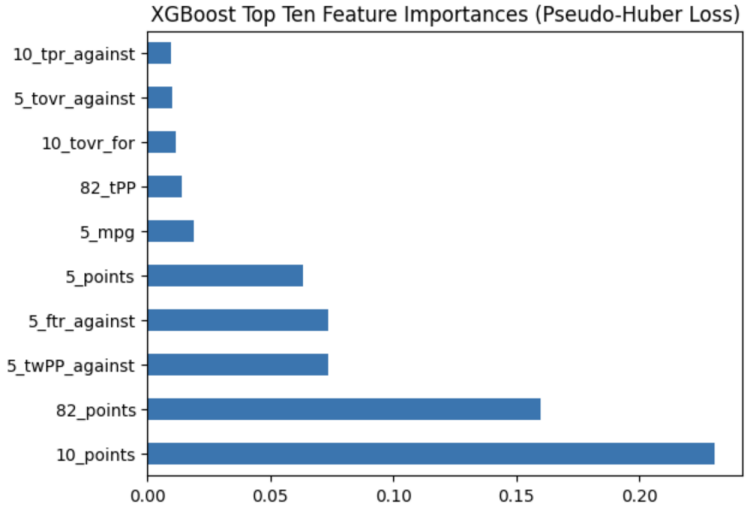
7 Conclusion and Discussion

7.1 Interpretation of Results

One of the key findings we had after running linear models is relative feature importance. The most important feature was the season average in points, which lends credence to the relative success of the completely naive method of simply predicting that average. While our models all performed better than the baseline, the increase in performance is not large compared to the computational cost associated with running each model.

As expected, our regression models tended to favor the smaller window variables for usage variables and larger window variables for efficiency variables. Interestingly, there was little discernible difference between in regression models using L2, L1 or no regularization penalty. This could be because the model only really focused on a few key features and minor differences in the others (whether they are set extremely small or zero) did not impact the overall predictions.
Further, changing the loss function to otherwise identical models has the expected effect of giving the model better (relative) performance on the performance metric that its loss function corresponds to. This means that depending on the practical use of the model, it may be better to specify a loss function that most closely resembles the performance you are looking for. (Although it is extremely likely the neural net is overfitting).
Looking at the xGBoost feature importances, we can see that different patterns emerge when looking at MSE versus Huber Loss.





When looking at MSE, the points features are the only features that have major importance (and the season average is the most important). However when transitioning to pseudo-Huber loss, other features start to become more important and the ten game window becomes more important then the season-average window. These features not only tend to be more shorter-window based, they also are more opponent-specific as opposed to player-specific. This fits our initial hypothesis that player-specific features are not the only features important for predicting scoring.

7.2 Extensions

The biggest area to extend this project is to utilise the market information we collected. Unfortunately, due to time and data quality constraints, we were unable to properly merge our market information with the rest of our data.
To handle the inherent variance in points per game, it may be more prudent to use the market closing price as the y variable. We could also use the market opening price as part of X (and then try to train on the residuals).
This could have a few potential applications:

- Predicting the closing line: There is strong evidence in other sports betting markets that the efficient markets hypothesis holds relatively true, meaning that the closing line can often represent the best possible prediction for the individual scoring. Trying to predict this price can not only help set better prices earlier, but also might provide directional signal as to which way prices will move.
- Using the opening line: The opening line may not be fair, but might be a good starting point for a model to then use signal from other variables to adjust up or down. Putting a lot of the contextual onus on the opening line, as opposed to our limited contextual information may prove more effective.
- If we have some regressor R that generates $\hat{y}$ and market price y_m, we can use a classifier to determine $\Pr[y > y_m|\hat{y}]$, which would allow us to create betting strategies in player-prop markets.

Our project provides some evidence that trying to predict the true outcome may be a difficult task, so pivoting the analysis to market prices instead may lead to more effective/interesting results.
That's not to say that there are no further extensions in predicting points directly. Here are a few potential extensions for our original problem:

- We could try to use different distributions to model the outcomes (or even use some kind of probability distribution to model the individual inputs and their relationship with the outcome).
- We could build in the use of priors and implement a Bayesian model that updates after each game.
- We could implement more time-series approaches using the full vectors as the inputs (as opposed to cutting into windows, which we initially did to reduce variance in the X vector).

7.3 Lessons

We have a few important takeaways from this project:

- When dealing with problems that have unique structure (such as NBA scoring), it may be prudent to provide your models with more structure, otherwise estimates may end up being not much better than simply taking a MLE.
- The choice of loss function has clear and direct implications for model performance in regards to different performance metrics. When selecting a loss function, you should consider computational efficiency, the performance metrics you care about and the implications of that loss function on the practicality of your model's applications.
- When dealing with a y that is extremely volatile, consider using a substitute or alternative as a proxy.

Acknowledgments

- nbastatR and BettingPros for data
- sklearn and xgboost documentation for model building (as well as CIS 5200 worksheets/HWs)
- https://python.plainenglish.io/nba-betting-using-linear-regression-beba050a50fb
- keras for our neural net/CNN models
- https://machinelearningmastery.com/cnn-models-for-human-activity-recognition-time-series-classification/