Predicting NBA Shots

Brett Meehan
Stanford University
https://github.com/BrettMeehan/CS229_Final_Project
bmeehan2@stanford.edu

Abstract

This paper examines the application of various machine learning algorithms to the problem of predicting the success of shots made by basketball players in the NBA. I will examine the effectiveness of logistic regression, SVMs, Naive Bayes, Neural Networks, Random Forests, and boosting. In this paper, I will show that based on the limited amount of information provided by the features, boosting is the optimal classifier out of the previously stated algorithms.

1. Introduction

My primary motivation for tackling the problem of predicting NBA shots was my desire to tackle an unconventional and challenging problem that wasn't already solved and that didn't have an obvious solution. Although the accuracy of the algorithms described in this paper is probably too low for applications such as sports betting, examining the importance of features could help basketball coaches develop better strategies for maximizing successful shot probabilities.

The main challenge in predicting a shot comes from the limited amount of information provided by the given features. Although there is some lower level information such as the player's distance from the basket and the distance of the closest defender, most of the features consist of higher level features such as whether the game is "home" or "away", time left on the game clock, etc. These higher level features tell us much less about an individual's performance, but there still could be some correlation with shot accuracy (e.g. later in the game the players will be more tired, which could affect their shooting).

There is not a clear accuracy benchmark for this problem since humans can't easily predict the success of a shot based on the given features. A clear minimum accuracy benchmark would be 50%, or simply random guessing. Other possible benchmarks could also be the percentage of the most common result. The most ambitious benchmark I found would be Kaggle user Pablo Castilla's results, as he was able to achieve 68% accuracy using boosting. Achieving something like 90-95% accuracy, however, seems unrealistic given the given the complexity of a shot made by a human and the limited information provided by the features. Simply having an elbow or foot out of place, or being slightly off balance, can affect the outcome of a shot.

The input I used for the machine learning algorithms was a vector consisting of a combination of the following features for each shot: location (home or away game), game outcome (win/loss), final point margin, player shot number, game period, game clock, shot clock, player dribbles, ball touch time, shot distance, points type (2 or 3 point attempt), closest defender, closest defender id, closest defender distance, field goal made, points scored, player name, and player id.

I then used either logistic regression, an SVM, Naive Bayes, a neural network, a random forest, or boosting to categorize the shot as a success or failure (1 or 0). If the output of the algorithm was a probability, I would threshold the values, with anything above 0.5 becoming a successful shot (1) and anything equal to or less than 0.5 becoming a failed shot (0).

2. Related Work

Based on the current literature, the state-of-the-art algorithms used appear to be boosting, random forests, and neural networks.

In "An Investigation of Three-point Shooting through an Analysis of NBA Player Tracking Data", Sliz [4] uses boosting on a richer set of player tracking data captured by the NBA SportVU player tracking system to determine the effectiveness of players attempting three-point shots. Sliz developed custom features based on the raw coordinate data provided by SportVU. Although Sliz does not report the accuracy of his boosting model, he concludes that some of the strongest predictors describe the distance between a shooter and the closest defender. Fortunately, our dataset includes closest defender distance, allowing us to verify this claim. Sliz also suggests that using a neural network could improve performance even further, although based on my ex-

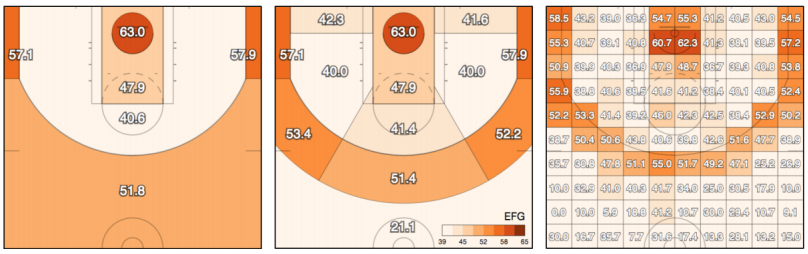

Figure 1. Visualization of shot likelihood from Chang et al.

periments, the dataset would probably have to include more features and the network more complexity to outperform boosting.

Wang and Zemel [5] tackled the problem of automatically classifying NBA offensive plays using the same NBA SportVU player coordinate data, and were able to achieve near perfect classification with a recurrent neural network. While not the same problem my paper is tackling, Wang and Zemel's work shows that it is possible to extract complex relationships just by using the spatial data from NBA SportVU. Although our dataset only uses shot distance and closest defender distance, Wang and Zemel's results mean that adding more spatial data to our dataset could improve accuracy.

Along this vein, Chang et al. [2] demonstrate the importance of spatial data in their diagrams. **Figure 1**, taken from their paper "Quantifying Shot Quality in the NBA", shows how Effective Field Goal percentage (EFG), which essentially measures the likelihood of a shot being made, varies along two dimensions. This paper looks less at machine learning algorithms for NBA shots and more at feature selection/creation.

In their paper "Predicting Shot Making in Basketball Learnt from Multiagent Adversarial Trajectories", Harmon et al. [3] use a convolutional neural network combined with a feed forward network to achieve 61% accuracy on shot prediction. Harmon et al., however, used images as their inputs, as opposed to game statistics. Although processing images is likely slower, an image could also provide extra information about player positioning that would increase the accuracy of the machine learning algorithm. Harmon et al. concluded that the layers of their network were using spatial data such as the location of the ball, offensive, and defensive players in making predictions.

Wright et al. [6] used a factorization machine model to make shot predictions based on 2015-16 NBA data. According to the authors, a factorization machine model is better able to deal with sparse data, and they claim it outperformed logistic regression and SVMs. Wright et al.'s use of factorization machines appears to be a unique and novel approach to the problem of predicting NBA shots. Although I didn't have time to try factorization machines, their ability

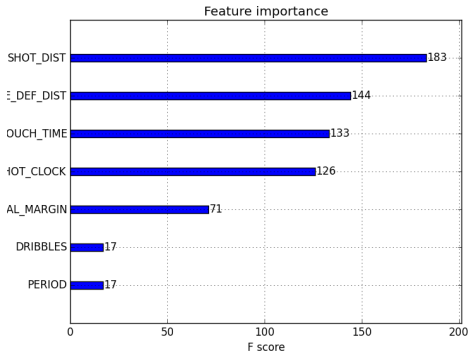

Figure 2. Features ranked by importance with respect to XGBoost model.

to deal with sparse data seems like a valuable strength. I was also impressed by the authors' tables that show the most and least recommended shots categorized by shot style, range, and court location. This paper is probably most similar to mine, although in their results, the authors don't report accuracy, instead reporting root mean squared error (RMSE).

3. Dataset and Features

The dataset this paper uses is from Kaggle and consists of 122502 examples of shots from the NBA 2014-2015 season [1]. The first thing I did was load the raw NBA shot data into Matlab to look at some examples and do some basic analysis. I first noticed that the features "FGM" (field goals made) and "PTS" (points made) were perfect predictors of whether a shot attempt succeeded and therefore removed them. I also converted "GAME_CLOCK" from a time string into seconds. For each of the six machine learning algorithms I tried, I used slightly different preprocessing techniques based on the expected input of each algorithm.

Interestingly, when I looked at feature importance with respect to the XGBoosting model, I found that spatial/distance data appeared to be most predictive of shot success (See **Figure 2**). The top two predictors of shot success were "SHOT_DIST" and "CLOSE_DEF_DIST".

Logistic regression generates a linear set of weights

θ, so I removed categorical variables that had more than two categories. These would include the features "GAME_ID", "MATCHUP", "CLOSEST_DEFENDER", "CLOSEST_DEFENDER_PLAYER_ID", "player_name" and "player_id". This is because when I replaced the categories with integer values, the integer categories had no inherent meaning or trend and would therefore "confuse" a model that is trying to learn linear weights. I removed these same variables for SVMs and neural networks since I figured that using categorical variables would be better suited for algorithms like Naive Bayes, random forest, and boosting.

To preserve temporal information when I removed categorical time variables such as "PERIOD", I sometimes combined the "PERIOD" and "GAME_CLOCK" features into a single feature: "TOTAL_GAME_TIME". The "GAME_CLOCK" feature counts down and resets after each period, but I wanted a feature that was continuous and would distinguish between 10 seconds left in the first period and 10 seconds left in the fourth period, when the players are more fatigued (and maybe more likely to miss a shot). This new feature was used in logistic regression, SVMs, and neural networks.

I normalized continuous data for logistic regression, SVMs, and neural networks to prevent the model from becoming ill-conditioned. In Naive Bayes, I binned continuous data into 4 categories: >1 std dev below mean, 1 std dev below mean, 1 std dev above mean, and >1 std dev above mean. Unfortunately, binning did not seem to make a noticeable difference in my results with Naive Bayes.

To deal with the relative sparsity of the data and high bias most of my models were experiencing, I typically used an approximate 95/5 split for the training and test data. This would give my models more data points to train on and hopefully reduce bias.

When I used a training dev set (such as for neural networks), I typically used a 90/10 split of the training data into a training set and training dev set. Again, my reasoning was that I wanted to train the model on as much data as possible to deal with persistently high bias.

4. Methods

4.1. Logistic Regression

Since a basketball shot has a binary outcome, logistic regression seemed like a logical first step. After removing categorical variables as described in section 3, and changing the response from $\{0, 1\}$ to $\{-1, 1\}$, I performed gradient descent on the empirical loss function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \log(1 + e^{-y^{(i)}\theta^T x^{(i)}})$$

$$= -\frac{1}{m}\sum_{i=1}^{m} \log(g(\theta^T y^{(i)} x^{(i)}))$$

where $g(x)$ is the sigmoid function. Taking the gradient gives us

$$\nabla_\theta J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} \big(1 - g(\theta^T y^{(i)} x^{(i)})\big) y^{(i)} x^{(i)}$$

So our gradient descent rule is therefore

$$\theta := \theta - \alpha \nabla_\theta J(\theta)$$

I had some issues with overly large weights, so I added L_2 regularization to the cost function. The extra added term is $\frac{1}{2}\lambda\|\theta\|$, which becomes $\lambda\theta$ when we take the gradient.

4.2. SVM

After experiencing high bias using logistic regression, I thought that perhaps an SVM would help since it generates extra features using the kernel trick. For this problem, I reused the SVM code from HW2. An SVM tries to maximize the minimum distance from the data points to the margins (support vectors). This optimization can be stated in the following equation, where w and b are the learned weights:

$$\min_{\gamma, w, b} \frac{1}{2}\|w\|^2$$

$$\text{s.t. } y^{(i)}(w^T x^{(i)} + b) \geq 1, i = 1, \ldots, m$$

4.3. Neural Networks

A neural network uses nodes called "neurons" at each layer that learn weights and output a non-linear response. The output of the layers of neurons can be combined to produce a response that essentially represents the combined decision of all the neurons in the network. The weights of each neuron are initialized to small random numbers, and then backpropagation is used to modify the weights in an effort to minimize the cost function.

I thought that using a 1-layer neural network might be able to learn more complex relationships than logistic regression or SVMs due to the non-linear activation functions. Since I chose the sigmoid function as the output activation for my networks, I used the log loss as the cost function:

$$\frac{1}{m}\sum_{i=1}^{m} -y^{(i)} \log p^{(i)} - (1 - y^{(i)}) \log(1 - p^{(i)})$$

Where $p^{(i)}$ is the sigmoid output for example i.
The two networks I tried had 50 units in the hidden layer. Inthe first network, both the hidden and output layers used the sigmoid activation function. The second network used the RELU activation function in the hidden layer and the sigmoid function in the output layer. Both networks also used L_2 regularization.

4.4. Naive Bayes

Naive Bayes is a relatively simple algorithm that makes the assumption that the features x_i are conditionally independent given the response y. In this paper, I used multinomial Naive Bayes since many of the features take on more than two values. As described in section 3, I binned continuous variables into four categories. To make a prediction using Naive Bayes, we simply use Bayes' rule to calculate:

$$p(y = 1|x) = \frac{p(x|y = 1)p(y = 1)}{p(x)}$$

4.5. Random Forests

Random Forests is a bagging algorithm that randomly takes a subset of features, creates an optimal decision tree from those features, then repeats the process again with new subsets of features to eventually create a "forest" of decision trees. When a test feature vector is given to the random forest, each of the decision trees in the forest vote to determine the classification. The mode of the decision trees is then output as the response. One advantage of random forests is that they deal well with categorical data due to the nature of decision trees. They are also robust to overfitting because they use multiple decision trees to classify data.

4.6. Boosting

Boosting is a machine learning approach that takes multiple weak learners, which are classifiers that predict a result better than random guessing but not by much, and combine them using weights into a strong learner. Typically decision trees are used as the learners. A strong learner is a classifier that is arbitrarily well-correlated with the true classification. Since most of the previous algorithms I tried resulted in weak learners (accuracy in the 50-60% range), I thought that boosting might improve these results. This paper uses the XGBoost algorithm created by Tianqi Chen. One notable advantage of boosting is that it not only reduces variance, but also bias.

5. Results

The main performance metric I used was accuracy, which is simply correct guesses divided by total guesses. I don't believe I really overfit to my training data in most cases, but to be safe I added regularization to most algorithms that support it.

Logistic regression was the first algorithm I tried, and although its accuracy wasn't striking (59% on 40k training examples), it still performed better than everything except random forests and boosting (See **Figure 3**). For the learning rate, I used $\alpha = 10$, which was the same rate used in logistic regression for HW2. As mentioned previously, I had to add L_2 regularization to prevent the algorithm from overfitting. The final λ I used was 0.0001, as it performed the best compared to 0.001 and 0.00001. Interestingly, when I looked at the weights θ generated by the algorithm, I noticed that "CLOSEST_DEFENDER_DIST" was negatively correlated with a successful shot. However, this result could make some sense when you consider that low probability shots, such as one from half court, are guarded less closely than high-probability shots like a slam dunk or layup.

	Pred p	**Pred n**	**total**
Actual p	587	1590	2177
Actual n	475	2348	2823
total	1062	3938	

Figure 3. Confusion matrix for logistic regression.

	Pred p	**Pred n**	**total**
Actual p	212	1849	2061
Actual n	406	2533	2939
total	618	4382	

Figure 4. Confusion matrix for support vector machines.

I thought that SVMs would provide a more accurate result with less bias since the kernel trick generates more features, but the SVM algorithm I used only reach a maximum accuracy of 55%, even on 40k training examples. Apparently generating extra features from the subset I used in logistic regression didn't help to reduce bias. These results could be due to the fact that we threw out categorical data, or it could be the case that even all of the given features do not completely describe the factors that determine the likelihood of a successful shot (See **Figure 4**).

For the two neural networks I tested, the optimal accuracy on the training dev set was achieved when 50 hidden neurons were used. During backpropagation, I used regularized batch gradient descent with the same learning parameters as the network we built for HW4: batch size 1000, learning rate $\alpha = 5$, $\lambda = 0.0001$. The entirely sigmoid neural network achieved a maximum accuracy of 56% on the training dev set and 55% on the test set after training on 90k examples (See **Figure 5**). The RELU/sigmoid network achieved a maximum accuracy of 53% on the training dev set and 55% on the test set after training on 90k examples (See **Figure 6**). Although neural networks are good at teasing complex relationships from data, the relationships have to be present in the data in the first place. I believe the fact that I removed categorical data from the input vectors negatively affected my results with neural net-

	Pred p	Pred n	total
Actual p	396	1592	1988
Actual n	683	2329	3012
total	1079	3921	

Figure 5. Confusion matrix for the sigmoid only neural network.

	Pred p	Pred n	total
Actual p	0	2270	2270
Actual n	0	2730	2730
total	0	5000	

Figure 6. Confusion matrix for the RELU/sigmoid neural network.

	Pred p	Pred n	total
Actual p	1	2269	2270
Actual n	0	2730	2730
total	1	4999	

Figure 7. Confusion matrix for Naive Bayes.

works. Naive Bayes performed unremarkably whether I binned the continuous data or not. The greatest accuracy I was able to achieve on the test set was 54% after training on approximately 100k examples (See **Figure 7**). The poor performance by Naive Bayes can probably be attributed to the drawbacks of the Naive Bayes assumption, the simplicity of the model, as well as the relative lack of information present in the features.

The random forest algorithm was the first one to score higher than the 50%'s, with a maximum accuracy of 61% after training on about 100k examples (See **Figure 8**). Matlab's TreeBagger implementation was used, with the optimal parameters after testing being: 80 trees, 50 minimum leaves, and sampling without replacement. The superior performance of the random forest algorithm is probably due to its natural handling of categorical variables, allowing us to make use of some features which we previously had to discard.

The best algorithm by far was boosting using XGBoost with parameter tuning. I was able to replicate Kaggle user Pable Castilla's results of 68% accuracy using approximately 100k training examples and optimized parameters using the GridSearchCV function from the sklearn toolkit

	Pred p	Pred n	total
Actual p	756	1514	2270
Actual n	419	2311	2730
total	1175	3825	

Figure 8. Confusion matrix for random forests.

	Pred p	Pred n	total
Actual p	1040	1875	2915
Actual n	496	2993	3489
total	1536	4868	

Figure 9. Confusion matrix for boosting.

(See **Figure 9**). The optimal XGBoost parameters found were: 1 estimator, learning rate $\alpha = 0.0001$, max depth 3, and a minimum child node weight of 0.0001. Boosting, like random forests, seems to be able to make the most out of categorical data that other algorithms couldn't deal with, in addition to having an intrinsic ability to deal with bias.

6. Conclusion

After running the six previously mentioned machine learning algorithms, boosting appears to be the clear winner with the limited amount of information provided by the dataset features. The strongest classifiers were boosting and random forests, both of which are ensemble algorithms that aggregate their results to make a more accurate prediction than any individual part. Both of these algorithms also seemed to be particularly suited to the dataset because of their ability to smoothly handle categorical data.

This paper's results also appear to confirm the status of random forests and boosting as state-of-the-art for shot predictions. Although neural networks performed poorly in this problem, I believe that with more spatial data, such as the SportVU player coordinate data used by Sliz, neural networks could become effective shot predictors. Chang et al.'s diagrams, as well as Wang and Zemel's encouraging results, suggest that there is valuable information to be gleaned from such two-dimensional spatial data.

References

[1] Kaggle nba shot logs. `https://www.kaggle.com/dansbecker/nba-shot-logs`, 2016.

[2] Y. Chang, R. Maheswaran, J. Su, S. Kwok, T. Levy, A. Wexler, and K. Squire. Quantifying shot quality in the nba. *MIT Sloan Sports Analytics Conference*, 2014.

[3] M. Harmon, P. Lucey, and D. Klabjan. Predicting shot making in basketball learnt from adversarial multiagent trajectories. 2016.

[4] B. Sliz. An investigation of three-point shooting though an analysis of nba player tracking data. 2016.

[5] K. Wang and R. Zemel. Classifying nba offensive plays using neural networks. *MIT Sloan Sports Analytics Conference*, 2016.

[6] R. Wright, J. Silva, and I. Kaynar-Kabul. Shot recommender system for nba coaches. *KDD Workshop on Large-Scale Sports Analytics*, 2016.