Predicting the Final Score of Major League Baseball Games

Nico Cserepy cserepy@stanford.edu
Robbie Ostrow ostrowr@stanford.edu
Ben Weems bweems@stanford.edu

***Abstract*—We extracted the result of every MLB plate appearance since 1921. These data allowed us to calculate arbitrary batter and pitcher statistics before every game. Leveraging these data, we simulated individual baseball games. We found that our simulations – which incorporated only data available before the start of the game – were a better predictor of true scores than the accepted state-of-the-art: over/under lines in Las Vegas. In fact, a bet based on the techniques described in this paper is expected to yield profit.**

***Index Terms*—Multinomial Logistic Regression, Softmax Regression, Baseball, Scores, Markov Chain, Betting, MLB**

I. Introduction

BASEBALL has always been known as America's favorite pastime. It is also a favorite speculation for the ambitious gamblers of Las Vegas.

This sport is unique in how much data is available online. Statistics on every plate appearance in Major League Baseball (the MLB) since 1921 are available for free. Given this abundance of data, we were interested to see whether we could improve upon odds' makers predictions about the over/under of a baseball game (also known as total).

An accurate simulator would be valuable to teams looking for a slight edge in managerial decisions. MLB managers, as of 2015, cumulatively make over $100 million to make decisions throughout a game. In a league where the most marginal improvement can make a difference between the World Series and elimination, every pitching change matters.

The input to our algorithm is the roster of the home team, the roster of the away team, statistics of all players on each team leading up to a game, and all metadata about the game, like the stadium or field conditions.

The output of our algorithm is simply a tuple (s_a, a_h) where s_a is the predicted score of the away team, and s_h is the predicted score of the home team.

Rather than attempt to predict the total scores of baseball games by feeding metadata about the game into a learning algorithm, we chose to simulate the game on a play-by-play basis. In order to do so, we needed to calculate the probability of each outcome of an at-bat.

The input to this sub-problem is the current batter's statistics, the current pitcher's statistics, the current game state, and metadata about the game. We ran multinomial logistic regression on these features to assign probabilities to each possible outcome of a plate appearance, like "single," "double," or "hit-by-pitch."

This project was done partially in conjunction with CS221. There was significant data-massaging that had to be done, and the database that we built was shared among classes, along with the general infrastructure for game simulation.

II. Related Work

Due to baseball's popularity and its fans' penchant for statistics, there has been substantial work published on the modeling of baseball games. The Markov chain model is well-suited for scenarios we encounter in baseball since each play can be considered a distinct state, and transition probabilities from that state can be estimated given historical data.

In 1974, Freeze used a Markov model to analyze the effect of batting order on runs scored [1]. His analysis was likely one of the first that used such techniques to simulate a baseball game. He found that batting order does not significantly effect game outcomes, but his simplifying assumptions were significant enough to raise questions about his results. However, his techniques provided a useful model for future research.

Pankin attempted a more sophisticated version of Freeze's research to optimize batting orders in [2] by attempting to maximize the number of runs per game. He used a Markov process to simulate games, but did his research in 1991 and thus did not have the resources to calculate sophisticated transition probabilities. He assumed that batters hit the same in all situations, and had only one season of data to work with. His methods were primitive compared to what is feasible today, but his was the first paper that proved that Markov processes can be leveraged for an advantage in baseball.

Bukiet and Elliote take Pankin's attempts a step further, with more computing power and statistics, and attempt to simulate baseball games using a small set of offensive probabilities [3]. This paper leveraged more advanced models, but ignored what is probably the biggest predictor of runs scored: pitching statistics. However, [3] suggests that their methods could be extended to include pitcher statistics.

Humorously, perhaps the most relevant paper is about calculating realistic probabilities for a board game. In [4], Hastings discusses the challenges and techniques he used to help design the probabilities of the Stat-O-Matic board game, a baseball simulation run with dice. Baseball simulation games were very popular in the 1990s, and players wanted a realistic experience. The insights in this paper about interactions between batter statistics and pitcher statistics were key for designing our algorithm. Luckily, we didn't have to roll several dice for every at bat, though.

III. Dataset and Features

To convey our need for a large volume of data, we have included our feature templates:

Binary feature templates. Each of these features assumes a value in $\{0, 1\}$. Brackets signify where more than one possible feature exists. For example, there is a separate binary feature for `man_on_base_2` and `man_on_base_3`.

{0, 1, 2}_out	{1, 2, ...}_inning
man_on_base_{1, 2, 3}	{0, 1, 2...}_stadium
{0, 1, 2...}_precipitation	{0, 1, 2...}_wind_direction
{0, 1, 2...}_field_condition	batter_and_pitcher_same_handed

Real-valued feature templates. Each of these features takes on a real value in $[0, 1]$. $\{e\} \in \{$`Generic Out, Strikeout, Advancing Out, Walk, Intentional Walk, Hit by Pitch, Error, Fielder's Choice, Single, Double, Triple, Home Run`$\}$.

`bat` signifies a batter's statistic, `pit` signifies a pitcher's statistic, and `limit` signifies that only the last 30 events for that player in that category were considered. For example, `single_bat_vs_diff_handed` is the number of singles per plate appearance against a different handed pitcher in this batter's career, and `double_pit_totals_limit` is the fraction of doubles given up by the pitcher in the last 30 at bats.

{e}_bat_vs_same_handed	{e}_bat_vs_same_handed_limit
{e}_bat_vs_diff_handed	{e}_bat_vs_diff_handed_limit
{e}_bat_totals	{e}_bat_totals_limit
{e}_pit_vs_same_handed	{e}_pit_vs_same_handed_limit
{e}_pit_vs_diff_handed	{e}_pit_vs_diff_handed_limit
{e}_pit_totals	{e}_pit_totals_limit

In total, these feature templates generate 365 features. We believe that these features provide a state-of-the-art description of an arbitrary plate-appearance, and they provide deeper knowledge than any model found in the literature.

To calculate these features, we extracted play-by-play data from retrosheet.org for all regular-season baseball games since the 1921 season [5]. We focused on the data most relevant to modern day by gathering statistics for players who were active in the 1980-2014 range. However, players whose careers started before 1980 have their career statistics stored.

There were about 7.2 million plate appearances between 1980 and 2014. We needed many statistics about each batter and each pitcher specific to the day of each game, but gathering these data in real-time would be infeasible. Gathering the data during each the simulation would be too slow, so we had to precompute all of these statistics. In order to perform accurate simulations, every player needed up-to-date statistics for every game. We pre-processed the play by play data to calculate all relevant statistics for every player in the starting lineups of every game between 1980 and 2014. This database ended up being about 50 gigabytes, but proper indexing provided a significant speedup relative to recalculating features every simulation.

Along with batter and pitcher statistics leading up to the game, we also had features that depended on the game state. Statistics such as the number of outs, the inning, or the handedness of the pitcher might affect the outcome of some plate appearance. Since these features could, by definition, not be precomputed, we added them to the feature set during each simulation.

Unfortunately, an example of the play-by-play data is too large to be displayed here. However, we used Chadwick [6] to help parse the data from retrosheet.org, and the approximate schema of the original play-by-play data can be found at [6][1].

To train our classifier, we randomly sampled one million events along with their results[2] from the 7.2 million at-bats available after 1980. We also sampled 300,000 at-bats as a "test set." In our case, since there is no ground truth, analyzing success on a test set is tricky. Rather than a percentage "correct," we calculate the log-likelihood of our whole test set given the trained classifier. This log-likelihood can be compared to log-likelihoods of other classifiers to determine which is superior.

IV. Methods

Baseball games are sufficiently random and complex that it is not reasonable to model a game as a single vector of features and expect that any machine-learning algorithm will be able to give accurate predictions. Instead, we model the game on a state-by-state basis by running Monte Carlo simulations on a Markov Decision Process (MDP).

To do so, we discretize each baseball game into states that represent plate appearances. This requires that we assume that nothing happens between the end of two plate appearances. As such, our simulation assumes no steals, passed balls, pick-offs, or other events that might happen during an at-bat. While this may seem like a large assumption to make, these plays turn out to have very little bearing on the score of the game since they happen comparatively infrequently [7].

Baseball is a unique sport in the sense that the beginning of each play can be considered a distinct state, and relevant information about this state can be easily described. In our program, we define each state to contain the following information: score of each team, which bases are occupied, the statistics of the baserunner on each base, the current batting team, the number of outs, the current inning number, statistics of the player at bat, statistics of the pitcher, and auxiliary information. "Auxiliary information" is miscellaneous information that does not change from state to state, like the ballpark, home team, etc.

Let `action` be any way a batter can end his at-bat. From each state, there are up to 12 possible actions: $\{$`Generic Out, Strikeout, Advancing Out, Walk, Intentional Walk, Hit by Pitch, Error, Fielder's Choice, Single, Double, Triple and Home Run`$\}$. Some actions are not possible from some states. For example, "Fielder's Choice" is not possible if there are no runners on base. From each state, we can calculate the probability of each possible action: $P(action|state)$. This calculation of $P(action|state)$ is fundamentally the most difficult and important part of the algorithm.

[1] See http://chadwick.sourceforge.net/doc/cwtools.html

[2] Computer ran out of RAM using more than this, but we could feasibly train on more events.

Once we have calculated the probability of each action, we take a weighted random sample. Having chosen an action, we can calculate the probability of each outcome given that action from the historical data and again take a weighted sample. For this step, we only take into account the positions of the baserunners and the action chosen. For example, $P(first\_and\_third, no\_runs\_scored, no\_outs\_made | first, single)$ is asking the question, "what's the probability that, after the play, there are players on first and third, no outs are made, and no runs are scored, given that there was a baserunner on first and the batter hit a single?" (The answer is $\frac{64253}{231220}$, or a little less than 28%.) These probabilities are pre-computed to make the simulation run quickly, so they do not depend on the batter or pitcher. Since there are a large number of states and transitions, many of which have to be computed on the spot, we cannot specify a full MDP. Instead, we run Monte Carlo simulations on the states and transitions specified above. So, at a very high level, to simulate a single game once, we do the following:

1) Enter start state (Away team hitting, nobody on base, etc.)
2) Repeat until game end
   a) Calculate $P(actions|state)$
   b) Choose weighted random action
   c) Calculate $P(outcomes|action)$
   d) Choose weighted random outcome
   e) Go to the state that outcome specifies
3) Gather statistics about simulation

In order to achieve statistical significance, we simulate each game many times. The key to simulating accurate games is learning $P(actions|state)$.

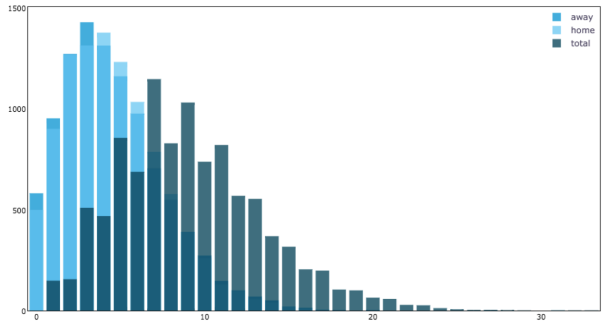

Figure 1: This histogram represents the scores predicted by 10,000 simulations of a Mets-Cardinals game. The median total score of the simulations is 8. The actual total score of the game was 9.

A. Calculating P(actions|state)

Recall that we have one million training examples each consisting of 365 sparse features, all labeled with their outcome.

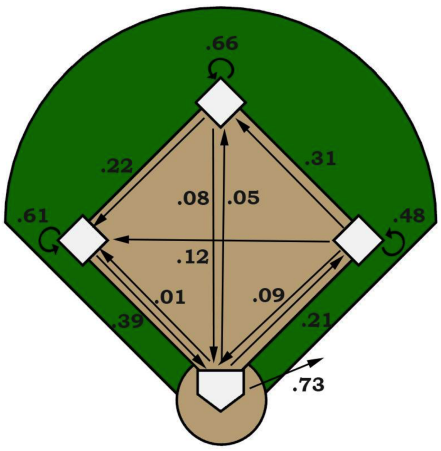

Figure 2: This figure demonstrates what the transition probabilities from a given state to the next state given an action might be. Note that these arrows do not all represent the same player: the transitions from home plate are for a player at-bat, while those from the bases represent transition probabilities given some action the batter takes (e.g. single).

In order to estimate these probabilities, we train these million examples on a multinomial logistic regression, or softmax regression, classifier.

Softmax regression is a generalization of logistic regression to n-class classification problems. We assume that the conditional distribution of y given a feature set x is

$$p(y = i|x; \theta) = \frac{e^{\theta_i^\top x}}{\sum_{j=1}^{k} e^{\theta_j^\top x}} \tag{1}$$

To learn the parameters θ of the model, we can maximize the log-likelihood:

$$\ell(\theta) = \sum_{i=1}^{m} \log \prod_{l=1}^{k} \left(\frac{e^{\theta_k^\top x^{(i)}}}{\sum_{j=1}^{k} e^{\theta_j^\top x^{(i)}}} \right)^{1\{y^{(i)}=l\}} \tag{2}$$

To solve this problem, we leveraged scikit-learn's implementation of the Broyden–Fletcher–Goldfarb–Shanno algorithm [8] [9]. BFGS approximates Newton's method. The algorithm approximates the Hessian instead of calculating it directly, which makes the problem more tractable (training 10GB of data on a laptop is no easy task...)

Once we know the parameters, it is trivial to calculate the probability of each action using equation 1 and substituting the results in for $p(action|state)$.

V. Results and Discussion

We quantified the success of our complete system based on two metrics: A) using the binomial test to quantify the probability of meaningful results relative to random selection of Vegas over/under and B) Return on Investment (ROI). We then analyze the success of C), our softmax algorithm and D), our simulation accuracy.

A. Binomial Test

Prediction Confusion Matrix

Predictions \ True Result	Over	Push	Under
Over	$c_i = 1$, $d_i = 0$	$c_i = 0$, $d_i = \alpha_i$	$c_i = 0$, $d_i = 0$
Push	No Bet	No Bet	No Bet
Under	$c_i = 0$, $d_i = 0$	$c_i = 0$, $d_i = \alpha_i$	$c_i = 1$, $d_i = 0$

Table I: Confusion Matrix that determines how to set the variables c_i and d_i based on a prediction and a result.

To calculate significance relative to random selection, we used a one-tailed binomial test. Our null hypothesis H_0 is that the probability of success (π) is 0.5, and our alternative hypothesis H_1 is $\pi > .5$. We evaluated our number of successes (k) out of total non-push[3] game attempts (n) to measure significance (p). We define c_i to be 1 if the game was correctly identified and 0 if the prediction is incorrect, and set d_i equal to the amount bet if the result is a push, otherwise it is set to 0.

$$k = max(\sum_{i=0}^{n} c_i, \sum_{i=0}^{n}(1 - c_i)) \tag{3}$$

$$p = Pr(X \geq k) = \sum_{i=0}^{k} \binom{n}{i} \pi^i (1-\pi)^{n-i} \tag{4}$$

Using our model, we correctly identified over or under on 4595 out of 9024 non-push games, which is over 50% but not statistically significant. If we focus on games in which we were 80% confident of our prediction, we find that we correctly identified 2322 of 4500 non-push games. Once again using Equation 3, we find that $p = 0.020$, providing statistical significance at $p < 0.05$. This 80% confidence is not hand-picked as the best confidence; as we see in Figure 3, we have statistical significance of $p < 0.05$ at 60% confidence and higher. The statistical significance decreases when we reach a high confidence threshold because fewer games are considered making achieve high confidence more difficult.

[3]A "non-push" game attempt is defined as a game in which the total runs scored does not exactly match the Vegas prediction. If the runs scored equals the prediction, all money bet would be returned and this is called a "push."

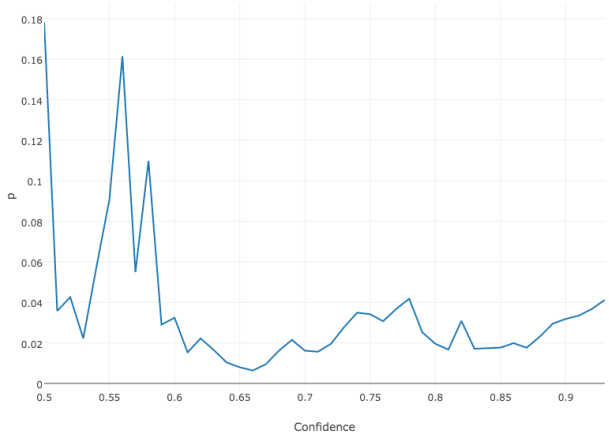

Figure 3: Confidence vs. Statistical Significance. The y-axis provides the p value of statistical significance for a one-sided binomial test, or likelihood that our results would be obtained if the null hypothesis $Pr(c_i) = .5$ were true. The x-axis gives the confidence we had in our predictions.

B. ROI

To calculate ROI, we first converted odds (ω) to payout (ϕ) for a given bet amount (α) by using the equation:

$$\phi = \begin{cases} \alpha * (1 + \omega/100) & \omega > 0 \\ \alpha * (1 + 100/\omega) & \omega \leq 0 \end{cases} \tag{5}$$

In our simulations, we always set $\alpha = 1$, but there is potential to alter the bet amount conditional on confidence.
We can then use Equation 5 to calculate the ROI by subtracting the bet-amount α and adding the payout ϕ if the prediction is correct. We get the total return (R), and divide by the amount bet (A) to get the ROI (σ).

$$R = \sum_{i=0}^{n} c_i \phi_i + d_i \tag{6}$$

$$A = \sum_{i=0}^{n} \alpha_i \tag{7}$$

$$\sigma = \frac{R}{A} = \frac{\sum_{i=0}^{n} c_i \phi_i}{\sum_{i=0}^{n} \alpha_i} \tag{8}$$

ROI can easily be converted to Percent Expected Return (PER, β) on a bet as below:

$$\beta = 100 * (1 + \sigma) \tag{9}$$

In Figure 4, we see the PER at each range of confidence intervals compared with pseudo-random predictions of over or under for those games, accomplished by pseudo-randomly setting c_i to either 0 or 1 in Equation 8. A confidence interval is determined based on the fraction of simulations of a game

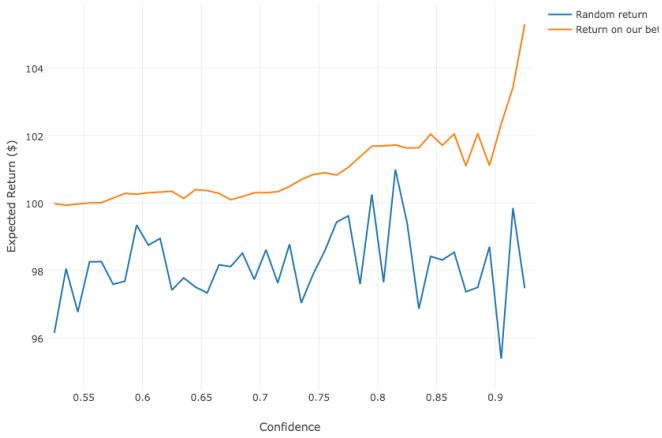

Figure 4: The percent expected return, smoothed across confidence intervals of 5% ($\gamma \pm 2.5$). Confidence is the fraction of simulations with the predicted result (over/under the total). 'Random' pseudo-randomly assigned over/under to the games in that confidence interval.

which give the predicted result. Confidence (γ) is calculated as follows:

$$\gamma = \frac{max(\sum_{i=0}^{n} c_i, \sum_{i=0}^{n}(1 - c_i))}{n} \tag{10}$$

Our lowest PER is found at the lowest confidence shown: 52.5%. This gives $\beta = 99.94\%$. This value continues to rise as our confidence increases, while the pseudo-random return stays constant near 98%.

C. Softmax Regression

Our results for softmax regression are probabilistic, so we worked to optimize the log-likelihood of its fit. It is difficult to assign a score of accuracy because a feature set does not belong to any specific label – its probability of each label is exactly what we are seeking. Instead of a traditional measure of accuracy, we say that our classifier is optimal if the log-likelihood of its fit to our test set is greater than any other classifier's.

We did not have the computing resources to train many classifiers on a million features then predict on 300,000, or even to cross-validate in a similar manner, but we did make two important discoveries. First, the removal of pitching data from the features significantly decreases the likelihood of the test set, as expected. Second, the learned probabilities performed very well in the larger scope of the baseball simulations, which suggests that the fit was at least moderately good. Additionally, we optimized the inverse of regularization strength to balance between strong regularization and weak regularization.

D. Simulations

To determine the accuracy of our simulations, we relied on years of baseball knowledge among team members. Figure 1 displays a histogram with scores from game simulations which presents an average score between seven and nine runs, which is in line with the 2013 MLB average of 8.33 runs, which we calculated. This, along with topical knowledge among group members, tells us that the simulations are running realistic game paths with realistic results for games.

VI. Conclusion and Future Work

After extracting the results of every MLB plate appearance since 1921, and simulating thousands of games as Markov chains using a multinomial logistic regression model, we generated a state-of-the-art tool for predicting outcomes of Major League Baseball games.

While we are very pleased with our results, we regret that we did not have the capability to formally analyze the performance of our softmax regression versus other possible classifiers and feature sets. We varied scikit-learn's parameters for logistic regression and found the choice that maximized the log-likelihood of the test set, but did not have the computational power to cross-validate or perform forwards or backwards-search, as each design iteration took hours to days of computer time. While our results show that the classifier was effective, we hope to formalize that notion on future iterations of this project.

Simulations of games provide the opportunity to examine many more statistics than only total score. The granularity provided yields a powerful tool for coaches and bettors alike.

References

[1] R. A. Freeze, "An analysis of baseball batting order by monte carlo simulation," *Operations Research*, vol. 22, no. 4, pp. 728–735, 1974.

[2] M. D. Pankin, "Finding better batting orders," *SABR XXI, New York*, 1991.

[3] B. Bukiet, E. R. Harold, and J. L. Palacios, "A markov chain approach to baseball," *Operations Research*, vol. 45, no. 1, pp. 14–23, 1997.

[4] K. J. Hastings, "Building a baseball simulation game," *Chance*, vol. 12, no. 1, pp. 32–37, 1999.

[5] D. Smith *et al.*, "Retrosheet website."

[6] T. L. Turocy, "Chadwick: Software tools for game-level baseball data." [Online]. Available: http://chadwick.sourceforge.net

[7] P. Beneventano, P. D. Berger, and B. D. Weinberg, "Predicting run production and run prevention in baseball: the impact of sabermetrics," *Int J Bus Humanit Technol*, vol. 2, no. 4, pp. 67–75, 2012.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] D. F. Shanno, "On broyden-fletcher-goldfarb-shanno method," *Journal of Optimization Theory and Applications*, vol. 46, no. 1, pp. 87–94, 1985.

[10] H. Cooper, K. M. DeNeve, and F. Mosteller, "Predicting professional sports game outcomes from intermediate game scores," *Chance*, vol. 5, no. 3-4, pp. 18–22, 1992.

[11] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.