Baseball as a Zero-Sum Stochastic Game

Binyamin Friedman[1], **Yevgeniy Vorobeychik**[2]
[1]Brandeis University
[2]Washington University in St. Louis
binyaminf@brandeis.edu, yvorobeychik@wustl.edu

Abstract

We start with an approach to modeling a full baseball game as a series of transitions in a zero-sum stochastic game, parameterized by pitcher and batter representations. We address some practical considerations about the usage and efficiency of such a model. Then, we demonstrate how a working stochastic model can be used to approach more complex baseball analysis, such as optimizing a team's batting order. Our system is well-documented and can be easily extended.

Code — https://github.com/BOBONA/SolvingBaseball/

Background

This project extends the previous work of Connor Douglas (2021), who modeled individual at-bats as zero-sum stochastic games. Although we approach our data model with similar assumptions, some key changes in our implementation allow for a more extensive model of the game. In the process of recreating the project's code base, we greatly reduced code redundancy and complexity. We moved from using Paul Schale's Kaggle dataset to directly fetching from MLB's Statcast data, which is the most up-to-date source for pitch-level baseball data. A single class is responsible for processing raw data into our Pitch objects and aggregating that data to generate our Pitcher and Batter objects. For convenience, we load and store our data in a single object, allowing us to iterate on our methods much more easily. In general, many efforts were made to make our code user-friendly and open to modification.

Breaking Baseball Down

In a baseball game, two teams of nine players alternate between batting and fielding. The basic play of the game is an at-bat, in which the fielding team's pitcher faces off against a player in the batting team. The pitcher throws the ball towards the batter, who must hit it successfully into the field. A batter's goal after hitting the ball is to score by running counterclockwise around the field. A batter is out when they either fail to hit the ball, their ball is caught from the air by a fielder, or they are tagged out with a fielder possessing the ball. A batter is safe while on one of the three bases, but ultimately must advance in order to allow the next runners through. Once the pitcher gains possession of the ball, the batter must reach a base safely and stop running, at which point the next batter in the cycle (determined at the start of the game) repeats the process. The two teams switch roles once three outs have occurred. A game is made up of nine innings, nine turns batting for each team in total.

When facing a pitch, a batter has less than an instant to decide whether to swing or whether to take (not swing). The possible outcomes are as follows: If the batter takes, then the count changes depending on where the pitch passes the batter. If the pitch passes through the strike zone (defined according to the batter's knees and shoulders), then one strike is added to the count. Otherwise, one ball is added to the count. Three strikes is an out, and four balls is an automatic walk to first-base. If the batter swings, then there are four possible outcomes. A miss results in a strike. A foul, in which the batter hits the ball out of bounds, also results in a strike, but will not end the at-bat. An out occurs when the batter hits the ball but cannot reach first-base successfully. The batter's objective is to ultimately score a hit, reaching first-base successfully before three strikes.

Modeling States and Actions

In translating the rules of baseball into the transitions of a zero-sum stochastic game, some major simplifications are inevitable. Since the stochastic game model requires finite states, we must decide how to limit and discretize each variable. Our first level of simplification is in defining the possible game states along with each teams' actions. To start, let's focus on the pitcher-batter interaction.

We simplify the pitcher's actions into combinations of pitch type and pitch zone. While the MLB defines 14 different pitch types, many of which are rarely used, we simplify by grouping all pitches into the 6 most common types: four-seam fastball, two-seam fastball, slider, changeup, curveball, and cutter. We divide the pitching area into 19 zones, as shown in Figure 1. The particular discretization is not crucial, but will ultimately involve a trade-off between speed and accuracy. The batter's actions are simply to hit or to take. We define our game state with the current inning, the number of outs, which batter in the cycle is currently up to bat, the number of balls, the number of strikes, whether first base is occupied, whether second base is occupied, and whether third base is occupied. This information is the minimum amount needed to transition through the game on a

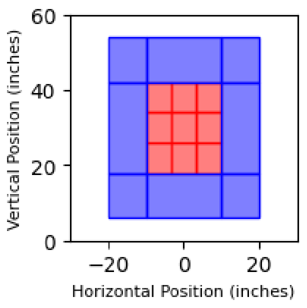

Figure 1: Red represents the strike-zones. The numbers shown are used as common default zone measurements. Note that the blue zones extend forever.

pitch-by-pitch level. A summary is shown below.

$$\text{batter actions} = \{\text{hit}, \text{take}\}$$

$$\text{pitcher actions} = \underset{0-5}{\text{type}} \times \underset{0-18}{\text{zone}}$$

$$\begin{aligned}\text{game states} = {} & \underset{0-8}{\text{inning}} \times \underset{0-2}{\text{outs}} \times \underset{0-8}{\text{batter}} \times \\ & \underset{0-3}{\text{balls}} \times \underset{0-2}{\text{strikes}} \times \underset{0-1}{\text{first}} \times \underset{0-1}{\text{second}} \times \underset{0-1}{\text{third}}\end{aligned}$$

Learning Important Distributions

The second level of simplification in our model is in the distribution of transitions between states. It's clear from the start that this distribution must rely on empirical data. When the batter takes, the outcome is deterministic; A pitch in the strike-zone results in a strike, otherwise a ball is added to the count. When the batter swings, the outcomes are stochastic; Either the batter misses, fouls, hits, or hits with an out. We must also decide how to model on-field interactions given these outcomes, particularly when the ball is hit. In the context of our stochastic model, two more factors complicate this distribution. First, pitchers do not have perfect accuracy, so the pitch that flies to the batter is influenced but not decided by the pitcher's intent. Second, while the stochastic game assumes that players choose their actions simultaneously, in reality the batter's decision to swing is influenced by the outcome of the pitch. With these considerations in mind, we introduce networks to learn the swing outcome distribution, the pitcher control distribution, and the batter patience distribution.

The Swing Outcome Distribution

The swing outcome distribution for a given pitch is determined by the pitcher and batter, along with the current state of the game. We follow Douglas's player representations. Pitcher representations are matrices of size $5 \times 5 \times 12$, representing normalized average speed and relative throwing frequency across each zone and pitch type. This allows us to capture spatial relationships at the cost of some slight redundancies, since the larger edge zones result in repeated values. Batter representations are similar, except the two statistics

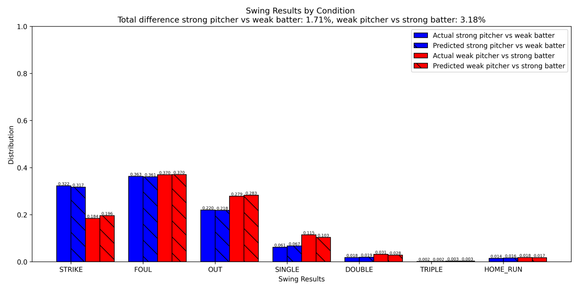

Figure 2: The model captures how skill levels affect outcomes. We classify weak and strong players according to their empirical OBP.

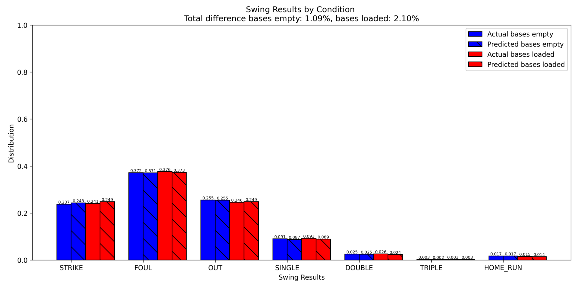

Figure 3: The model can also capture very subtle differences. Performance seems to improve when bases are loaded (a runner on each base).

captured are batting average and relative swinging frequency (across each kind of pitch). Pitches are represented as a one-hot encoding of possible pitch types. We run the pitcher, batter, and pitch individually through a series of convolutional layers before concatenating all of them together with the number of strikes, balls, outs, and values for whether a runner is on first, on second, and on third. This information is passed through several dense layers before a softmax layer transforms the output to a probability distribution over the possible swing results. We apply some dropout to the pitcher and batter representations before and after the set of convolutional layers to prevent overfitting.

We ignore the intricacies of how different hits interact with fielders, and opt for a simple set of swing results: strike, foul, out, single, double, triple, and home-run. Figures 2 and 3 demonstrate the efficacy of this approach.

The Pitcher Control Distribution

Pitcher control cannot be learned regularly from raw data, as this would require knowledge of the pitcher's intention for each pitch. Instead, we assume consistency on a high stakes but uncommon scenario, the 3-0 count (3 balls, 0 strikes). We fit a bivariate Gaussian distribution to arrive at empirical estimates of a pitcher's control for each pitch type. Since this scenario is quite uncommon, many pitchers will lack enough data to accurately fit a distribution, so we train a simple network to estimate the distribution parameters $\{\mu_x, \mu_y, \sigma_x^2, \sigma_y^2, \sigma_{xy}\}$ given a pitcher and each pitch type. We use our standard pitcher representation and a one-

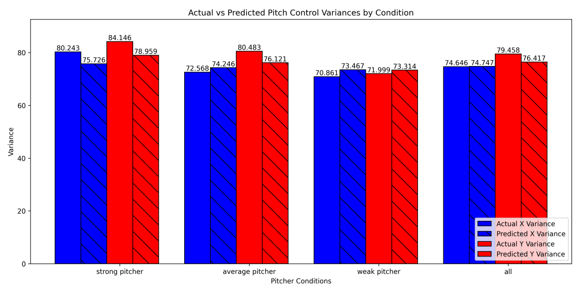

Figure 4: The model also fits mean and covariance, but these parameters are less relevant. Units are in in^2.

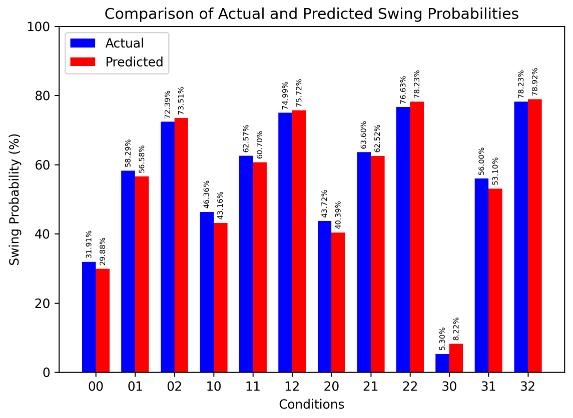

Figure 5: The current number of balls and strikes greatly impacts the batter's patience.

hot encoding of the pitch type. This network is almost the same as the swing outcome model, except here the batter representation has no relevance. The model has limited training data but still learns how to estimate variance within an acceptable margin. See Figure 4.

Once we are able to estimate a normal distribution for any given pitcher and pitch type, we can use random sampling to estimate the outcome zone distribution given any intended pitch. We use the center of a pitch's intended zone as the mean of the estimated distribution. For non-strike zones, whose edges go to ∞, the centers must either be defined manually or kept as is, such that pitches thrown at a non-strike zone always end up in a non-strike zone.

The Batter Patience Distribution

A batter's tendency to hold back from swinging on a pitch is known as their patience. To capture this reaction, we train a model to learn the distribution of swings given a batter, a pitch, and the current count. We are only concerned with this effect with regards to borderline pitches; For far out balls, we will later assume that the batter will never swing. We define borderline pitches as pitches that fall within the "shadow zone," 3 inches, or a baseball's width, of the edge of the strike zone. Our network for learning the batter patience distribution is very similar to our other networks, except here the pitcher representation has no relevance. Figures 5 and 6

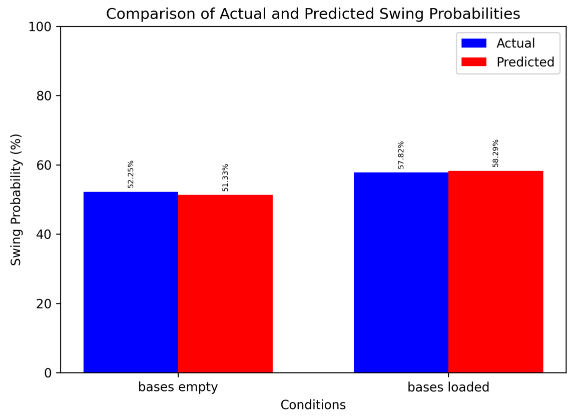

Figure 6: Unsurprisingly, batters are more likely to swing when bases are loaded and the potential reward is much higher.

show how this network can capture the nuances of the distribution.

Obtaining a Transition Distribution

To obtain a transition distribution, we piece together our three learned distributions. We use the pitcher control distribution to map from an intended pitch to a distribution of outcome zones. If the batter takes, then the outcome depends on the location of the pitch, either a *called ball* or a *called strike*. If the batter swings, then we use the swing result distribution to obtain a distribution of *strike*, *foul*, *out*, *single*, *double*, *triple*, and *home-run*. However, if the pitch ends up in a non-borderline ball zone, then we override the batter's decision to swing and always take. If the pitch ends up in a borderline zone, we override the batter's decision to swing with our learned batter patience distribution. See Figure 7 for a visualization of this process. Note that we need to calculate distributions for each batter in the lineup and use them accordingly.

The rules for modifying the game state follow standard baseball rules. A *called ball* adds a ball to the count. A *called strike* or *swinging strike* adds a strike to the count. A *foul* also adds a strike to the count but will not end an at-bat. Three strikes is an out. An *out* ends the at-bat and adds an out to the count. Four balls is an automatic walk, which ends the at-bat and moves all batters one base. We simplify on-field interactions and automatically move all batters one, two, three, or four bases when a batter hits a *single*, *double*, *triple*, or *home-run* respectively. Each at-bat moves the batting cycle forward by one. Three outs ends the inning, which adds one inning to the count and resets. The game ends after nine innings.

Since no state has more than 8 transitions, we implement this as a fixed-size adjacency list for each state-action pair combination. This way, the whole transition distribution method can be vectorized for efficiency.

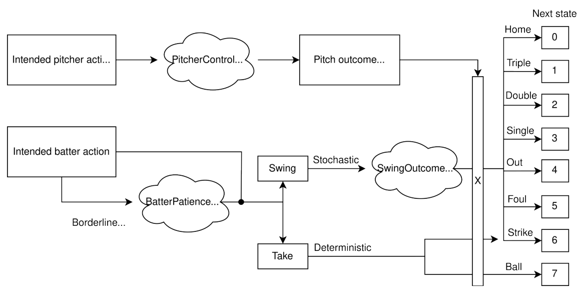

Figure 7: We visualize the flow of our transition distribution.

Solving the Stochastic Game

The transition distribution is the defining characteristic of our stochastic game. Littman (1994) presents an approach for solving finite zero-sum games, using value iteration and linear programming. Value iteration converges on values V_s for each state s, along with a mixed strategy for the pitcher. For $\forall s \in S$, the policy $\pi_s(a)$ denotes the optimal distribution of pitcher actions $a \in A$. A mixed strategy is necessary because a deterministic strategy is generally not appropriate for stochastic games (think Tic-Tac-Toe).

We begin by initializing $V_s^{(0)}$ to random values. Then, while changes are greater than some prespecified threshold β , we update each V and π for each s according to the following recursive relationship.

$$V_s^i = \min_{\pi_s} \max_{o \in O} \sum_{a \in A} \Big[\pi_s(a) \cdot Q_s^{(i)}(a, o) \Big] \tag{1}$$

where $Q_s(a, o)$ denotes the *quality* of action a against action o,

$$Q_s^{(i)}(a, o) = \sum_{s' \in S} \Big[T_s(a, o, s') \cdot \Big(r_s(s') + V_{s'}^{(i-1)} \Big) \Big] \tag{2}$$

Here, $r_s(s')$ denotes the immediate reward for transitioning from state s to s', or in other words, the amount of runs scored. $V_{s'}$ can be viewed as a measure of long-term reward. That means the *quality* of an action-pair at a state is a complete measure of the value of that action. In our game, the pitcher's goal is to minimize the maximum expected value, which is expressed clearly in our equation for V_s^i. We can solve Equation 1 with linear programming.

V will converge after a certain number of iterations. Since we define our reward function r as the number of runs scored, V_s represents the expected number of runs that will be scored from s onwards. Using baseball terminology, V_0 (where 0 is an initial state) represents the pitcher's expected run average (ERA) against the given lineup.

Small Improvements on Value Iteration

While the most basic formulation of value iteration works effectively, solving linear programs is a very time consuming task. Luckily, we can make some simple changes to drastically lower the amount of computation necessary for convergence.

1. Each out's values are dependent only on the following out. Therefore, we can reach a correct solution by first iterating only the 2-out states of the 9th inning to convergence, then the 1-out states of the 9th inning, all the way down to the 0-out states of the 1st inning. This is more efficient than iterating over all game states at once, because earlier out states are not able to converge to their final values before later out states.
2. Within a set, when we iterate each state simultaneously as implied in our definitions above, values take a long time to propagate down to the initial states (0 strikes, 0 balls, empty bases). These initial states cannot converge before later states have converged. We use our knowledge of the structure of baseball's transition graph to order the states within an inning, such that at-bats with more players out, later in the batter cycle, with runners farther on the bases are iterated before "earlier" at-bats with less outs and less players on base. Within an at-bat, we iterate states with more balls and strikes before states with less. Then, instead of iterating the states simultaneously and calculating quality off of the previous list of values, we use the most recent values $V_{s'}^{(i)}$.
3. One transition is responsible for significantly increasing the required number of iterations: the self-loop, which happens when fouls occur on a 2-strike count. Since this loop occurs with high probability, many more iterations of these states are necessary before their values will converge. To draw an obvious contrast, if the entire transition graph could be strictly ordered, only one iteration would be necessary at all. The reason we cannot simply use backward induction through the game states, running value iteration only on the individual 2-strike count states, is because of the larger paths that run through the entire batter cycle. So to deal with the self-loops, we can make a simple change, iterating the 2-strike count states additional times according to the current value iteration loop. Experimentally, we found that iterating these states two additional times works best on our setup. We experimented with more complex schedules to no avail.

Optimizing Batting Order

To the extent that our model captures relevant factors in a baseball game, we now have a way of estimating the ERA of a match-up between a pitcher and a batter lineup. In a real baseball game, choosing the order of a team's batter lineup is considered one of the key strategic decisions of the game. This is a binding decision which must be made before the game begins, although some substitutions can occur throughout the game. As such, the ability to compare different batting orders and potentially find an optimal batting order is very valuable.

Our model has the promising ability to directly compare batting orders, however the computing costs are very high. More troubling, there are an unapproachable number of permutations of batters that we'd need to explore in order to guarantee an optimal ordering, $9! = 362880$. One saving grace is that as a side-effect of our game state definition, our model computes values for all rotations of a given permutation, meaning a single run will compute 9 values. We can also get a noticeable performance boost by initializing V to the final weights of the previous run.

Search Strategies

Instead of attempting to brute-force our way through 40320 full simulations, we devise strategies for efficiently searching through the space of permutations.

1. The greedy hill climbing strategy tests its best permutation with swaps.
2. The stochastic hill climbing strategy also tests its current permutation with swaps, but will accept new permutations with a probability according to their percentile in the explored space.
3. The short term hill climbing strategy acts like regular greedy hill climbing, except it only considers a limited number of recent searches.
4. The genetic algorithm strategy uses the edge recombination operator (Whitley, Starkweather, and Fuquay 1989) to merge together the two best results it has found. We also perform a random swap on the child to create more variance.
5. The one-by-one strategy attempts to mimic a common-sense approach. First, see who performs best as first in the lineup. Once all nine batters have been checked, freeze the best option in first place and see who improves the lineup most as second. Repeat until the entire lineup is selected. This should take 37 steps uninterrupted, however it will reset when another solution is found as a rotation of a tested permutation, or when it finishes.

As an implementation detail, note that these strategies ought to account for better solutions found in the rotations of permutations they test. We give each strategy, along with a random baseline, a budget of 60 steps on 20 random matchups. For this experiment, to ease the computation time, we use a simplified version of the game with 2 strikes, 3 balls, 2 outs, 3 innings, and only 2 bases. Figure 8 shows the results.

Surprisingly, both our stochastic and short term hill climbing strategies perform worse than random. These were both made with the intent to mitigate the problems of local maxima that greedy algorithms run into, however our modifications appear to have harmed their performance. While it is confusing that any strategy performs worse than random, it is possible to inadvertently tune a strategy to search for bad permutations. In particular, both of these variants likely require hyperparameter tuning to be effective (the acceptance curve for the stochastic variant, and the memory size for the short term variant). The genetic algorithm and greedy hill climbing strategy perform about the same. The one-by-one strategy wins this test by a margin, so we will use this strategy for our next experiments.

Tests on the Full Game

To get a sense for the distribution of improvements our model predicts, we test one-by-one for 60 iterations against

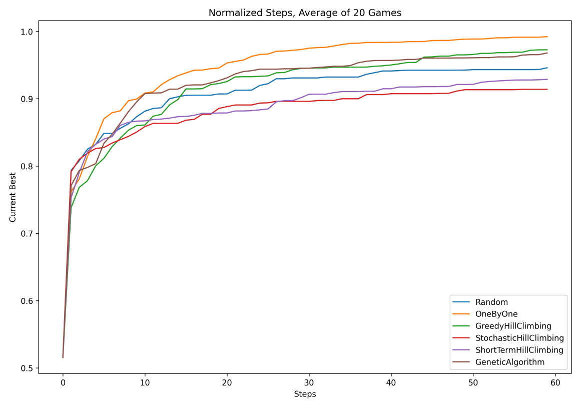

Figure 8: Over 20 games, one-by-one is the clear winner, and wins over half of them (individual results are not shown in the figure).

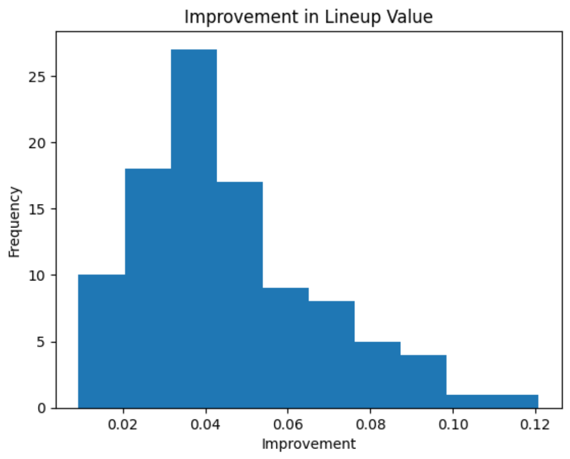

Figure 9: Histogram of batting order improvements (ERA) over 100 trials. The median is 0.041.

100 recent MLB lineups. We face each lineup against an average pitcher. Figures 9 and 10 show our results. The improvements are small but add up to 6.5 runs over the course of a full season, which falls in the range of standard sabermetric estimations.

Lineups According to Batting Average

A common discussion with regards to batting lineup is where to place batters according to their rank. Conventional wisdom says to place a team's fastest runners in the 1st and 2nd spots, and the best hitters in the 3rd or 4th spots. A more modern approach is putting the best batters at the start, in order to maximize number of at-bats. We construct a simple experiment to see how our model weighs in on this discussion. We calculate the average batter and artificially construct a team by scaling the batting average data. We run two instances of one-by-one on this match for 120 iterations, one starting from a lineup in order of decreasing skill, and one starting from a random lineup.

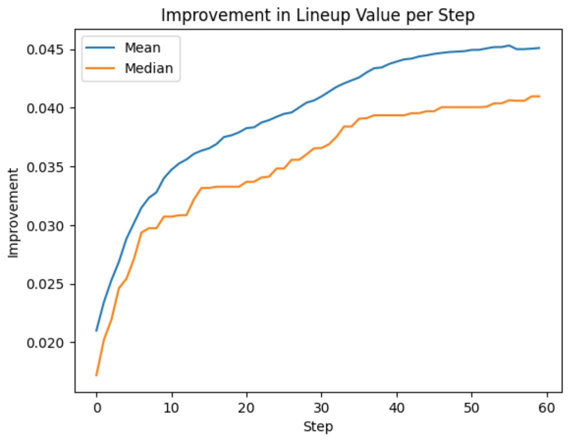

Figure 10: These curves are not linear, and after 60 steps the improvements are marginal.

Separately, both runs arrived at very similar lineups: $(1, 5, 7, 6, 2, 3, 4, 8, 9)$ and $(1, 5, 6, 7, 2, 4, 8, 9, 3)$, where 1 is the batter with the highest batting average and 9 is the one with the lowest. The evaluated ERAs are within ± 0.001, so they are about equivalent. These lineups affirm the common intuition that worse batters should be placed later and better batters should be placed first. Interestingly, the optimized lineup places batters 5, 6, and 7 before 2, 3, and 4. It is difficult to say how a strategic implications from our model translate to real-life baseball. We hypothesize that placing some better batters later in the lineup results in a more even selection inning to inning. Of course, there are many batter traits which might be relevant to the batting lineup which our model does not directly account for, however measure of overall skill level are likely the most important factor.

Conclusion

We extend an existing stochastic model of a baseball at-bat to encompass the entire game, and show how the model can be used to offer insight into complex baseball problems. While the computational requirements for such a model are high, due to the use of linear programming, we offer insight and several optimizations that improve its viability. We experiment on the problem of selecting an optimal batting order and find a promising strategy for exploring this space. Our model predicts modest improvements to existing lineups and an interesting recommendation for ordering batters relative to batting average.

References

Douglas, C.; Witt, E.; Bendy, M.; and Vorobeychik, Y. 2021. Computing an Optimal Pitching Strategy in a Baseball At-Bat. arXiv:2110.04321.

Littman, M. L. 1994. Markov games as a framework for multi-agent reinforcement learning. In Cohen, W. W.; and Hirsh, H., eds., *Machine Learning Proceedings 1994*, 157–163. San Francisco (CA): Morgan Kaufmann. ISBN 978-1-55860-335-6.

Whitley, L. D.; Starkweather, T.; and Fuquay, D. 1989. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, 133–140. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1558600663.