Optimal Batting Order through a Markov Chain Genetic Algorithm

Henry Fisher

May 9, 2025

Abstract

This report explores the problem of finding the optimal batting order for the Boston Red Sox. I create a Markov simulation for a baseball game based on real-world statistics, evolving a population of random lineups through a genetic algorithm to find the top-scoring lineup. My analysis shows that the Red Sox lineup can be significantly improved by front-loading on-base-percentage hitters, regardless of slugging percentage, and putting a particular emphasis on double hitting. This paper demonstrates that traditional batting heuristics can be outperformed by statistically optimized orders, and provides a robust batting model which can be extended for real-world use.

1 Introduction

Baseball is a multi-billion dollar business bringing in fans from around the globe, and teams can optimize their performance by running simulations based on game statistics, as famously featured in the movie *Moneyball*. In particular, one essential question in baseball is lineup order, which can involve a complex interplay of individual player abilities (i.e. on base percentage vs. home run hitters), situational probabilities (i.e. base running statistics), and the frequency with which each lineup spot comes to the plate, since the first batter gets dozens more at-bats than the ninth through the course of a season. One classic question is whether a team should place their worst hitter 7th or 9th in the lineup. After all, batters in the 7th position tend to come to the plate more often than those in the 9th, but putting a more competent player 9th can create a leadoff effect when the lineup cycles back to players 1-3, so the team's best hitters can allow the 9th player to score more frequently. Tradeoffs like these are always present in any choice of a lineup, so the question remains: has a team like the Boston Red Sox chosen their lineup optimally, or might they be able to do better?

I explore these questions and more through a Markov simulation which simulates a baseball game probabilistically, using batting statistics from the Red Sox 2025 season. Since the state space is very large, we cannot use a typical analytical Markov approach, but rather use an individual agent simulation. We treat maximizing simulation score as a "black box" optimization problem, which we approach with a genetic algorithm, maintaining a population of lineups which reproduce like DNA through generations based on natural selection.

2 Model Design and Assumptions

There are several ways to approach the problem of optimal batting order. One idea was to construct a traditional Markov chain with an explicit transition matrix, where each state could encode the inning, the number of outs, the number of runs, the current state of all 3 bases, and the current batter at the plate. Let's take a look at how many states this would involve:

2^3 combinations of base positions * 9 innings * 3 outs * potentially boundless number of runs (unless capped) * 9 batters

As you can see, the transition matrix would be enormous, and the approach quickly becomes intractable. Instead, I used an individual agent simulation. We assume batters come up to the plate in a given lineup order, batting either a walk, Single, Double, Triple, Home Run, or out, based on their real-world probability, and advancing the game state. An array keeps track of the base position of each runner, and the base runners advance with their own probabilities, gathered from the Red Sox 2025 base running statistics. The game is advanced through nine innings, and results are aggregated over hundreds of games to track average runs / game of a given lineup.

Figure 1 and table 1 give the batting and base-running probabilities in the simulation.

	name	1B	2B	3B	HR	BB	out
0	Jarren Duran	0.156977	0.052326	0.023256	0.011628	0.058140	0.697674
1	Rafael Devers	0.112426	0.065089	0.000000	0.029586	0.159763	0.633136
2	Alex Bregman	0.141104	0.085890	0.000000	0.049080	0.092025	0.631902
3	Triston Casas	0.107143	0.026786	0.000000	0.026786	0.098214	0.741071
4	Trevor Story	0.197279	0.013605	0.000000	0.034014	0.040816	0.714286
5	Wilyer Abreu	0.139706	0.044118	0.000000	0.051471	0.154412	0.610294
6	Kristian Campbell	0.152672	0.061069	0.000000	0.030534	0.145038	0.610687
7	Connor Wong	0.117647	0.000000	0.000000	0.000000	0.088235	0.794118
8	Ceddanne Rafaela	0.142857	0.025210	0.008403	0.016807	0.058824	0.747899

Figure 1: Batting probabilities, in order of the usual 2025 Red Sox lineup.

Batter	1B	2B	3B	HR
Runner on 1st	P(2B) = 0.67 P(3B) = 0.3 P(Home) = 0.03	P(3B) = 0.67 P(Home) = 0.33	Home	Home
Runner on 2nd	P(3B) = 0.4 P(Home) = 0.6	Home	Home	Home
Runner on 3rd	Home	Home	Home	Home

Table 1: Simplified base running probabilties from Baseball Reference. Columns represent the play, rows represent the base runner, cells represent transition probabilities, i.e. P(3B) = 0.3 in the "runner on 1st" row and "1B" column is the probability a runner on 1st reaches 3rd base if the batter hits a single. "Home" cells mean the player is guaranteed to score.

Given that we are doing "black box" optimization with this simulation, there are still several options for finding the best lineup. One idea was simply to brute force all possible lineups and see which one scores the best. Using time.time(), it took 0.05 seconds to simulate 1000 games with a given lineup on my macbook. If we did this for all 9! lineups of the top 9 batters, it would take 18,144 seconds, or just over 5 hours. It's not *completely* unreasonable, but any extensions of this project – like, say, considering 10 candidates for a 9-person lineup – would become unfeasible. For this reason, we instead have to use some sort of algorithm to sample the input space.

I considered 3 algorithms for this purpose: greedy algorithm, genetic algorithm, and MCMC. Greedy algorithm, which picks players in order based on who creates the highest increase in expected runs, will converge to a locally optimal solution, but will not necessarily capture the nuances of batting order – going back to the introductory question, greedy algorithm would never consider putting the worst batter in position 7, since it will always postpone the worst batter to the last possible moment. MCMC is more complex and slower to converge than the genetic algorithm, and might be good for sampling a larger distribution of good solutions, but genetic algorithm is best for finding one great solution fast, which is all we need.

In the genetic algorithm, we treat the lineups like DNA. We begin with an initial set of *population_size* random lineups of the nine players in the batting order. We calculate the fitness of each member of the population, which is just their average runs calculated through simulating *n_games* games. Then we stochastically select parent lineups to reproduce. Selection occurs via the function *select_parents*; the reproduction probability is influenced by the fitness score so fitter parents are more likely to reproduce. Reproduction randomly combines the lineups of both parents through the *crossover* function. We introduce a random swap in each child's lineup with probability *mutation_rate*, and these children form the next population. We then repeat this process for *generations* generations.

In addition to choosing the parameters of the genetic algorithm, we must also design the functions *select_parents* and *crossover*.

I tried two functions for *select_parents*: one simply performed random sampling, with a parent's probability of selection directly weighted by their fitness score. However, the populations didn't generally improve, but just randomly, since the weights for fitter parents (around 4.5 runs/game) simply weren't strong enough compared to the weights for weaker parents (around 4 runs/game). The other function was tournament sampling, where two random samples of 10 population members were in-

dependently chosen, with the parents being the fittest members of those two samples. This function applied much stronger pressure for survival of the fittest, and resulted in a general upward trend through generations. I also used elitism in the model: the top two parents are always directly carried over as children in the next generation.

I display my experiments with parent selection mechanisms and elitism in the sensitivity analysis section.

When designing the *crossover* function, I was careful to design a function which could preserve synergies between players if one parent has a really effective lineup subsequence. We take a random slice of the first parent's lineup (i.e. take the sublist starting from a random start index and going to a random end index) and fill the rest of the lineup with the remaining players not included in the slice based on the order of the second parent's lineup. Then the first and second parents are swapped to create the second child, so each pair of parents has two children.

Let's go through a simple example with two 6-person lineups reproducing:

Parent 1: [A B C D E F]

Parent 2: [F A B C D E]

The slice of the first parent is randomly chosen to be from index $1 \rightarrow 3$. The child's lineup is filled in correspondingly:

[_ B C D _ _]

Since A, E, F are missing, they are filled in based on the order of the second parent, which has F before A before E.

[F B C D A E]

This will be the first child of the two parents; the second child will be based on the same process but with the second parent's lineup being randomly sliced and the first parent's lineup being filled in.

Finally we add a small mutation probability that two additional basemen are swapped to help prevent converging too early. Let's say in this case the mutation does occur, and indices 0 and 3 are selected:

[D B C F A E]

This is the final child for the next generation.

3 Model Testing

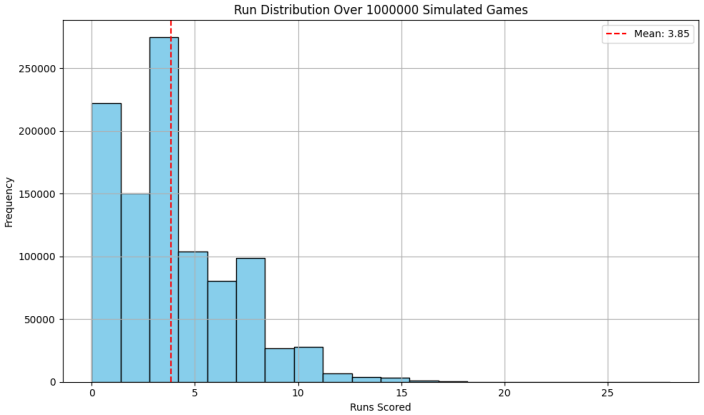

Figure 2: Run histogram over 1 million simulated games

I tested the baseball simulator by running it with the true 2025 Sox lineup; the sample mean of 1M simulated games was 3.85 and the sample STD was 2.86. (Figure 2) The average runs per game in MLB history is usually around 4.5 - 5, and the Red Sox have averaged 4.86 runs per game this season, so common sense dictates this is underestimating somehow.

We can also run a t-test to confirm this. Assuming a null hypothesis that the true mean is 3.85, with the sample STD being 3 (this is approximated based on a quick Google search of historical data rather than the simulation),

$$\bar{x} = 4.86 \quad \mu_0 = 3.85 \quad s = 3 \quad n = 39$$

$$t = \frac{4.86 - 3.85}{\frac{3}{\sqrt{39}}} \approx 2.10$$

With this t-stastistic and $df = 38$, we have

$$p\text{-value} \approx 0.0355$$

Since the p-value is less than 0.05, we reject the null hypothesis. I talk about reasons why the model might be underestimating in the "discussion" section. Suffice it to say that the model produces a close enough approximation that we can still get a general sense of which lineups are better than others.

N_games, the number of games to simulate each time when gauging fitness, was chosen based on a simple statistical analysis using the CLT: for a desired 95% confidence interval width of 0.4, which is about 5% of the mean in each direction, we must choose $n \approx 785$. (Figure 3)

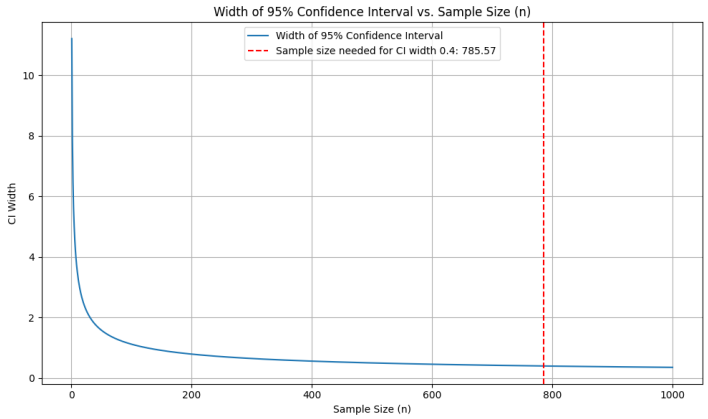

Figure 3: Using the CLT to determine what sample size we need

I first tested the genetic algorithm with a sanity check, a "silly lineup" (Figures 4, 5) which has a pretty obvious best order (I used parameters population_size=20, generations=100, mutation rate=0.1, n games=785, but many different parameter sets worked for this simple problem). Batting heuristics dictates that the harder-hitting players should go later, so we can load up the bases first with OBP batters and then get them all to run. This means the order should be something like: William, Stephen, Daniel, Terrence, Harry, Larry. The simulation agrees with this conclusion, except it consistently suggests that Daniel should go before Stephen, which actually makes sense since based on the Baseball Reference base running statistics we used, the chances for a player on 2nd to score on a single are quite much higher than the chances for a player on 1st to score on a double.

	name	1B	2B	3B	HR	BB	out
0	Stephen Singlehitter	1	0.0	0.000000	0.00	0	0.000000
1	Daniel Doublehitter	0	0.5	0.000000	0.00	0	0.500000
2	Terrence Tripplehitter	0	0.0	0.333333	0.00	0	0.666667
3	Harry Homerunner	0	0.0	0.000000	0.25	0	0.750000
4	William Walker	0	0.0	0.000000	0.00	1	1.000000
5	Larry Loser	0	0.0	0.000000	0.00	0	1.000000

Figure 4: Silly lineup

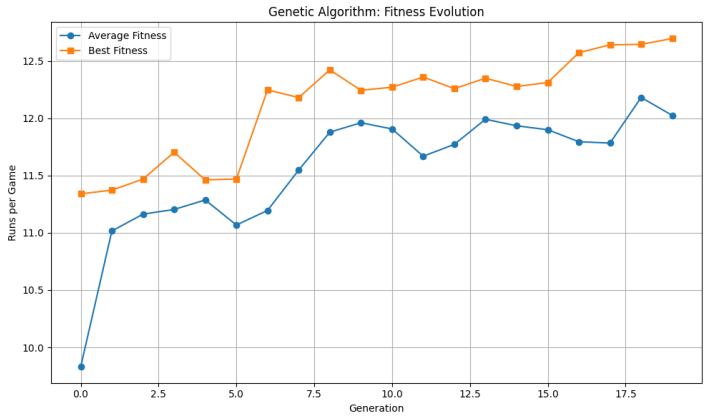

Figure 5: The algorithm quickly finds an effective order for the silly lineup

I then ran the genetic algorithm on the 2025 Red Sox lineup, with parameters population size=30, generations=100, mutation rate=0.1, n games=785 (see the following section for how these parameters were chosen). (Figures 6, 7) The results were promising, with average runs per game = 4.37, an 0.52-run increase from the original lineup. I ran the simulation a few times and it resulted in extremely similar lineups each time, sometimes with Abreu and Campbell swapped (they have very similar profiles), suggesting that we have found a globally optimal solution or gotten close. The evolution graph shows a steady upward trend and then a plateau, indicating that the genetic algorithm is working and has found a high-quality solution.

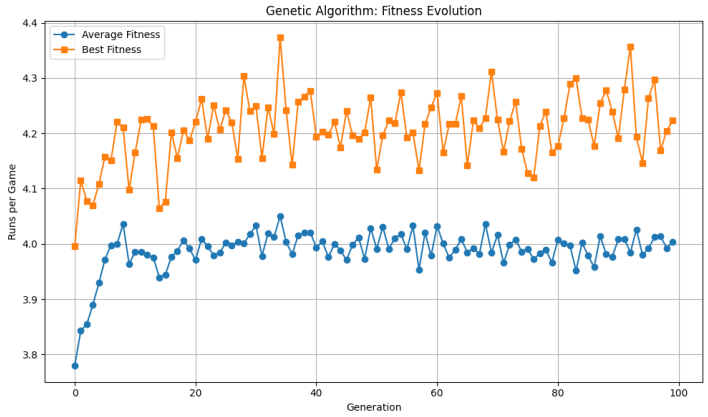

Figure 6: Sox lineup evolution with population_size=30, generations=100, mutation_rate=0.1, n_games=785, with the best lineup averaging 4.37 runs per game

	name	1B	2B	3B	HR	BB	out
0	Rafael Devers	0.112426	0.065089	0.000000	0.029586	0.159763	0.633136
1	Kristian Campbell	0.152672	0.061069	0.000000	0.030534	0.145038	0.610687
2	Wilyer Abreu	0.139706	0.044118	0.000000	0.051471	0.154412	0.610294
3	Alex Bregman	0.141104	0.085890	0.000000	0.049080	0.092025	0.631902
4	Jarren Duran	0.156977	0.052326	0.023256	0.011628	0.058140	0.697674
5	Trevor Story	0.197279	0.013605	0.000000	0.034014	0.040816	0.714286
6	Ceddanne Rafaela	0.142857	0.025210	0.008403	0.016807	0.058824	0.747899
7	Triston Casas	0.107143	0.026786	0.000000	0.026786	0.098214	0.741071
8	Connor Wong	0.117647	0.000000	0.000000	0.000000	0.088235	0.794118

Figure 7: Final Sox lineup maximizing runs per game

This optimal lineup is pretty different from the original Sox lineup, with Duran and Casas much later in the lineup, and Abreau and Campbell much higher. Interestingly, the algorithm puts the

batters pretty much in increasing order of out percentage, with the exception of Devers because of his exceptionally high BB and 2B percentage, which the silly model indicated were the most important statistics for early hitters to have. The model simply doesn't care for the idea that Abreau should go later in the lineup because is a "heavy hitter" with high HR – it wants him to go early because he just gets on base a lot. As for the question of whether the worst hitter should go 7th or 9th, we have a decisive answer: Connor Wong, significantly worse than the other hitters, is placed 9th in the optimal lineup.

4 Sensitivity analysis

The following sensitivity analysis assumes all parameters and functions are the same as the final model above – population_size=30, generations=100, mutation rate=0.1, n games=785, tournament selection, elitism – except the parameter or function being analyzed. I did not do sensitivity analysis on the number of generations because basically all models converged before 100 generations or showed no signs that they were ever going to converge.

The model produces nearly identical lineups to the final lineup when mutation_rate = 0 and mutation_rate = 0.3 (0 and 1 swaps away respectively), indicating that perhaps I didn't need to implement mutation at all. Mutation_rate = 0.3 converges slower, though, and at mutation_rate = 1, the model becomes too noisy and doesn't show general improvement because there's too much randomness; the resulting lineup is totally different.

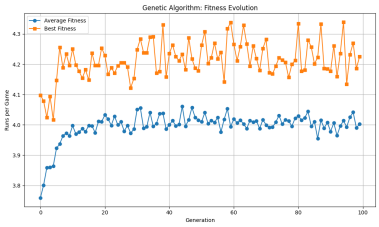

Figure 8: $mutation\_rate = 0$

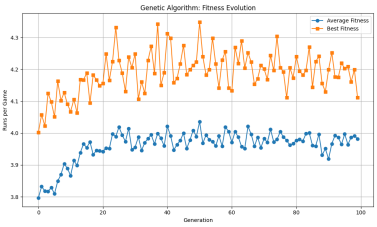

Figure 9: $mutation\_rate = 0.3$

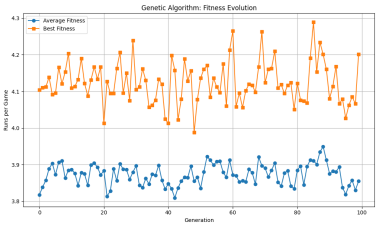

Figure 10: $mutation\_rate = 1$

We calculated earlier that $n\_games = 785$ produces a confidence interval I consider to be acceptable; we can see what happens when the model makes a fitness evaluation after running fewer games. The lineups produced by choosing $n\_games = 300$ and $n\_games = 50$ are quite different, and much less reliable, than the final lineup. As n_games decreases, the "Best Fitness" statistic jumps around a lot more and is way less accurate to a lineup's "true fitness". For instance, the spike in the middle of Figure 11 is used to determine the optimal lineup, but it may have just been a lucky series of 300 games – an event which happens much more frequently in a population of 30 over 100 generations than a lucky series of 785 games.

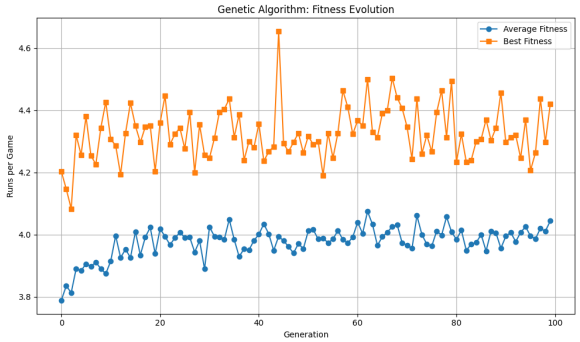

Figure 11: $n\_games = 300$

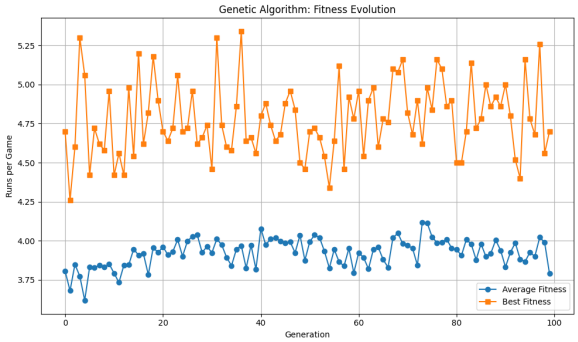

Figure 12: $n\_games = 50$

The model produces nearly identical lineups to the final lineup when population_size = 10 and 20 (both are 1 swap away), although reducing the population size increases the number of generations

required towards convergence, since population diversity is significantly reduced at each generation. This is why mean and average fitness get so much closer together when population is reduced.

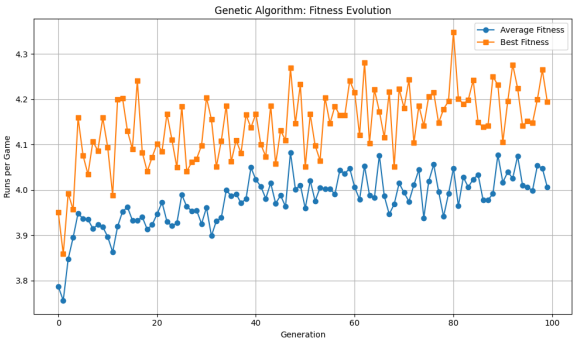

Figure 13: $population\_size = 10$

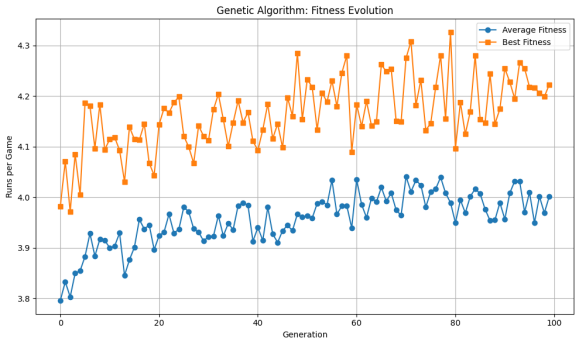

Figure 14: $population\_size = 20$

I originally tried running the model with the fitness scores being directly used as sample weights for the parents of the next generation: the results were very poor, with a totally noisy and random graph. That's why I had to change it to tournament sampling (randomly select 10, pick the fittest as the parent), which is used for the final model. I also tried variations on this model with and without elitism (directly carrying over the top 2 players into the next generation). Removing elitism resulted in a nearly identical lineup to the final (1 swap away) but slower convergence and a bit noisier history graph.

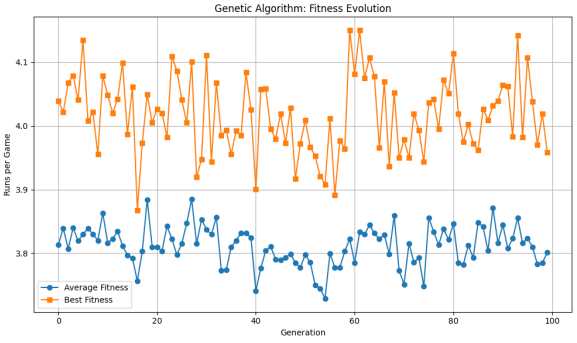

Figure 15: Weighted random sampling by fitness for parent selection

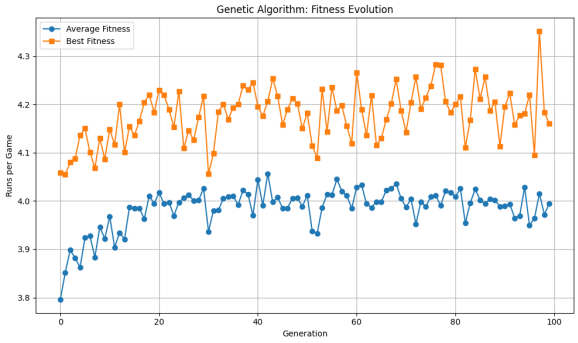

Figure 16: Without elitism

Since the batting probabilities for each player (1B, 2B, etc.) are also random variables which approximate "true" probabilities calculated from the average of only 39 games, I added random Gaussian noise to all of them to see how sensitive the model was to those probabilities (and normalized to make sure they still add to 1). With $\sigma = 0.01$, the best batting lineup found was similar to the final, but with a couple adjacent batters swapped: Devers, Abreu, Bregman, Campbell, Duran, Rafaela, Story, Casas, Wong. With $\sigma = 0.03$, the lineup was still somewhat similar, with the first batters being shuffled around a bit, but the latter batters mostly the same: Campbell, Abreu, Casas, Rafaela, Devers, Duran, Story, Bregman, Wong. With $\sigma = 0.05$, the lineup was completely different. This indicates that even if the "true" batting percentages were one or two percentage points away from those used in this project, the lineup found would still be nearly optimal, especially the latter half of the lineup.

	name	1B	2B	3B	HR	BB	out
0	Jarren Duran	0.133601	0.048465	0.012712	0.000615	0.060858	0.743750
1	Rafael Devers	0.112073	0.061224	0.000000	0.037551	0.166849	0.622303
2	Alex Bregman	0.143778	0.087836	0.006590	0.065515	0.096732	0.599549
3	Wilyer Abreu	0.120626	0.043747	0.000000	0.050077	0.164241	0.621308
4	Trevor Story	0.190582	0.009172	0.006114	0.040675	0.039644	0.713813
5	Triston Casas	0.087294	0.044644	0.002262	0.033937	0.079435	0.752428
6	Kristian Campbell	0.158268	0.064090	0.000000	0.025755	0.113858	0.638029
7	Connor Wong	0.119800	0.000000	0.000000	0.007675	0.086773	0.785751
8	Ceddanne Rafaela	0.154523	0.005916	0.002541	0.022193	0.058175	0.756653

Figure 17: Sox lineup with with $\sigma = 0.01$ gaussian noise added to each cell, then normalized.

5 Discussion

One key flaw in the model is that when we plug in the actual Sox lineup, it consistently underestimates their average performance across n games, expecting them to score 3.85 runs per game when really they score around 4.86. There are a few reasons why this occurs.

- There are offensive plays we did not include in this simulation, like stealing, hit by pitch. A more complex simulation could could scrape a more comprehensive set of stats to include these plays.
- Batting stats can vary heavily from game to game depending on the opposing team. In this simulation, the Red Sox are basically up against an average team all the time, but in reality the Red Sox sometimes plays really bad teams, scoring tons of runs in those games which skews the distribution. An extension to this project could randomly generate a "defensive score" for the opposing team, perhaps scrape a distribution of game-by-game batting statistics for each player and sample different quantiles of this distribution based on the opposing team's "defensive score."
- We don't take into account batter choice: each player can *choose* to try to hit different plays depending on the field state. Some might be able to hit a play but simply haven't chosen to go for it this season (perhaps because of their usual lineup position), resulting in a "0" percentage for that play in the table. Indeed, looking in the table, there are a lot of 0s, where realistically there should be nonzero percentages for each player to hit each play. If I was the manager of the Sox, I could rectify this by having the real players try out the new lineups, forcing them to try out different plays and getting a sense of how the batting percentages can change depending on their position in the lineup.

Although we've clearly erred on the side of underestimating offensive power, there are also *defensive* plays we did not account for in the model, like double outs, which a better simulation would include.

Other improvements we could make to this project include considering larger sets of players (i.e. we could consider the top 15 batters in the Sox for candidacy in the 9-personal lineup), aggregating career data to see if that gives us more accurate batting percentages, and incorporating individual base-running statistics rather than just using the team's overall base-running statistics (i.e. a faster player is not just more likely than others to get to 1st, but more likely to get to 3rd after he is on 1st).

Of course, there are always going to be flaws in modeling sports performance with probabilistic simulations – it's extremely difficult to capture the psychological aspects of gameplay, like how when a team reaches a certain momentum, a player hits better, and how certain players to work together better than other players based on personalities and playstyles.

6 Conclusion

Although this model was quite simplified, it was able to teach us about some important lessons about batting order: BB (perhaps obviously) and 2B (more surprisingly) are essential for the first hitters, but otherwise, OBP matters more than anything else, including how "heavy" a hitter is (their probability of hitting plays like 3B or HR). We learned that although the idea that a better 9th hitter can create a leadoff effect for the 1st hitter is tempting, it's actually probably better to have the worst player as 9th simply because he plays less frequently. Most importantly, we learned that pairing a Markov simulation with a genetic algorithms creates a promising, robust model which can help teams optimize their performance – if the Sox aren't using this model, they certainly should.

7 Acknowledgements

I used ChatGPT to help debug and fine-tune my baseball simulation and genetic algorithm code. I also used the AI tool on Overleaf to help me do the latex formatting.