Rock Paper Scissors Bot (CS109 Challenge Project)

James Chen

December 6, 2023

Introduction

For my project, I used Beta functions to beat humans at rock paper scissors. Humans naturally have patterns in the way they play games or try to be random, and using probability you can create a bot that exploits these patterns! The concept is that the bot looks back at the most recent moves the user has played and examines the history of what the user has done in this specific scenario. Then, based on that it makes the decision for its move. For instance, let's say that in the last round the user played rock. Then, we look at what happened all of the previous rounds we have played where the user had played rock in the previous round. We maintain this information in a set of three beta functions, one for our past performance when going rock, one for paper, and one for scissors. This might look something like this:

When user had played rock in the previous round:
Rock = Beta(3, 4)
Paper = Beta(6, 3)
Scissors = Beta(4, 6)

Our Betas were initialized with Laplace priors, so using this information we can calculate our past performance in the round directly after the user played rock. When the bot played rock it won 1 time and lost 2 times, when it played paper it won 4 times and lost 1 time, and when it played scissors it won 2 times and lost 4 times. Seeing these results, it seems like we should probably play paper since it had significantly better performance than the other two. However, if we just always went with the move we had the best past performance with it would make our bot really predictable and easy to beat. Therefore, we add in an element of randomness by randomly sampling from the three Beta distributions and choosing the move that got the highest sample (this is very similar to a three-armed bandit problem). Then, once we make our move and see the user's we would update these three Beta functions and repeat.

Variations

I tried a couple of different ways of updating the beta functions, each of which led to slightly different results. My first question was how far back into the past to look. If we look 1 move back, we have more limited information that we can draw upon, but we only need 9 Beta functions, 3 for each of the 3 moves our opponent could have made last turn, so our model can learn very quickly. If we instead look 2 moves back, we can use more information but we need 27 Betas, 3 for each of the 9 combinations of moves our opponent could have made in the past two turns. We can see that the amount of data we need to make effective predictions scales exponentially with the distance we look back. Even with looking two steps back, having 27 Betas we need to fill with information is a very difficult task if the user isn't willing to play rock paper scissors for a very long time. Therefore, for all my tests I only looked one move back in order to optimize the speed at which the model would learn and be able to improve it's play.

Another question is whether or not to consider your own moves (or equivalently whether the opponent won or lost) in your analysis. This is certainly very useful information, but we encounter the same problem that the amount of data we need scales exponentially as we add more parameters. Also, in some initial tests it seemed as if looking at the opponent's past two moves produced better models than looking at both the opponent's and bot's last move. Since both of these methods have the same number of Beta functions, it seems as if this might not be the most effective method.

A final question was how to update the Beta functions in the model. In my initial iterations of the program, when there was a situation where the bot had accessed the three Beta functions for the opponents last move and made it's decision, I only updated the Beta function associated with the bot's choice. If the bot won, the α parameter would be increased by 1. If the bot lost, the β parameter would be increased by 1. If the bot tied, I tried two strategies of either not changing the parameters of the Beta or increasing both the α and β parameters by 1. However, I realized that this was not the most efficient way to get information and learn about the user's patterns. My second approach was to look at the user's move instead of the bot's. For instance, if the user played rock, no matter what the bot played it would add 1 to the α parameter for the paper Beta function since paper would have beaten the user's rock, and it would add 1 to the β parameter for the scissors Beta function since if the bot had played scissors, the user would have won. This ended up getting much better results since the bot was able to get a lot more information from each round, and was therefore able to pick up on patterns faster.

Performance

For set sequences of moves, even relatively complicated ones, this model performs extremely well. For instance, if we were to repeat the moves "r, p, s, p, r, p, s, s, r" 10 times in a row for a total of 90 moves, we get 38 wins, 19 losses, and 32 ties (fig1 below). Here, the bot seems to be performing very well and picking up on our pattern. If we instead play 900 rounds, this becomes even more clear. We get 391 wins, 104 losses, and 404 ties (fig2 below). Our bot is winning almost 4 times as much as it is losing, very convincingly beating our set pattern.

However, even though humans are never perfectly random they normally don't follow a set pattern of moves. In my next text, I tried to be as random as possible while quickly inputting moves. Over the course of 200 rounds, the bot ended up beating me with 86 wins, 44 losses, and 70 ties (fig3 below). When I was instantly typing my move without giving it any thought except generally trying to random, I fell into very recognizable patterns and was beaten very convincingly by the bot.

For my final test, I played how I would against another human if I really wanted to win, thinking back and analyzing the past couple of rounds and making my decision based on that (like how the bot does!). This was naturally a much slower progress, so I only went for 100 rounds. In the end, the computer still beat me with 35 wins, 25 losses, and 40 ties (fig4 below). This was the longest set of games I played where I was trying, and in shorter runs I had beat the bot a few times but I was very impressed by the performance in this test. While we started off pretty equal, the bot pulled ahead after about 50 rounds and was consistently beating me!

Future Work

I had a few extra ideas that could be cool to experiment with to improve this model:

1. One idea I had would be to extend the distance you look into the past based on how much data you have. To do this, you would maintain different record dictionaries, each of which looks back a different number of moves. The most practical version of this would probably have three dictionaries, one for $n = 1$, one for $n = 2$, and one for $n = 3$. The primary reason looking back further didn't work as well was because it took too much time for the model to amass enough data, but if you were able to adapt how far you look back based on the amount of data it could improve model performance when you are playing a very large number of rounds in a row.

2. Another way to address the problem of not being able to look further back because of the lack of data would be to establish a prior based on what humans normally play. However, I found this to be quite difficult to do. I couldn't find databases online of rock paper scissors games, and also if I were to do this strategy it would make the bot much more fallible to people with knowledge of how the bot works or common rock paper scissors patterns. If I were to find a way to establish an effective prior it would be very interesting to see how it impacts the model's performance against different strategies.

The program!

Here is a link to the code repository for this project: https://github.com/apple-314/rock-paper-scissors.

You have the option to save the model weights after every set of games so next time you can load it in and use it as the starting point for future games. When you save, you can also find graphs in the data folder that show you how the bot performed over time. Check it out, and see if you can beat the bot!

Appendix

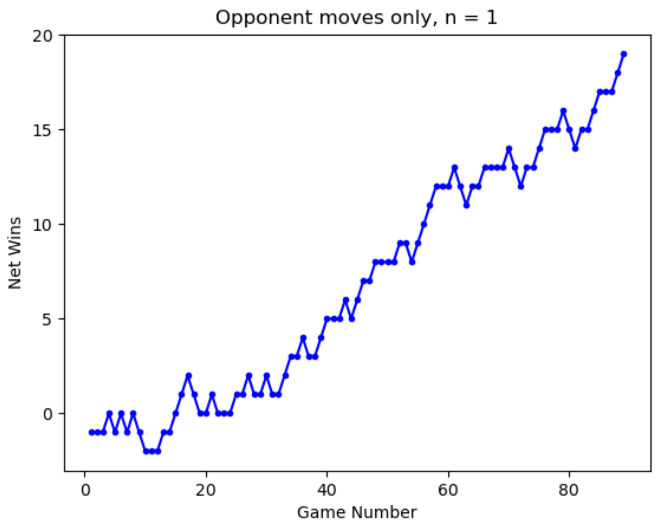

Figure 1: 90 rounds of set pattern

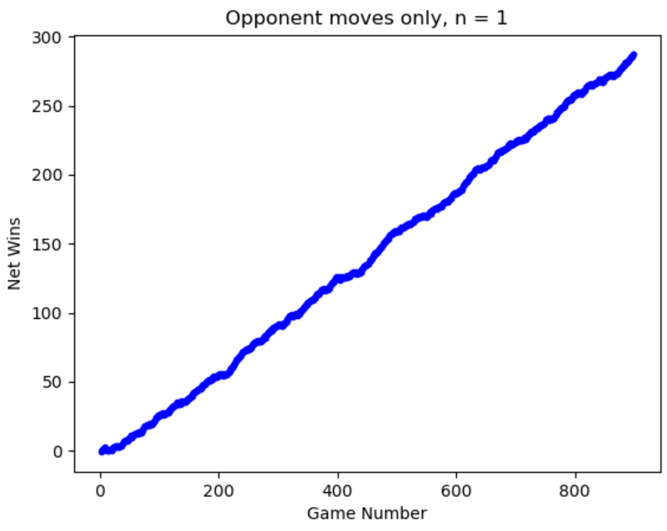

Figure 2: 900 rounds of set pattern

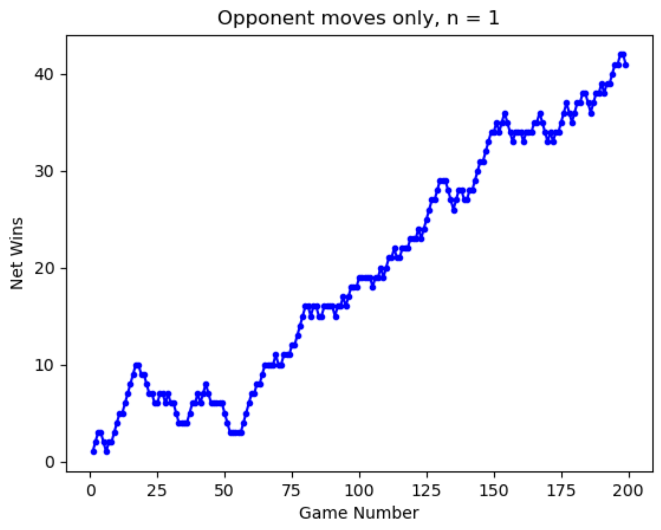

Figure 3: 200 rounds pseudorandom choices

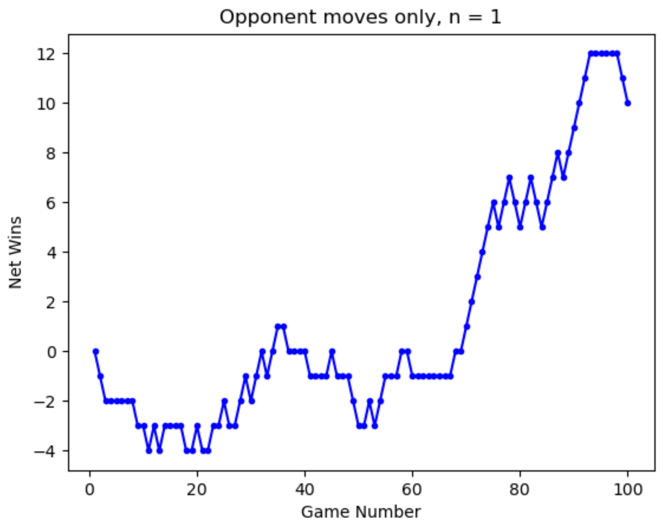

Figure 4: 100 rounds of careful play