Rock-Paper-Scissors AI Agent

Alp Gokcek, Erdal Sidal Dogan, Mert Komurcuoglu
{gokcekal, doganer, komurcuoglum}@mef.edu.tr
MEF University
December 6, 2020

I. Introduction

Rock-Paper-Scissors is a game that has been around for while and well known to almost everyone. The game is played with two players, goal is to defeat the opponent by making a choice that prevails the opponents. Rules to defeat is as follows;

1) Paper>Rock
2) Rock>Scissors
3) Scissors>Paper

In the beginning of the game, two players make a random choice among the Rock-Paper-Scissor triplet simultaneously. Whoever made the choice that defeats the opponents choice wins the game.

Even though it seems that the choices are random and probabilities of winning for each player is equal, scientist discover that "if a player wins for a round, they are much more likely to win to following one too".Furthermore, the choices that players make are prone to biases. [?]

II. Solution Design Process

For this problem, our initial approach was to design a rule based system where the each case was covered with the if-else statements. Later on, we searched for another applicable method that we have covered in the lecture so far. Upon our researches, we decided to use *Discrete Markov Chains*

DMC methods suggest creating a *State Transition Matrix*. Each column of the matrix is the probability score of the upcoming move of the opponent; rows of the matrix corresponds to a previous set of plays and stochastic.

UML Diagram of the `RPS_Agent` class can be seen from Figure 1.

TABLE I: Transition Matrix with initial transition values

State	R	P	S
RRR	0.333	0.333	0.333
RRP	0.333	0.333	0.333
RRS	0.333	0.333	0.333
⋮			
SSS	0.333	0.333	0.333

III. Implementation

The program has been implemented with *Python Language*. In order to have concise structure in the program, we defined our AI Agent as an object. This class, consist of transition matrix and all the complementary methods such as create matrix, update matrix etc.

Most important function and variables in this class are as follows; `update_transition_matrix()`, `predict()`

A. Update Transition Matrix

The program makes probability calculations relying on last 3 moves of the opponent, hence, before this calculations game must be played at least three times. This method checks the game has played for at least 3 rounds, if not, doesn't update the matrix at all. Otherwise, it multiplies each value on the row with a predefined `decay` value in order to reduce the weight of the earlier moves and increase the effect of the last move on the probability score.

Finally, it updates the `transition_sum_matrix` where the number of occurrence for the each case is stored. As the `transition_sum_matrix` is updated, new probability scores are calculated on the `transiton_matrix`

Listing 1: `update_transition_matrix()` method

```
def update_transition_matrix(self,
    opponent_move):
     global POSSIBLE_MOVES
     if len(self.moves) <= len(
         LAST_POSSIBLE_MOVES[0]):
          return None
     for i in range(len(self.
         transition_sum_matrix[self.
         last_moves])):
          self.transition_sum_matrix[self.
              last_moves][i] *= self.decay
     self.transition_sum_matrix[self.
         last_moves][POSSIBLE_MOVES.index(
         opponent_move)] += 1

     transition_matrix_row = deepcopy(self
         .transition_sum_matrix[self.
         last_moves])

     row_sum = sum(transition_matrix_row)
     transition_matrix_row[:] = [count/
         row_sum for count in
         transition_matrix_row]
     self.transition_matrix[self.
         last_moves] =
         transition_matrix_row
```

B. Predict

This method predicts the opponents move.

Since we don't have any data to make our predictions to be based on, in the initial predictions we used the statistics that psychologists observed when conduction a research about the human behavior on Rock-Paper-Scissors.

For the later moves, we use the transition matrix as lookup table where the row corresponding the last 3 moves presents us with the probabilities of the next move. Intuitively, we choose the next state with the highest probability score. However, if one of the scores are significantly higher than the others, we assume that it will be predicted by our opponent agent and it will make their move accordingly. In order to protect ourselves from this move, we make the move with least probability score to surprise the opponent.

Listing 2: `predict()` method

```
def predict(self):
    global POSSIBLE_MOVES, beats
    if len(self.predictions) == 0:
        prediction = random.choices(
            population=POSSIBLE_MOVES,
            weights=[0.45, 0.35, 0.20], k
            =1)
        return prediction[0]
    elif len(self.predictions) in range
        (1, len(LAST_POSSIBLE_MOVES[0])):
        last_prediction, last_move =
            beats[self.predictions[-1]],
            self.moves[-1]
        last_result = self.get_result(
            last_prediction, last_move)

        if last_result in [0,1]:
            prediction = beats[
                last_prediction]
        else:
            prediction = beats[last_move]
        return prediction
    else:
        row = self.transition_matrix[self
            .last_moves]
        if max(row) == min(row):
            return random.choices(
                POSSIBLE_MOVES, [0.35,
                0.30, 0.35], k=1)[0]
        else:
            if min(row) * random.uniform
                (1.7, 3) <= max(row):
                return random.choices(
                    POSSIBLE_MOVES,
                    weights=[1-prob for
                    prob in row], k=1)[0]
            else:
                return random.choices(
                    POSSIBLE_MOVES, row, k
                    =1)[0]
```

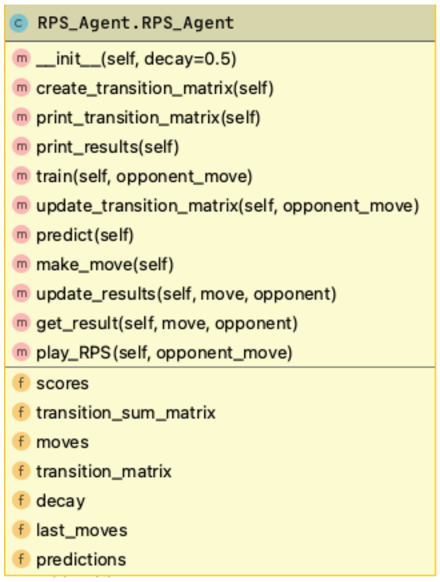
RPS_Agent.RPS_Agent
__init__(self, decay=0.5)
create_transition_matrix(self)
print_transition_matrix(self)
print_results(self)
train(self, opponent_move)
update_transition_matrix(self, opponent_move)
predict(self)
make_move(self)
update_results(self, move, opponent)
get_result(self, move, opponent)
play_RPS(self, opponent_move)
scores
transition_sum_matrix
moves
transition_matrix
decay
last_moves
predictions

Fig. 1: "UML Diagram of the RPS Agent Class"

IV. Testing

During the testing stage, we saw that our initial approach where the outcome with the highest probability is chosen and if there is a significant gap with other options, our predictions would be obvious to the opponent as well. Thus, we've implemented a system that will take precautions by reverting the probability scores by subtraction from 1 and converting the least likely one to be the most likely. By this approach, we increased our chances against the intelligent systems.

In this stage, we've also tested for different values of the `decay` value. Our observations indicated that the highest success value is reached when the decay has taken values between the $0.4 - 0.6$ range. We chose the set this value to 0.5

V. Teamwork

Most of the implementation is carried out by Alp & Mert, Erdal has contributed on the testing stage and forming a report.

References