Garrett Lawson

MATH 381 - Monte Carlo Simulation

Introduction

Monte Carlo methods are algorithms that involve the use of random sampling to obtain numerical results. As discussed in class, Monte Carlo methods can be used for optimization, simulation of random processes, and numerical integration, as well as a variety of other things. The purpose of this notebook is to demonstrate the use of Monte Carlo methods and the Central Limit Theorem to simulate a random game and estimate the probability of winning such a game with different strategies.

The Game

In this notebook we will analyze the following dice game:

Two players alternately roll a die. They can either

1. Add the roll to their score
2. Subtract the roll from their opponent's score.

The first player to reach the preset limit wins.

If the limit is not reached within 50 turns, the player with the highest score wins.

Fundamentally, this game deals with the differences in scores between players. Adding to one player’s score and subtracting from the other player’s score does not change the difference between scores. However adding may be preferred because each player wants to reach the win threshold. We can think of many different strategies for this game. In this notebook, we will analyze the following strategies:

Strategy A: Always adding to your score

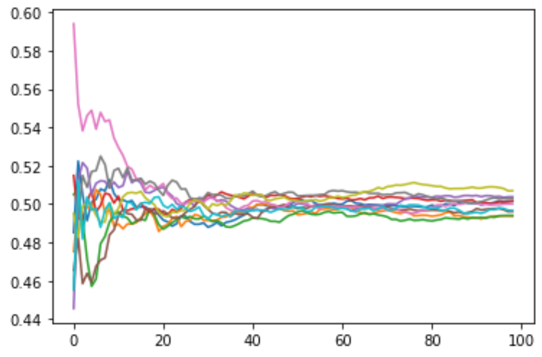
Strategy B: Only subtracting from the opponent’s score if they are close to winning, unless adding to your score would result in a win.

Strategy C: Only subtracting if the opponent is ahead by a certain amount, unless adding to your score would result in a win.

The Central Limit Theorem and Convergence

Using Monte Carlo methods, we will need to run many, many simulations with a given strategy to obtain an estimate for the win rate with that strategy. However, it can be difficult to gauge how accurate that estimate really is. In this case, we will use multiple runs of many simulations to obtain a more accurate estimate.

Here is what happens with 10 runs of 10,000 games using strategy A against an opponent who is also using strategy A (python3 A):

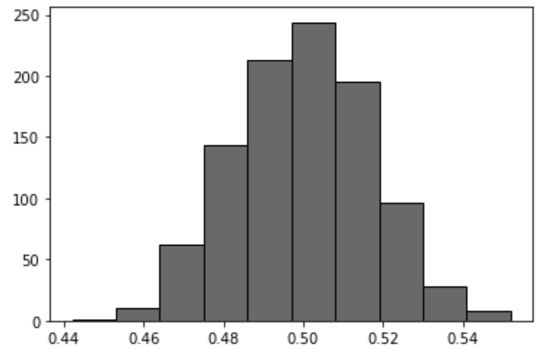


Win rate per 100 simulations

Because the win rates for each run tend to converge to a single value, we can be sure that we are using enough simulations to obtain an accurate result. But to which value do the runs converge to? Unfortunately, we cannot obtain the single, true value through the use of numerical methods. However, we can find the range of values that might contain the true value with a certain level of confidence. This is where the Central Limit Theorem comes into play.

The Central Limit Theorem states that if we take a large number of random samples from an unknown distribution and do this multiple times, the means of each set of random samples will be approximately normally distributed around the mean of the unknown distribution. In other words, the win rates obtained from each run are approximately normally distributed around the true win rate.

Here is the distribution of win rates obtained after 1000 runs:



As is seen in the graph above, the distribution of win rates is approximately normal. Using the fact that the distribution of win rates is very nearly normal, we can create confidence intervals for the true win rate of each strategy:

Suppose we have 10 estimates for the win rate of a strategy from running many simulations 10 different times. As was just shown, these estimates are approximately normally distributed by the Central Limit Theorem.

Using the fact that normally distributed implies symmetric about the mean, the probability that all 10 estimates are on one side of the true win rate is the following:

$$\frac{2}{2^{10}} = \frac{1}{2^9} = \frac{1}{512} \approx 0.001953125$$

This is the probability that all estimates are greater or less than the true win rate. In other words, we can say with $1 - 0.002 = 99.8\%$ confidence that the true win rate is between the highest and lowest estimate.

For example, if we run 10 runs of 100,000 games with strategy A vs strategy A, we obtain the following results (python3 A):

```
0.49983 0.49877 0.49917 0.49919 0.49917
0.50117 0.50074 0.49951  0.5012 0.50338
```

Then we can say with 99.8% confidence that the true win rate for strategy A vs A is in the interval [0.49877, 0.50338].

This is the method by which all confidence intervals will be calculated in this notebook.

Calculating Win Rates

Strategy B

As we just calculated, when both players blindly choose to add to their score no matter what score the opponent has, the win rate is approximately 50%. But what if the opponent is one point away from winning? Or three points away from winning? Unless you could win immediately with your current roll, the best strategy is not to add to your score. Instead, let us take on strategy B, where we subtract the roll from the opponent's score if they are α points close to winning, unless adding to your score would result in a win.

Let us be player one. These are the win rates for strategy B against an opponent that is also using strategy B for varying levels of α (computed with Python3 B):

Player 1						Player 2					
α	0	1	2	3	4	5	6	8	10	15	2
0	(0.493, 0.508)	(0.479, 0.495)	(0.463, 0.479)	(0.447, 0.459)	(0.419, 0.440)	(0.401, 0.418)	(0.394, 0.398)	(0.399, 0.408)	(0.420, 0.435)	(0.466, 0.481)	(0.490
1	(0.509, 0.520)	(0.496, 0.510)	(0.480, 0.490)	(0.462, 0.475)	(0.438, 0.458)	(0.413, 0.426)	(0.391, 0.412)	(0.408, 0.418)	(0.423, 0.436)	(0.471, 0.482)	(0.486
2	(0.525, 0.531)	(0.505, 0.518)	(0.497, 0.510)	(0.473, 0.479)	(0.448, 0.461)	(0.419, 0.436)	(0.396, 0.412)	(0.412, 0.425)	(0.422, 0.436)	(0.472, 0.481)	(0.49
3	(0.538, 0.550)	(0.528, 0.538)	(0.510, 0.527)	(0.486, 0.509)	(0.468, 0.483)	(0.435, 0.447)	(0.413, 0.428)	(0.413, 0.435)	(0.431, 0.444)	(0.471, 0.484)	(0.492
4	(0.560, 0.574)	(0.553, 0.560)	(0.534, 0.554)	(0.513, 0.535)	(0.489, 0.506)	(0.463, 0.475)	(0.433, 0.446)	(0.435, 0.454)	(0.444, 0.460)	(0.475, 0.489)	(0.492
5	(0.580, 0.592)	(0.575, 0.589)	(0.564, 0.577)	(0.548, 0.560)	(0.529, 0.541)	(0.496, 0.507)	(0.463, 0.478)	(0.466, 0.481)	(0.475, 0.486)	(0.490, 0.501)	(0.493
6	(0.596, 0.612)	(0.599, 0.604)	(0.587, 0.600)	(0.580, 0.586)	(0.560, 0.563)	(0.533, 0.539)	(0.495, 0.501)	(0.508, 0.514)	(0.508, 0.516)	(0.504, 0.510)	(0.502
8	(0.579, 0.594)	(0.593, 0.597)	(0.582, 0.586)	(0.571, 0.579)	(0.554, 0.559)	(0.519, 0.530)	(0.484, 0.492)	(0.495, 0.504)	(0.497, 0.506)	(0.498, 0.503)	(0.493
10	(0.567, 0.575)	(0.561, 0.578)	(0.563, 0.574)	(0.556, 0.571)	(0.541, 0.558)	(0.516, 0.528)	(0.478, 0.491)	(0.490, 0.502)	(0.490, 0.508)	(0.494, 0.509)	(0.497
15	(0.516, 0.532)	(0.517, 0.530)	(0.514, 0.528)	(0.515, 0.526)	(0.51, 0.521)	(0.501, 0.516)	(0.488, 0.501)	(0.487, 0.503)	(0.489, 0.507)	(0.496, 0.505)	(0.493
20	(0.497, 0.507)	(0.497, 0.510)	(0.498, 0.513)	(0.496, 0.512)	(0.500, 0.511)	(0.488, 0.51)	(0.494, 0.509)	(0.490, 0.505)	(0.498, 0.508)	(0.487, 0.507)	(0.493

Let α_1 be the α level for player one and α_2 the α level for player 2. Also note that $\alpha = 0$ is equivalent to strategy A.

The data suggests that against a player using strategy A, $\alpha_1 = 6$ is the best alpha value to use with strategy B. In general, $\alpha_1 = 6$ gives the highest win rate and its confidence intervals are significantly different from $\alpha_1 = 8$ and $\alpha_1 = 5$ for all other values of α_2. As we increase α past this range, the confidence interval for the win rates approaches an interval around 50%.

Intuitively this means that if the opponent is one roll away from winning, we should always subtract from their score so that they cannot win. If we choose to decrease their score when they are further away from the goal, at $\alpha_2 = 20$ for example, we may potentially decrease our chances of winning but our chances will still be close to 50%.

So, if we know the opponent is using strategy B with $\alpha_2 = 6$, we could combat it with a large α_1. However $\alpha_1 = 6$ will have the same effect and is much better in other cases.

Strategy C

Suppose that instead of subtracting from the opponent's score when they are close to winning, we choose to subtract from the opponent's score if they are ahead by a certain amount. We will use strategy C, where we will subtract from the opponent's score if they are β points ahead.

Here are the win rates for strategy C against another player using strategy C for different values of β (Python3 C):

Player 1						Player 2					
β	0	1	2	3	4	5	6	8	10	15	3
0	(0.5, 0.5)	(0.034, 0.037)	(0.033, 0.036)	(0.033, 0.036)	(0.033, 0.037)	(0.033, 0.038)	(0.032, 0.036)	(0.033, 0.035)	(0.033, 0.037)	(0.034, 0.037)	(0.034,
1	(0.964, 0.967)	(0.499, 0.503)	(0.275, 0.285)	(0.219, 0.226)	(0.227, 0.230)	(0.252, 0.262)	(0.265, 0.281)	(0.302, 0.313)	(0.322, 0.335)	(0.332, 0.343)	(0.333,
2	(0.963, 0.967)	(0.713, 0.728)	(0.494, 0.509)	(0.424, 0.435)	(0.414, 0.437)	(0.421, 0.437)	(0.439, 0.450)	(0.474, 0.489)	(0.495, 0.510)	(0.506, 0.523)	(0.507,
3	(0.964, 0.968)	(0.766, 0.771)	(0.559, 0.574)	(0.497, 0.504)	(0.474, 0.490)	(0.480, 0.493)	(0.499, 0.518)	(0.538, 0.546)	(0.555, 0.566)	(0.566, 0.580)	(0.573,
4	(0.964, 0.967)	(0.773, 0.782)	(0.580, 0.584)	(0.514, 0.518)	(0.497, 0.503)	(0.503, 0.508)	(0.520, 0.523)	(0.550, 0.558)	(0.569, 0.586)	(0.586, 0.592)	(0.587,
5	(0.963, 0.966)	(0.742, 0.75)	(0.564, 0.579)	(0.505, 0.515)	(0.492, 0.497)	(0.492, 0.506)	(0.516, 0.523)	(0.542, 0.552)	(0.568, 0.586)	(0.578, 0.584)	(0.579,
6	(0.964, 0.968)	(0.721, 0.731)	(0.544, 0.563)	(0.484, 0.506)	(0.471, 0.483)	(0.477, 0.490)	(0.491, 0.509)	(0.529, 0.543)	(0.547, 0.56)	(0.556, 0.574)	(0.564,
8	(0.962, 0.967)	(0.684, 0.699)	(0.513, 0.526)	(0.452, 0.466)	(0.438, 0.450)	(0.440, 0.455)	(0.462, 0.475)	(0.486, 0.501)	(0.514, 0.529)	(0.524, 0.535)	(0.524,
10	(0.962, 0.967)	(0.668, 0.681)	(0.487, 0.501)	(0.426, 0.445)	(0.417, 0.434)	(0.425, 0.438)	(0.434, 0.455)	(0.477, 0.486)	(0.492, 0.504)	(0.501, 0.519)	(0.504,
15	(0.965, 0.968)	(0.659, 0.667)	(0.475, 0.491)	(0.417, 0.429)	(0.406, 0.418)	(0.408, 0.430)	(0.428, 0.437)	(0.456, 0.472)	(0.481, 0.498)	(0.490, 0.508)	(0.491,
30	(0.964, 0.967)	(0.657, 0.664)	(0.484, 0.497)	(0.419, 0.430)	(0.406, 0.419)	(0.410, 0.431)	(0.431, 0.441)	(0.459, 0.479)	(0.483, 0.499)	(0.493, 0.51)	(0.494,

Let β_1 correspond to player 1and β_2 correspond to player 2.

$\beta = 0$ is a losing strategy. With $\beta = 0$, we subtract if we are tied with the opponent. Because we start with the same score, we will always subtract. With this score, the best a player can hope for is a tie. In the case of $\beta_1 = \beta_2 = 0$, both players will always subtract and so every game will result in a tie. A tie is counted as half of a win, so the probability of 'winning' is exactly 0.5.

Because the winning score is 30 points, using $\beta = 30$ is equivalent to strategy A. The best β to use against a player using strategy A is $\beta = 4$ with a win rate in the interval (0.587, 0.591).

The best β overall to use when playing against another player using strategy C is $\beta_1 = 4$. In some cases it ties for the best win rate with $\beta = 3$ and $\beta = 5$, but it also outperforms the other β values in other cases.

This strategy is more punishing for $\beta = 0$ and $\beta = 1$. However it can be just as rewarding as strategy B in some cases.

Conclusion

Using strategy B at $\alpha_1 = 6$ yields, with 99.8% confidence, maximum win rates in the intervals (0.596,0.612), (0.599,0.604), and (0.587,0.600) for $\alpha_2 = 0$, $\alpha_2 = 1$, $\alpha_2 = 2$, respectively. These intervals are not significantly different but they each yield close to a 60% win rate. Strategy B also has a minimum win rate in the interval (0.495,0.501) with 99.8% confidence for $\alpha_1 = \alpha_2 = 6$.

Strategy B's lowest overall win rate is in the interval (0.394,0.398) with 99.8% confidence.

Using strategy C with $\beta = 0$ will always result in a loss or a tie. With β >0, strategy C has a lowest overall win rate in the range (0.219,0.226).

Using $\beta_1 = 4$ with $\beta_2 > 0$ leads to a maximum win rate close to 78% and a minimum win rate close to 50%.

Against a player who is using strategy A, strategy B with $\alpha_1 = 6$ is preferred. This strategy has close to a 60% win rate in the interval (0.596,0.612) with 99.8% confidence.
Strategy C with $\beta_1 = 4$ is a close second with a win rate in the interval (0.587,0.591).

This notebook has only scraped the surface of what is possible with these simulations. This notebook could be improved by using more game simulations to produce better confidence intervals and cleaner results. I would also like to analyze more combinations of strategies as well as different strategies, such as always subtracting after reaching a certain amount of points (this is similar to strategy C with a larger β). Also, it would be interesting to look at different variations of the game, such as variations where you must pick to add to your score or subtract from your opponent's score *before* you roll.

Appendix

Python3 A:

```
# Garrett Lawson, 2023
# MATH381

import random
import math
import numpy as np
import matplotlib.pyplot as plt

class Player:

    def __init__(self, score):
        self.score = score

WIN_SCORE = 30
MAX_TURNS = 50

numPlayers = 2 # Two player game. We could potentially change the game to include more
than two
                # players but the current code only supports two.
numGames = 10000
numTrials = 10

Player0WRs = np.zeros(numTrials)
simRates = np.zeros((numTrials, int(numGames/100)-1)) # make sure numGames divisible
by 100

for trial in range(numTrials):

    wins = np.zeros(numPlayers)

    for game in range(numGames): # game loop

        ### initialize players
        players = []
        for i in range(numPlayers):
            players.append(Player(0))

        #### Start playing the game
        currentTurn = 1
        currentPlayer = game % numPlayers ### start with a different player each time
        #print("Scores: [0, 0]")
        while(currentTurn<=MAX_TURNS):

            # each player takes a turn
            # roll the die
            dieRoll = random.randint(1, 6)
            #print("Player " + str(currentPlayer) + " rolls a " + str(dieRoll))

            # Player either adds to their score or subtracts from opponent's score
            if (currentPlayer in [0]):
                # Strategy for this player:
                # Always add
                players[0].score+=dieRoll

            if (currentPlayer in [1]):
                # Strategy for this player:
                # Always add
                players[1].score+=dieRoll

            # print the scores for each turn
            #print("Scores: [" + str(players[0].score) + ", " + str(players[1].score)
+ "]")

            # If the current player has reached the goal score, they win
            if (players[currentPlayer].score>=WIN_SCORE):
                wins[currentPlayer]+=1
                #print("Player " + str(currentPlayer) + " wins!")
                #print("Scores: [" + str(players[0].score) + ", " + str(players[1].s-
core) + "]")
                break
            elif (currentTurn==MAX_TURNS): # If we reach max turns, whoever has higher
score wins
                if (players[0].score>players[1].score):
                    wins[0]+=1
                elif (players[1].score>players[0].score):
                    wins[1]+=1
                else:
                    wins[0]+=0.5
                    wins[1]+=0.5
                break

            # next turn
            currentPlayer = (currentPlayer+1) % numPlayers
            currentTurn += 1

        ## game is over

        # print win probability for player 0 every so often
        #if ((game % 10000==0) and (game>0)):
         #   print(wins[0]*1./game)

        if ((game % 100==0) and (game>0)):
            simRates[trial][int(game/100)-1] = wins[0]*1./(game+1)
```

```
    ## output win percentage for all players
    “””print(“-------------------“)
    print(“Stats Over All Games”)

    print(wins)

    for i in range(numPlayers):
        print(i, “ “, wins[i]*1./numGames)”””

    Player0WRs[trial] = wins[0]*1./numGames

print(Player0WRs)


for i in range(numTrials):
    plt.plot(simRates[i])
plt.show()
```

Python3 B:

```
# Garrett Lawson, 2023
# MATH381

import random
import math
import numpy as np

class Player:

    def __init__(self, score):
        self.score = score

WIN_SCORE = 30
MAX_TURNS = 50

numPlayers = 2 # Two player game. We could potentially change the game to include more
than two
                # players but the current code only supports two.
numGames = 100000
numTrials = 10

Player0WRs = np.zeros(numTrials)

for trial in range(numTrials):

    wins = np.zeros(numPlayers)

    for game in range(numGames): # game loop

        ### initialize players
        players = []
        for i in range(numPlayers):
            players.append(Player(0))

        #### Start playing the game
        currentTurn = 1
        currentPlayer = game % numPlayers ### start with a different player each time
        #print(“Scores: [0, 0]”)
        while(currentTurn<=MAX_TURNS):

            # each player takes a turn
            # roll the die
            dieRoll = random.randint(1, 6)
            #print("Player " + str(currentPlayer) + " rolls a " + str(dieRoll))

            # Player either adds to their score or subtracts from opponent's score
            if (currentPlayer in [0]):
                scoreThresh = 0
                # Strategy for this player:
                # Always add unless the opponent is within scoreThresh points of win-
ning
                # and adding the roll would not result in a win
                if (players[1].score>=(WIN_SCORE-scoreThresh) and players[0].s-
core<WIN_SCORE-dieRoll):
                    players[1].score-=dieRoll
                else:
                    players[0].score+=dieRoll

            if (currentPlayer in [1]):
                scoreThresh = 0
                # Strategy for this player:
                # Always add unless the opponent is within scoreThresh points of win-
ning
                # and adding the roll would not result in a win
                if (players[0].score>=(WIN_SCORE-scoreThresh) and players[1].s-
core<WIN_SCORE-dieRoll):
                    players[0].score-=dieRoll
                else:
                    players[1].score+=dieRoll

            # print the scores for each turn
            #print("Scores: [" + str(players[0].score) + ", " + str(players[1].score)
+ "]")

            # If the current player has reached the goal score, they win
            if (players[currentPlayer].score>=WIN_SCORE):
                wins[currentPlayer]+=1
                #print("Player " + str(currentPlayer) + " wins!")
                #print("Scores: [" + str(players[0].score) + ", " + str(players[1].s-
core) + "]")
                break
            elif (currentTurn==MAX_TURNS): # If we reach max turns, whoever has higher
score wins
                if (players[0].score>players[1].score):
                    wins[0]+=1
                elif (players[1].score>players[0].score):
                    wins[1]+=1
                else:
                    wins[0]+=0.5
                    wins[1]+=0.5
                break

            # next turn
            currentPlayer = (currentPlayer+1) % numPlayers
            currentTurn += 1

        ## game is over

        # print win probability for player 0 every so often
        #if ((game % 10000==0) and (game>0)):
         #   print(wins[0]*1./game)
```

```
    ## output win percentage for all players
    “””print(“-------------------“)
    print(“Stats Over All Games”)

    print(wins)

    for i in range(numPlayers):
        print(i, “ “, wins[i]*1./numGames)”””

    Player0WRs[trial] = wins[0]*1./numGames

print(Player0WRs)
```

Python3 C

```
# Garrett Lawson, 2023
# MATH381

import random
import math
import numpy as np

class Player:

    def __init__(self, score):
        self.score = score

mynums = [0, 1, 2, 3, 4, 5, 6, 8, 10, 15, 20]

for row in mynums:

    print(“-----------------------------------------“)
    print(“Beta = “ + str(row))

    WIN_SCORE = 30
    MAX_TURNS = 50

    numPlayers = 2 # Two player game. We could potentially change the game to include 
more than two
                    # players but the current code only supports two.
    numGames = 10000
    numTrials = 10

    Player0WRs = np.zeros(numTrials)

    for trial in range(numTrials):

        wins = np.zeros(numPlayers)

        for game in range(numGames): # game loop

            ### initialize players
            players = []
            for i in range(numPlayers):
                players.append(Player(0))

            #### Start playing the game
            currentTurn = 1
            currentPlayer = game % numPlayers ### start with a different player each
time
            #print(“Scores: [0, 0]”)
            while(currentTurn<=MAX_TURNS):

                # each player takes a turn
                # roll the die
                dieRoll = random.randint(1, 6)
                #print(“Player “ + str(currentPlayer) + “ rolls a “ + str(dieRoll))

                # Player either adds to their score or subtracts from opponent’s score
                if (currentPlayer in [0]):
                    scoreBuffer = row
                    # Strategy for this player:
                    # Always add unless the opponent is at least scoreBuffer points
ahead
                    # and adding the roll would not result in a win
                    if (players[1].score>=(players[0].score+scoreBuffer) and player-
s[0].score<WIN_SCORE-dieRoll):
                        if (players[1].score-dieRoll<0):
                            players[1].score=0
                        else:
                            players[1].score-=dieRoll
                    else:
                        players[0].score+=dieRoll

                if (currentPlayer in [1]):
                    scoreBuffer = WIN_SCORE # player 2 always adds
                    # Strategy for this player:
                    # Always add unless the opponent is at least scoreBuffer points
ahead
                    # and adding the roll would not result in a win
                    if (players[0].score>=(players[1].score+scoreBuffer) and player-
s[1].score<WIN_SCORE-dieRoll):
                        if (players[0].score-dieRoll<0):
                            players[0].score=0
                        else:
                            players[0].score-=dieRoll
                    else:
                        players[1].score+=dieRoll

                # print the scores for each turn
                #print(“Scores: [“ + str(players[0].score) + “, “ + str(players[1].s-
core) + “]”)

                # If the current player has reached the goal score, they win
                if (players[currentPlayer].score>=WIN_SCORE):
                    wins[currentPlayer]+=1
                    #print(“Player “ + str(currentPlayer) + “ wins!”)
                    #print(“Scores: [“ + str(players[0].score) + “, “ + str(player-
s[1].score) + "]")
                    break
                elif (currentTurn==MAX_TURNS): # If we reach max turns, whoever has
higher score wins
                    if (players[0].score>players[1].score):
                        wins[0]+=1
                    elif (players[1].score>players[0].score):
                        wins[1]+=1
                    else:
                        wins[0]+=0.5
                        wins[1]+=0.5
                    break

                # next turn
                currentPlayer = (currentPlayer+1) % numPlayers
                currentTurn += 1

            ## game is over

            # print win probability for player 0 every so often
            #if ((game % 10000==0) and (game>0)):
             #   print(wins[0]*1./game)

        ## output win percentage for all players
        """print("-------------------")
        print("Stats Over All Games")

        print(wins)

        for i in range(numPlayers):
            print(i, " ", wins[i]*1./numGames)"""

        Player0WRs[trial] = wins[0]*1./numGames

    print(Player0WRs)
    print("("+str(round(min(Player0WRs),3))+", "+str(round(max(Player0WRs),3))+")")
```