Monte Carlo Simulations for Predicting Tennis Match Outcomes: A Comparative Analysis of Grand Slam and Regular Match Formats

November 20, 2023

Abstract

In this research, we present a Monte Carlo simulation approach to evaluate outcome probabilities and score distributions in tennis matches between two players, designated as Player A and Player B. Unlike traditional point-based simulations, our method focuses on simulating complete games within a match, where a game is won by the first player to reach four points or more with a margin of at least two points. The simulation continues until a player wins according to the rules of tennis, either in a regular match format or a Grand Slam format, where the number of sets required to win differs.

Each serve is simulated with varying probabilities for targeting the left, center, and right zones of the opponent's court, with distinct probabilities assigned based on the server's court position, left or right. The service alternates between the left and right service boxes in accordance with official tennis rules. The outcome probabilities for serves and returns are derived from historical match data, which includes frequency-based estimates of winning a point when serving to each zone.

One million simulations are conducted to provide a robust statistical basis for estimating the likelihood of different match scores and determining the winner. This simulation allows us to assess the advantage conferred by the format of the match, be it a regular match or a Grand Slam match, on each player. Our findings aim to contribute valuable insights into strategic preparations for tennis athletes and a deeper understanding for enthusiasts of how match format influences player performance and match dynamics.

Methodology

To effectively simulate tennis match outcomes and score distributions for the given project, the following methodology is structured into several key components.

Initialization of Player Data

- Set up each player (Player A and Player B) with specific probabilities for serving to the left, center, and right service boxes. These probabilities are differentiated based on the side of the court from which the player is serving.

- Determine the probability of winning a point when serving to each of these areas based on historical performance data.

Simulation Structure

- Game Simulation: A game is simulated by playing points until one player reaches at least four points or more and is ahead by at least two points.

- Set Simulation: A set is simulated by playing games until one player wins six games with a margin of two, or more games, or wins a tiebreak if the game score reaches 6-6.

- Match Simulation: A match is simulated by playing sets until a player wins the required number of sets; three out of five for Grand Slam matches, or two out of three for regular matches.

Point Simulation Process

- The serving player is randomly determined at the start of each game.

- Serve direction is randomly chosen based on the serving player's left or right position and their corresponding serve direction probabilities.

- The outcome of the serve (point won or lost) is determined using the player's historical win probability for the chosen serve direction.

- The server alternates between the left and right service boxes with each new game.

Repetition and Data Collection

- The complete match simulation (point, game, set) is repeated 1 million times to ensure statistical robustness.

- The results of each simulation, including the final score and the winner, are recorded.

Statistical Analysis

- From the simulation data, calculate the frequency of different match scores and the probabilities of each player winning under regular and Grand Slam match conditions.

- Analyze how often matches go to a tiebreak under both conditions.

Probability Distribution and Visualization

- Use the collected data to create probability distribution graphs for different match scores.
- Compare the probability distributions between regular and Grand Slam formats to determine which format may offer an advantage to a particular player.

This methodology leverages historical match data to simulate realistic tennis match scenarios and provides a comprehensive analysis of how different match formats can affect the outcomes of matches between two players.

Pseudocode

Algorithm 1: Monte Carlo Prediction

```
while condition do
    Do something;
    if another condition then
        Do something else;
for each element in a list do
    Process the element;
```

Python code implementation

```python
# -*- coding: utf-8 -*-
"""Tennis_Monte_Carlo.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/17dZHXh7NoG_Br_N6s62sFBJr80YR6VlQ
"""

import numpy as np
import matplotlib.pyplot as plt
import time
import multiprocessing
from functools import partial

# Constants
N_SIMULATIONS = 100000
POINTS_TO_WIN_GAME = 4
POINTS_TO_WIN_TIEBREAK = 7
POINTS_TO_WIN_TIEBREAK_SLAM_LAST_SET = 10 # The last set (or deciding set, or 5th set) tiebreak is 10 points
```

```
MIN_MARGIN = 2
SETS_TO_WIN_MATCH_REGULAR = 2
SETS_TO_WIN_MATCH_GRAND_SLAM = 3

# Players' serving probabilities for left/right service boxes and winning the point
# Players' serving target probabilities for left/right service boxes and targeting a particular direction
player_serve_win_probs = {
    'A': {'left': (0.4, 0.3, 0.3), 'right': (0.2, 0.5, 0.3)},
    'B': {'left': (0.5, 0.35, 0.25), 'right': (0.25, 0.35, 0.4)}
}
player_serve_target_probs = {
    'A': {'left': (0.1, 0.6, 0.3), 'right': (0.5, 0.1, 0.4)},
    'B': {'left': (0.7, 0.1, 0.2), 'right': (0.25, 0.3, 0.45)}
}

def simulate_point(server, serving_from):
    # Determine the winning probablity based on the server and the serving side (left or right)
    serve_win_probs = player_serve_win_probs[server][serving_from]

    # Determine the target direction (left, middle, or right)
    serve_direction_probs = player_serve_target_probs[server][serving_from]
    target_direction = np.random.choice(['left', 'middle', 'right'],
                                        p =[serve_direction_probs[0], serve_direction_probs[1],
                                            1-serve_direction_probs[0]-serve_serve_direction_probsprobs[1]]
                                        )

    # Determine if the server wins the point based on the chosen and targetting direction
    target_index = 0 if target_direction == 'left' else 1 if target_direction == 'middle' else 2
    point_winner = server if np.random.rand() < serve_win_probs[target_index] else 'A' if server == 'B' else 'B'
    return point_winner

def simulate_game(initial_server):
    score = {'A': 0, 'B': 0}
    server = initial_server
    serving_from = 'right' # alternates between 'left' and 'right'
    while True:
        point_winner = simulate_point(server, serving_from)
```

```
        score[point_winner] += 1

        # Check for game winner
        if score[point_winner] >= POINTS_TO_WIN_GAME and (score[point_winner] - score['A' if point_winner == 'B' else 'B']) >= MIN_MARGIN:
            return point_winner

        # Alternate serving side
        serving_from = 'left' if serving_from == 'right' else 'right'

def simulate_tiebreak(last_server):
    set_score = {'A': 0, 'B': 0}
    server = 'A' if last_server == 'B' else 'B'
    serving_from = 'right' # alternates between 'left' and 'right'

    # First server only serves for one point and then switch server
    point_winner = simulate_point(server, serving_from)
    score[point_winner] += 1
    serving_from = 'left' if serving_from == 'right' else 'right'
    server = 'A' if server = 'B' else 'B'

    while True:
        # Every player serves for two points
        for _ in range(2):
            point_winner = simulate_point(server, serving_from)
            score[point_winner] += 1

            # Check for tiebreak winner
            if score[point_winner] >= POINTS_TO_WIN_TIEBREAK and (score[point_winner] - score['A' if point_winner == 'B' else 'B']) >= MIN_MARGIN:
                return point_winner

            # Alternate serving side
            serving_from = 'left' if serving_from_A == 'right' else 'right'

        server = 'A' if server = 'B' else 'B'

def simulate_set(initial_serve):
    set_score = {'A': 0, 'B': 0}
    server = initial_serve
    while True:
        game_winner = simulate_game(server)
        set_score[game_winner] += 1

```

```
        # Check for set winner
        if (set_score[game_winner] >= 6 and (set_score[game_winner] - set_score['A' if game_winner == 'B' else 'B']) >= MIN_MARGIN) or set_score[game_winner] == 7:
            initial_server = server # pass the last server to the simulate_match function
            return game_winner

        # Check for tiebreak
        if (set_score[game_winner] == 6 and set_score['A' if game_winner == 'B' else 'B'] == 6)
            simulate_tiebreak(server)

        server = 'A' if server == 'B' else 'B'

def simulate_match(sets_to_win):
    match_score = {'A': 0, 'B': 0}
    initial_server = random.choice(['A', 'B'])

    while True:
        set_winner = simulate_set(initial_server)
        match_score[set_winner] += 1

        if match_score[set_winner] == sets_to_win:
            return set_winner

        # The initial server of the new set is the opposite of the last server from the last set
        # i.e. if A served for the last game in the last set, the initial server of the new set is B
        initial_server = 'A' if initial_server == 'B' else 'B'

# Helper function to perform parallel simulations for a given number of simulations and sets to win
def simulate_matches(num_simulations, sets_to_win):
    match_results = {'A': 0, 'B': 0}
    for _ in range(num_simulations):
        winner = simulate_match(sets_to_win)
        match_results[winner] += 1
    return match_results

# Helper function to aggregate the results from all parallel processes
def aggregate_results(results):
    total_results = {'A': 0, 'B': 0}
    for result in results:
        total_results['A'] += result['A']
        total_results['B'] += result['B']
```

```
    return total_results

# Start the timer
start_time = time.time()

# Determine the number of processes to create based on the available CPU cores
N_PROCESSES = multiprocessing.cpu_count()
# Calculate the number of simulations each process will handle
chunk_size = N_SIMULATIONS // N_PROCESSES

# Create partial functions with the 'sets_to_win' argument already set
simulate_regular = partial(simulate_matches, sets_to_win=SETS_TO_WIN_MATCH_REGULAR)
simulate_grand_slam = partial(simulate_matches, sets_to_win=SETS_TO_WIN_MATCH_GRAND_SLAM)

# Create a pool of processes and distribute the workload
with multiprocessing.Pool(processes=N_PROCESSES) as pool:
    # Map the partial functions to the pool for parallel processing
    regular_results = pool.map(simulate_regular, [chunk_size for _ in range(N_PROCESSES)])
    grand_slam_results = pool.map(simulate_grand_slam, [chunk_size for _ in range(N_PROCESSES)])

# Aggregate the results from all processes
regular_match_results = aggregate_results(regular_results)
grand_slam_match_results = aggregate_results(grand_slam_results)

# Calculate the probabilities based on the aggregated results
regular_match_probabilities = {player: wins / N_SIMULATIONS for player, wins in regular_match_results.items()}
grand_slam_match_probabilities = {player: wins / N_SIMULATIONS for player, wins in grand_slam_match_results.items()}

# Stop the timer and calculate elapsed time
end_time = time.time()

# Print the calculated probabilities and the elapsed time
print("Regular Match Probabilities: ", regular_match_probabilities)
print("Grand Slam Match Probabilities: ", grand_slam_match_probabilities)
print("Time elapsed: ", end_time - start_time)

# Data for stacking
regular_a = regular_match_probabilities['A']
regular_b = regular_match_probabilities['B']
```

```
grand_slam_a = grand_slam_match_probabilities['A']
grand_slam_b = grand_slam_match_probabilities['B']

# Positions of the bars on the x-axis
ind = np.arange(2)

# Heights of the A and B portions
p1 = [regular_a, grand_slam_a]
p2 = [regular_b, grand_slam_b]

# Plotting
fig, ax = plt.subplots()

# Stack 'A' and 'B' for each match type
bars_a = ax.bar(ind, p1, label='Player A Wins', color='blue')
bars_b = ax.bar(ind, p2, bottom=p1, label='Player B Wins', color='
    orange')

# Adding labels and title
ax.set_xlabel('Match Type')
ax.set_ylabel('Probabilities')
ax.set_title('Probability of Winning for Player A and B')
ax.set_xticks(ind)
ax.set_xticklabels(['Regular', 'Grand Slam'])
ax.set_yticks(np.arange(0, 1.1, 0.1))
ax.set_ylim([0, 1]) # Set y-axis to go from 0 to 1
ax.legend()

# Adding the percentage on top of the bars
for r1, r2 in zip(bars_a, bars_b):
    h1 = r1.get_height()
    h2 = r2.get_height()
    ax.text(r1.get_x() + r1.get_width() / 2., h1 / 2., f'{h1:.1%}', ha='
        center', va='center', color='white')
    ax.text(r2.get_x() + r2.get_width() / 2., h1 + h2 / 2., f'{h2:.1%}',
         ha='center', va='center', color='white')

# Show plot
plt.show()
```