The Gamble of the First Basket: A Study on Betting the First Scorer in NBA Games

Hugo De Vere

Publication date: 8/14/2023

Abstract—This study aimed to analyze the viability of placing bets on the first player to score in a basketball game by examining multiple strategies. The motivation behind this focus is the perceived increased randomness in betting on the first scorer compared to conventional money-line wagers. Given the limited access to only five games of historical data, the research approximated casino odds using a function that classified players by position and recent scoring performance. This methodology was refined by optimizing player odds within a set range, mirroring typical casino measures. Notably, the broad categorization by the NBA of players into three categories posed challenges in truly discerning player capabilities, as evidenced by disparities between our derived odds and actual casino odds. Two key insights emerged: the significant fees imposed by casinos and the sophisticated models they employ, which often surpass the depth of freely available NBA statistics. While the research sought to exploit betting inefficiencies, it underscored the complexities inherent in replicating casino odds using public domain data. Further collaboration and access to richer datasets are encouraged for future endeavors.

Keywords—Sports Betting, NBA Betting, Betting Strategies, Probability Theory

I. Introduction

Like financial trading, sports betting also functions in a highly competitive environment. The challenge of consistently achieving profits in sports betting is often amplified due to the discrepancy between available information and expertise between sportsbooks and individual bettors. Just as strategies are developed to outsmart opponents in financial markets, similar tactics are employed to gain an advantage in sports betting. Comparable to the unpredictability of stock prices, the randomness in sporting events mirrors this unpredictability. Furthermore, as demonstrated in Moskowitz [2021], the behavioral biases that influence stock movements can similarly affect sports betting odds, highlighting the parallel influence of bettor psychology on betting prices.

In this paper, I introduce a different approach from existing literature which predominantly centers on money-line bets (wagers on the outright winner of a match). This research explores various strategies to analyze the practicality of placing bets on the first player to score in a basketball game. The decision to focus on this area stems from the observation that betting on the first scorer tends to exhibit greater randomness compared to money-line wagers and, thus, might present more inefficiencies that can be leveraged against sportsbooks. The paper is organized into two major sections: The first section dives into the probability theory behind determining the odds for who will score the first basket, including an exploratory analysis and a methodology to extrapolate historical casino odds for the first scorer. The second section implements a series of basic betting strategies.

The paper concludes with a summary of the key findings and suggests directions for future research. The intention is to contribute to a comprehensive understanding of sports betting and to offer practical insights into this intriguing domain.

Contact data: Hugo De Vere, hugo.devereb@gmail.com

II. Nature of Betting on the First Scorer

a. Probability Theory Behind Determining Odds

Predicting who will score the first basket in an NBA game is complex. It's more uncertain than simply forecasting the winning team. This uncertainty arises from the fact that ten starting players might score, and the first basket happens quickly, leaving less time for the player's true abilities to influence the outcome. In probability theory, this uncertainty is referred to as "randomness." In this case, we're dealing with a specific form called "discrete randomness" since there are a finite number of possible outcomes.

Now, if we make a simplified assumption that each player is equally likely to score the first basket, we can begin to model this randomness. Under these premises, the probability for each player n scoring first can be determined as follows:

$$\text{player}_n\{\text{scoring first ball}\} = \frac{1}{10} \tag{1}$$

Solving for player n, we find a 10% chance for each player to score the first basket. Using the underdog American odds formula (which is applicable when the probability is less than 50%), we can convert this probability into casino odds as follows:

$$\begin{aligned}\text{American Odds} &= \frac{100 - \text{probability} \times 100}{\text{probability}} \\ &= \frac{10}{100 - 10} \times 100 = 900\end{aligned} \tag{2}$$

Equation (2) calculates the casino odds as +900. This implies that for every $100 bet, a gain of $900 is expected under the assumption that the casino doesn't introduce any spread and all players have equal chances of scoring.

Having considered the odds for a single game, we might now extend our strategy to betting on two games simultaneously—a tactic I will explore later. In probability theory, two events are independent if one occurrence does not affect the probability of the other. Therefore, the probability of these independent outcomes is simply the product of their individual probabilities.

For example, if we want to consider the probability of Adebayo scoring against the Celtics and LeBron scoring against the Heat at the same time. Since each player has a 10% chance the calculation is as follows:

$$\frac{1}{10} \times \frac{1}{10} = \frac{1}{100} \tag{3}$$

Now, using this probability, we can calculate the casino odds for such an event using (2):

$$\text{Odds} = \frac{100 - 1 \times 100}{1} = 9900 \tag{4}$$

As shown in Equation (4), the calculated American odds for betting on these linked events are +9900. This means that a $100 bet could potentially win $9,900. However, now that we've explained how odds are figured out, it's important to understand that the actual casino odds can differ a lot. There are several reasons for this difference:

- Probability Distribution: The probability is not uniformly distributed across all players. Individual players may have unique chances of making the first basket, and significant differences in team strength can further influence these odds. For example, an underdog lineup might have a slightly lower chance of scoring first.

- Unpredictable Factors: Even though the starting lineup is usually known 20-30 minutes before the game, unforeseen circumstances can alter the odds. Factors such as injuries or last-minute changes to the lineup introduce an element of randomness, meaning that bench players might have odds ranging from +1500 to +2000. This uncertainty adds complexity to the calculation of casino odds and contributes to variations from the theoretical odds of +9900.

The calculated odds can vary from casino to casino, but usually, the difference is minimal due to the existence of arbitrage. On average, odds for the players on the field will range from +400 to +1400.

b. Exploratory Analysis of First Scorer

In this section, I aim to examine the nature of the first scoring basket by probing and affirming basic assumptions that might logically be anticipated. Before diving into these assumptions, it's important to outline the dataset on which this study is based. I gathered data through the NBA API, encompassing regular and preseason games as well as playoffs from 2015 to the 2022-23 season, totaling 10,000 games. The exploratory analysis will focus exclusively on the training data, avoiding lookahead bias. Following a well-regarded rule of thumb in quantitative finance to avoid overfitting, the dataset was split using a 60/40 training/testing ratio, with the training data extending up to January 1, 2020.

The analysis begins with the hypothesis that the height difference between the jumpers at the game's start could significantly influence the jump's outcome. More specifically, does a greater height infer an advantage in winning the initial jump? While conventional wisdom might suggest that a taller stature equals better jumping ability, the data did not corroborate this view. In 37% of instances, height advantage correlated with winning the jump, as illustrated in Figure 1.

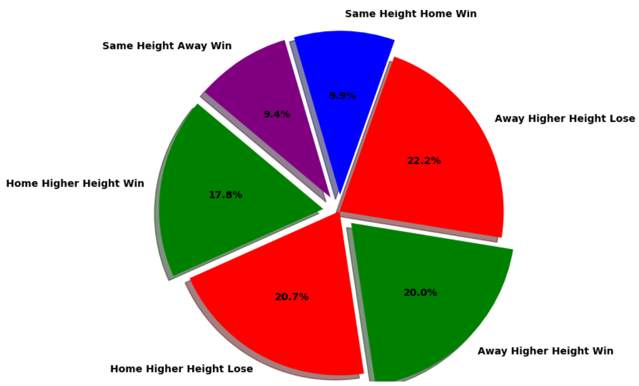

Fig. 1: Effect of Player Height on Initial Jump Results

Additionally, one might assume that the height difference between two jumpers is crucial. To examine this claim, I segmented the observations into four quantiles. Interestingly, the height disparity between the jumpers was minimal in most cases (See Figure 2). However, the results were consistent with prior findings, focusing on the 4th quantile, where the difference in player heights exceeded 0.17 meters. As illustrated in Figure 3, a height advantage does not necessarily translate to superior jumping ability over an opponent.

A second hypothesis is predicated on the belief that winning the initial jump often leads to scoring first.

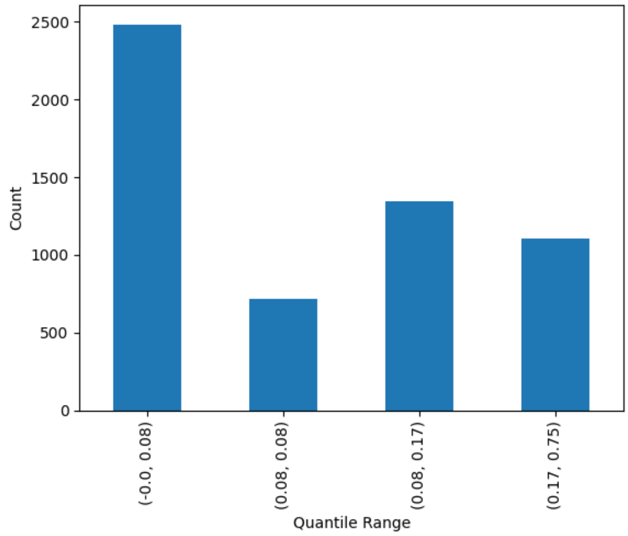

Fig. 2: Height Difference Quantile Ranges among Jumpers

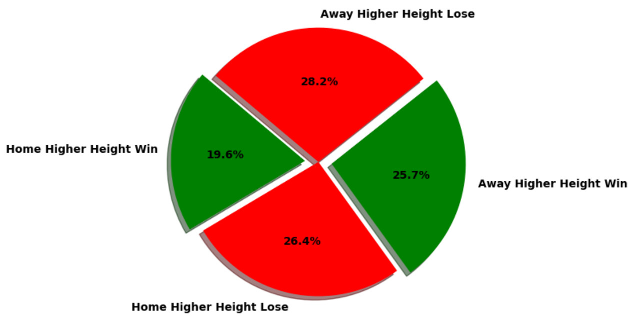

Fig. 3: Effect of Player Height on Initial Jump Results: 4th Quantile Range Height Difference

The data affirmed this belief (see Table 1), with 64.77% of teams that won the initial jump also scoring first. Thirdly, there's a common belief in sports that home teams usually have an edge against away teams when playing in their own hometown. However, as found in Table 1, this idea didn't hold up when we looked at whether playing at home could predict who wins the initial jump or scores first. The results were nearly evenly split with 50-51% odds.

Lastly, I explore the assumption that forwards would likely score more than guards or centers due to their dominant possession time and role as play creators. However, as shown in Figure 4, even when normalizing each position by their respective player counts on the field (e.g., two forwards and two guards), this notion doesn't hold. Contrarily, centers exhibit a significantly higher occurrence of scoring first. This trend is mirrored in the upcoming chapters, where historical casino data consistently shows centers with higher odds compared to forwards.

Table 1: Results of Hypotheses Testing

Hypothesis	**% of Occurrences**
Teams that won the initial jump are more likely to score the first basket	64.77%
Home teams that won the initial jump	49.95%
Home teams are more likely to score the basket	51.00%

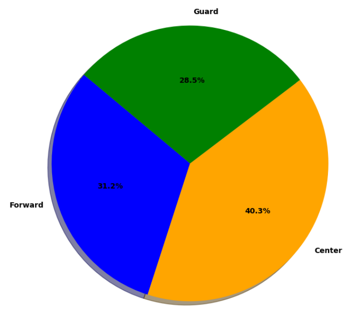

Fig. 4: Distribution of First Scorers by Position in NBA Matches

c. Extrapolating Historical Casino Odds for First Scorers

This section is pivotal, as it will dictate the viability of our betting strategies (See all code functions in b). Our primary objective is to approximate casino odds by analyzing a sample of games. Assuming casino odds and public perception operate under the "evidential" interpretation, I consider the odds to be a function of the moving average of the last x games. For example, I have analyzed Nikola Jokic and Jimmy Butler's PTS lines set by BetMGM and the corresponding realized results. Typically, a casino sets the over/under PTS line in such a way that either side has around a -100 to -120 payout [1].

Table 2: Players Casino PTS line versus realized

	Game 1	2	3	4	5
Nikola Jokic					
BetMGM line	27.5	27.5	28.5	30.5	29.5
Realized	27	41	32	23	28
Jimmy Butler					
BetMGM line	27.5	26.5	25.5	27.5	26.5
Realized	13	21	28	25	21

In order to extrapolate this lookback window, we can establish the following expression to estimate the

[1] I have obtained the over/under points (PTS) line for each player in the NBA finals games, as set by BetMGM, via the website WYMT Sports. On average, these lines corresponded to payout odds ranging from -100 to -120.

casino's moving average (MA) line:

$$\text{Estimated Casino MA line} = \frac{\text{realized}_n + \ldots + \text{realized}_{n+5}}{x} = \text{game 5 line} \quad (5)$$

$$x \cdot \text{game 5 line} = \text{realized}_n + \ldots + \text{realized}_{n+5} \quad (6)$$

$$x = \frac{\text{realized}_n + \ldots + \text{realized}_{n+5}}{\text{game 5 line}} \quad (7)$$

Where n is the game number, and we aim to solve for x, the length of the moving average window.

Utilizing equation (5) with statistics for Nikola and Butler, as presented in Table 2, I derived moving averages of 5.12 and 4.08, respectively. This calculation underscores the "Evidential" approach to casino odds. Initially, I assumed that the odds of a player scoring first would correlate with the player's average first-scoring frequency over the past five games. However, I observed this five-game span insufficient for determining a "first scorer" and subsequently expanded the requirement to at least 20 observations per season.

To derive casino odds for each player position, I used insights from historical odds statistics from three games this season, as shown in Table 3, and incorporated specific constraints influenced by positional variations. The odds for every player derive from the universal average, approximately +750. A player's historical performance and position dictate subsequent modifications to this base value.

The odds are computed using the following formula:

$$\text{Odds} = 750 + 100 \cdot S + P \quad (8)$$

Where:

- S represents the player's inverse ranking, an integer ranging from 0 to 4. For instance, a top-ranked player will receive a value of 0, thus neutralizing the +100 multiplier. Lower ranks signify decreased scoring probabilities and, consequently, higher odds.
- P is an adjustment based on the player's position in the lineup.

The P adjustment is determined as:

$$\begin{cases} 0 & \text{for forwards} \\ -200 & \text{for centers} \\ +200 & \text{for guards} \end{cases}$$

Post-calculation, I impose certain constraints on the odds to ensure they mirror practical boundaries, informed by Table 3. To mitigate the influence of outliers, especially from players not frequently in starting lineups, the maximum boundary is set at the 75th percentile. This lends a more conservative and pragmatic approach to casino odds.

Finally, the common adage in Wall Street and sports betting is "There's no such thing as a free lunch." The question then arises: How do we validate the accuracy of our odds calculations? The answer lies in the concept that when a bet is placed on all ten starting lineups, the potential for profitability should be negligible. To adjust for this, I formulated a linear optimization approach.

The primary goal of this optimization problem is to consistently adjust player odds such that their aggregate implied probabilities fall within a specific target range, namely 110% to 120%. This range was selected based on an analysis of how casinos set odds for starting players. For instance, examining the odds for Game 5 of the 2022-23 finals reveals that the average implied probability across all casinos hovers around 116%[2]. By aligning our starting lineup odds around a 110-120% range, I argue that this modification accounts for the house margin and safeguards, thus the extrapolation is more conservative.

Mathematically, the updated optimization is delineated as:

$$\text{Minimize:} \quad \sum_i (o_i - \delta)^2 \quad (9)$$

$$\text{Subject to:} \quad 1.1 \leq \sum_i \frac{100}{o_i - \delta + 100} \leq 1.20 \quad (10)$$

$$400 \leq o_i - \delta \leq 1500 \quad \forall i \quad (11)$$

Where:

- δ represents the uniform reduction applied to all players' odds.
- o_i symbolizes the original odds assigned to the i-th player.
- The constraints ensure that the aggregated implied probabilities fall within the range of 110% and 120%.
- Bounds are imposed on the uniformly adjusted odds to ensure they remain between 400 and 1500.

Once the odds for each game in our dataset were adjusted using Eq 11, I conducted a Monte Carlo simulation to evaluate this optimization's performance. This entailed 1,000 iterations of 32 consecutive bets, with an initial bankroll of $1000. A $1 bet was placed on every starting player in every bet iterated. Figure 5 presents the results of this simulation. Notably, none of the simulations ended with a bankroll surpassing the initial amount, indicating the robustness of our odds extrapolation. Furthermore, the statistics of our extrapolated odds, drawn from a dataset of 9,000 games, are detailed in Table 4.

When assessing the statistics of our extrapolated odds, derived from a dataset spanning 9,000 games (as depicted in Table 4), against our sample (See Table 3) and the odds from Game 5 (See Table 5), there's a notable discrepancy on ranges. On average, there's a difference of 200 between my extrapolated odds and the casino's actual odds, a gap that widens to as much as 400 when considering guards.

[2] Details for the Game 5 finals odds can be viewed at: https://www.bettingpros.com/articles/nba-finals-game-5-first-basket-scorer-player-prop-odds-heat-vs-nug

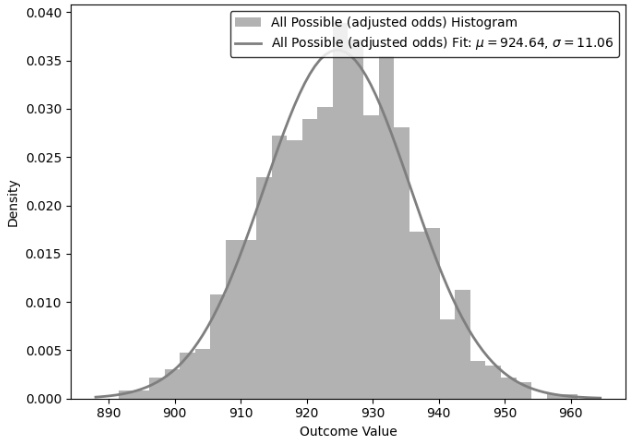

Fig. 5: Distribution Fit of 1,000 Simulations: 32 Consecutive Bets on All 10 Starting Players

The main issue lies in the NBA's API categorization of players. The league broadly classifies players into just three categories, which, despite our comprehensive methodology, doesn't adequately capture the nuanced capabilities of players based on their specific on-court roles. For instance, the guard category encompasses point guards and shooting guards, while the forwards include power forwards and small forwards. Within these distinctions, each has a player type more inclined towards shooting than the other. This categorization limits our ability to validate the presence of inefficiencies accurately. As a result, while our strategies can be evaluated in relation to each other, they must be interpreted with caution.

Table 3: Our sample of Odds Statistics by Player Position

	Forwards (F)	Centers (C)	Guards (G)
mean	820	570	960
std	360	170	350
min	480	400	500
25%	550	500	730
50%	750	550	900
75%	900	550	1100
max	1400	850	1700

Table 4: Statistics of Extrapolated Casino Odds for Each Player Positions

	Forwards (F)	Centers (C)	Guards (G)
mean	770	480	970
std	30	25	30
min	700	450	900
25%	725	450	925
50%	775	500	1000
75%	800	500	1000
max	800	500	1000

	Extrapolated	Actual	Difference
Porter (F)	650	830	-180
Gordon (F)	800	920	-120
Jokic (C)	500	432	68
Pope (G)	950	1380	-430
Murray (G)	1000	550	450
Butler (F)	650	640	10
Love (F)	800	1040	-240
Adebayo (C)	500	580	-80
Strus (G)	950	870	80
Vincent (G)	1000	1180	-180
Total	7800	8422	-622

Table 5: American odds for DEN (above) and MIA (below). Extrapolated vs Average Casino odds.

III. Betting Strategies

a. Single Game Strategies

I will evaluate two distinct betting strategies using the derived casino odds in this section. The initial strategy is based on player positions, while the latter leverages scoring rankings. Both strategies will be assessed using a fixed bet-sizing method of 2% of total capital.

With that said, since we're using estimated odds, it is essential to validate the effectiveness of our strategies. First, to ensure robustness, I eliminated early-season games in our backtest to avoid some of the existing early-season inefficiencies on our odd setting. Second, I will benchmark the results of all our betting strategies against a reference: betting on all team players using the odds previously provided (see Figure 5).

In addition, I will again employ a Monte Carlo simulation approach for a robust evaluation. Specifically, I will run 1,000 simulations of 32 consecutive bets for each strategy. The rationale behind the Monte Carlo simulation is to gauge the susceptibility of our betting strategy to random fluctuations. By examining a range of potential betting scenarios that could have plausibly occurred, I aim to measure how liable to chance and randomness each betting strategy is.

Concluding our evaluation methodology, the betting strategies' efficacy will be gauged using two criteria: the distribution fit of the strategy relative to the benchmark and the probability of attaining a profitable bankroll

1. Position-Based Strategy

Our exploratory analysis indicated that centers and forwards are the most likely positions to score. Based on this insight, I will employ several strategies: betting solely on each team's centers, forwards, and guards. Additionally, combined approaches: betting on both forwards-guards and guards-centers pairs. The outcome distributions from these strategies, using a fixed bet of 2% of total captial, can be found in Table 6, Figure 6.

Unfortunately, the strategies didn't perform well.

Strategy	Probability (%) > $1000	W/L
Forwards	3	37.0
Centers	1.90	-
Guards	26.10	37.0
Forw.-Centers	0	63.0
Guards-Centers	5.10	63.0
All Possible	0.00	100.0

Table 6: Position-based Strategies Results

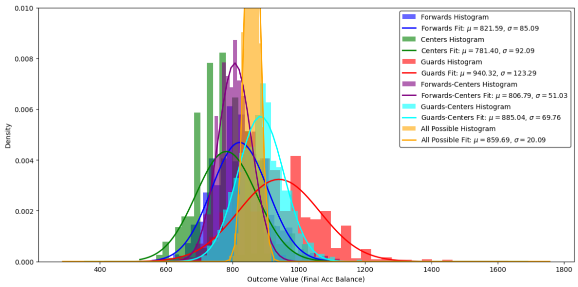

Fig. 6: Distribution Fit of 1,000 Simulations of Each Strategy using Fixed Bet Size

Every approach exhibited a greater than 56% likelihood of depleting half of the starting bankroll. Comparatively, wagering on guards might be a promising strategy in relation to the "betting all possible" strategy. However, this superior performance of the guard strategy is attributed to the marked difference observed between our projected odds and those of the actual casino.

2. Ranking Strategy

Drawing parallels with momentum-based strategies in financial trading, the key assumption here is that past performance can be indicative of future results. The percentage score will be our ranking metric. I implemented three strategies: betting on the top 1, top 2, and top 3 players in each team per game. Players are ranked by their scoring percentage over the last 20 games, mirroring the methodology used for odds extrapolation. The outcomes are presented in Table 7 and Figure 7.

Strategy	Probability (%) > $1000	W/L
Top 1 Player in Each Team	0.40	23.0
Top 2 Players in Each Team	0.00	44.0
Top 3 Players in Each Team	0.00	64.0

Table 7: Ranked-based Strategy Results

Looking at the results, there's a notable dip in performance when compared with position-based strategies. Despite an overall boost in win-loss ratios (adjusted for the number of players we bet on), the projected casino odds fall short of breakeven. Of all the approaches, betting on the top-ranked player in each game was the least bad, though the probability of positive returns is almost non-existent at 0.40

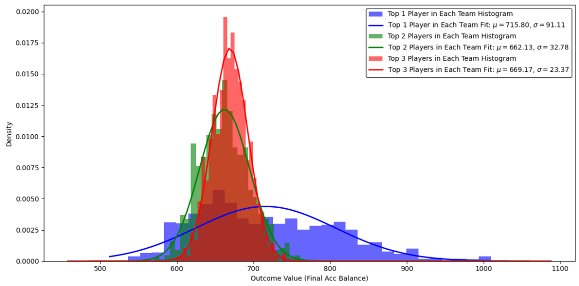

Fig. 7: Distribution Fit of 1,000 Simulations of Each Strategy using Fixed Bet Size

b. Parlay Betting Strategies

In this section, I explore strategies for parlay betting, which involves wagering on multiple events occurring concurrently. The goal is to bet on the first scorer in three games played on the same day. However, given our assumption that we know the starting lineup before placing bets, we only consider games played on the same day.

As discussed in the initial section, casinos treat these events as independent when calculating the odds. As an illustration, I analyzed two games scheduled for 8/12/2023: a soccer match between *Barcelona* and *Getafe*, and a baseball game featuring the *Red Sox* against the *Tigers*. The bookmaker *Playdoit* quoted the odds for the underdogs, *Getafe* and the *Tigers*, as 550 and 165, respectively. Converting these American odds, I derived probabilities of 15.38% and 37.73%. The combined probability of both teams winning stands at 5.8%. Converting this combined probability back to American odds yields 1,624, closely aligning with *Playdoit*'s parlay odds at 1,620. This concurrence underscores the validity of our methodology, leading me to test the previous position and ranking-based strategies. The outcomes are presented in Table 8, Figure 8, and Figure 9.

Strategy	Probability (% > $1000)	W/L
Forwards	6.60	5.0
Centers	10.20	-
Guards	49.00	6.0
Forwards-Centers	0.00	24.0
Guards-Centers	24.60	27.0
All Possible	0.00	100.0
Top 1 Player in Each Team	5.80	4.0
Top 2 Players in Each Team	6.80	4.0
Top 3 Players in Each Team	6.20	4.0

Table 8: Strategy Results for Balances above $1000 and Win/Loss Ratios

When compared with our single-game strategies, the results demonstrate a distinct increase in the likelihood of breaking even and a corresponding reduction in the win-loss ratio. For instance, single game rank-based betting strategies exhibited no probability of breaking even (Refer to Table 7) versus parlay strategies seeing > 5% chance. Of particular note, the strategy focused on betting on guards witnessed a substantial rise from 26% to 49%, with certain instances achieving nearly double the starting bankroll. Nevertheless, as previously discussed, this superior performance of

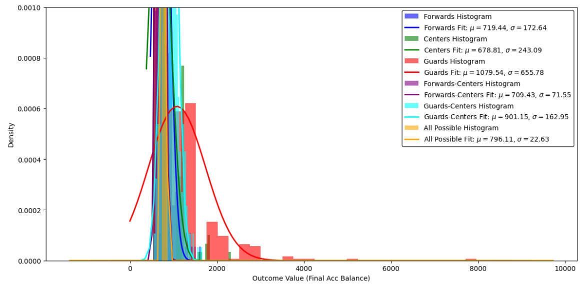

Fig. 8: Distribution Fit of 1,000 Simulations of Each Position Parlay Strategy

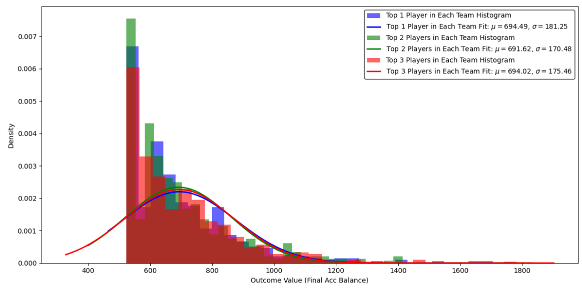

Fig. 9: Distribution Fit of 1,000 Simulations of Each Ranking Parlay Strategy

the guard strategy is attributed to the marked difference observed between our projected odds and those of the actual casino. With that said, the only significant takeaway is that employing parlay-style bets to exploit inefficiencies may prove to be profitable.

IV. Conclusion

The initial aim of this paper was to analyze the practicality of placing bets on the first player to score in a basketball game by exploring various strategies. The decision to focus on this area stems from the observation that betting on the first scorer tends to exhibit greater randomness compared to money-line wagers and, thus, might present more inefficiencies that can be leveraged against sportsbooks.

Furthermore, due to the constraint of having access to historical data from five games, there was a need to extrapolate casino odds. To tackle this, I developed a function that classified each player based on their position. For example, the exploratory analysis indicated that certain positions were more likely to score than others. In tandem with this, I also incorporated a player's ranking, gauged by how often they had been the first to score in the previous 20 games.

Building upon this, I optimized the odds for every player in each game to fine-tune our approach. Employing a linear scaling factor, I ensured the cumulative odds lay within the 110% to 120% range. This specific range was chosen to reflect casinos' typical measures to safeguard their interests, often by imposing fees. As evidenced in events like the 2022-23 Game Five finals, Casino payouts tend to diminish, pushing up the implied probabilities. However, this methodology had limitations. The NBA's broad categorization of players into three categories doesn't encapsulate the diverse capabilities of players based on specific on-court roles. For example, the 'guard' category combines point and shooting guards. Each of these subcategories has members more inclined to shoot. The generic categorization hampers our ability to detect true player probabilities properly.

From this intricate process, two main insights emerged. First, casinos charge substantial fees, and this impact becomes palpable when considering the power of a +100 adjustment. This seemingly modest modification can boost a strategy's likelihood of success from a baseline of 0% to a remarkable 13%.

Second, our reliance on publicly available data exposed some notable differences compared to actual casino odds. While our dataset broadly categorized players and ranked them primarily based on their past performances, the casino odds often deviated from these simplistic classifications. This divergence hints at the sophisticated models and the wealth of information at the casinos' disposal, setting a high bar for individual bettors dependent solely on NBA's public domain statistics.

With that said, this research was an endeavor to identify and exploit inefficiencies in bets placed on the first baskets in games. All codes related to data extraction, odds extrapolation, and strategy backtesting can be in the Appendix section and on my Github Repo. I'm open to collaborations.

V. Appendix

a. Code Functions to Extract Game Data

Objective: Extract NBA data.

```python
import re
import pandas as pd
from nba_api.stats.endpoints import commonplayerinfo, playbyplayv2, playercareerstats
from tqdm.auto import tqdm
import time
from nba_api.stats.endpoints import leaguegamelog
import os
import pickle
class GameIDFetcher:
    def __init__(self, cache_dir='nba_cache', delay=1.0):
        self.cache_dir = cache_dir
        self.delay = delay
        os.makedirs(cache_dir, exist_ok=True)  # Ensure cache directory exists

    def fetch_game_ids(self, start_season, end_season):
        game_ids_all_seasons = []
        season_types = ['Regular Season', 'Pre Season', 'Playoffs']

        for season in range(start_season, end_season+1):
            season_str = f'{season}-{str(season+1)[-2:]}'  # Convert season to NBA season format
            cache_file = os.path.join(self.cache_dir, f'game_ids_{season_str}.pkl')

            # Try to load cached data
            if os.path.exists(cache_file):
                with open(cache_file, 'rb') as f:
                    game_ids = pickle.load(f)
                    game_ids_all_seasons.extend(game_ids)
            else:
                game_ids_all_types = []
                for season_type in season_types:
                    log = leaguegamelog.LeagueGameLog(season=season_str, season_type_all_star=season_type)
                    data = log.get_data_frames()[0]
                    game_ids = data['GAME_ID'].unique()
                    game_ids_all_types.extend(game_ids)

                # Save data to cache
                with open(cache_file, 'wb') as f:
                    pickle.dump(game_ids_all_types, f)

                # Delay to prevent exceeding rate limit
                time.sleep(self.delay)

                game_ids_all_seasons.extend(game_ids_all_types)

        return game_ids_all_seasons


class NBA_FirstScore:
    def __init__(self, game_ids):
        self.game_ids = list(game_ids)
        self.height_cache = {}
        self.results = []

    def convert_height_to_decimal(self, height_str):
        if not height_str or '-' not in height_str:
            return None
        try:
            feet, inches = map(int, height_str.split('-'))
            height_decimal = feet + (inches / 12)
            return height_decimal
        except ValueError:
            return None

    def get_player_height(self, player_id):
        if player_id in self.height_cache:
            return self.height_cache[player_id]
        player_info = commonplayerinfo.CommonPlayerInfo(player_id=player_id)
        player_info_df = player_info.get_data_frames()[0]
        height = player_info_df.iloc[0]['HEIGHT']
        height_decimal = self.convert_height_to_decimal(height)
        self.height_cache[player_id] = height_decimal
        return height_decimal


    def get_first_scorer(self,df, team_who_won_ball):
```

```
        scoring_events = df[df['HOMEDESCRIPTION'].notnull() | df['VISITORDESCRIPTION'].notnull()]
        for _, row in scoring_events.iterrows():
            description_home = row['HOMEDESCRIPTION']
            description_visitor = row['VISITORDESCRIPTION']
            if isinstance(description_home, str):
                match_home = re.search(r"(\w+.*?) (\(\d PTS\))", description_home)
                if match_home:
                    times = row['PCTIMESTRING']
                    player_name = row['PLAYER1_NAME']
                    team = row['PLAYER1_TEAM_ABBREVIATION']
                    return player_name.replace(' ', '_'), team,times

            if isinstance(description_visitor, str):
                match_visitor = re.search(r"(\w+.*?) (\(\d PTS\))", description_visitor)
                if match_visitor:
                    times = row['PCTIMESTRING']
                    player_name = row['PLAYER1_NAME']
                    team = row['PLAYER1_TEAM_ABBREVIATION']
                    return player_name.replace(' ', '_'), team,times

        return None, None

    def get_results(self, game_id):
        from nba_api.stats.endpoints import playbyplayv2
        playbyplay_df = playbyplayv2.PlayByPlayV2(game_id=game_id).get_data_frames()[0]
        from nba_api.stats.endpoints import boxscoresummaryv2
        boxscore_summary = boxscoresummaryv2.BoxScoreSummaryV2(game_id=game_id)
        game_date = pd.Timestamp(boxscore_summary.get_data_frames()[0]['GAME_DATE_EST'].values[0]).strftime('%Y-%m-%d')
        first_quarter_playbyplay = playbyplay_df[playbyplay_df['PERIOD'] == 1]
        jump_balls = playbyplay_df[playbyplay_df['EVENTMSGTYPE'] == 10]
        if playbyplay_df.empty:
            print('Invalid game_id or no data available for this game!')
            return (None, None), (None, None)
        first_jump_ball = jump_balls.iloc[0][['PLAYER1_ID', 'PLAYER3_TEAM_ABBREVIATION', 'PLAYER2_ID', 'PLAYER1_TEAM_ABBREVIATION', 'PLAYER2_TEAM_ABBREVIATION']]


        #Extracting starting lineup for that game
        from nba_api.stats.endpoints import boxscoretraditionalv2
        boxscore = boxscoretraditionalv2.BoxScoreTraditionalV2(game_id=game_id).get_data_frames()[0]
        # Assume df is your DataFrame
        df = boxscore
        # Filter rows where 'START_POSITION' is not null
        starting_lineup_df = df[df['START_POSITION'].notna()]

        # Get the list of starting players and their positions for each team
        teams = starting_lineup_df['TEAM_ABBREVIATION'].unique()

        home_team = teams[1]
        away_team = teams[0]

        home_team_df = starting_lineup_df[starting_lineup_df['TEAM_ABBREVIATION'] == home_team]
        away_team_df = starting_lineup_df[starting_lineup_df['TEAM_ABBREVIATION'] == away_team]

        home_players = home_team_df['PLAYER_NAME'].tolist()[0:5]
        home_players = [name.replace(' ', '_') for name in home_players]
        home_positions = home_team_df['START_POSITION'].tolist()[0:5]
        home_positions = [name.replace(' ', '_') for name in
```

```
home_positions]
        away_players = away_team_df['PLAYER_NAME'].tolist()[0:5]
        away_players = [name.replace(' ', '_') for name in away_players]
        away_positions = away_team_df['START_POSITION'].tolist()[0:5]
        away_positions = [name.replace(' ', '_') for name in away_positions]
        if first_jump_ball['PLAYER1_TEAM_ABBREVIATION'] == home_team:
            height1 = self.get_player_height(first_jump_ball['PLAYER1_ID'])
            height2 = self.get_player_height(first_jump_ball['PLAYER2_ID'])
        else:
            height1 = self.get_player_height(first_jump_ball['PLAYER2_ID'])
            height2 = self.get_player_height(first_jump_ball['PLAYER1_ID'])
        height_diff = height1 - height2
        outcome = first_jump_ball['PLAYER3_TEAM_ABBREVIATION']

        return self.get_first_scorer(first_quarter_playbyplay, first_jump_ball['PLAYER3_TEAM_ABBREVIATION']), (game_date,height_diff,outcome),(home_team,away_team,home_players,away_players,home_positions,away_positions)

    def analyze(self):
        cols = ['Player_first_score', 'Team_first_score','Game_time', 'Game_date', 'Height_home_minus_away', 'Team_bounce_winner','Home_team','Away_team','Home_lineup','Away_lineup','Home_positions','Away_positions']
        dates = []
        failed_game_ids = []
        for game_id in tqdm(self.game_ids, desc='Computing outcomes..'):
            try:
                first, jump,other = self.get_results(game_id)
                self.results.append(list(first + jump+ other))
                dates.append(game_id)
            except:
                failed_game_ids.append(game_id)
                pass
        all_data = pd.DataFrame(self.results, columns=cols, index=dates)
        return all_data,failed_game_ids
```

b. Code Functions to Extrapolate Casino Odds

compute_scorer_percentage

Objective: Calculate the scoring percentage of players based on their previous matches.

```
def compute_scorer_percentage(row, home=True, x=20):
    """
    Calculate the scoring percentage of players based on their previous matches.

    Args:
    - row: A data row containing information about a match.
    - home (bool): If True, calculations are based on the home team. Otherwise, they are based on the away team.
    - x (int): The number of previous matches to consider when calculating the scoring percentage.

    Returns:
    - list: A list of scoring percentages for each player in the lineup.
    """
    # Import ast to evaluate string representations of complex data structures
    import ast
    # Determine the team's name based on the 'home' flag
    team_name = row['Home_team'] if home else row['Away_team']
    # Extract the last x games where the team has played, either as the home or away team
    last_games = df[((df['Home_team'] == team_name) | (df['Away_team'] == team_name))
                    & (df.index < row.name)].tail(x)
    # Ensure that the number of games extracted is at least 'x' (default 20)
    if len(last_games) < 20:
        return []
    # Determine the current match's lineup based on whether the team is playing at home or away
    team_lineup = row['Home_lineup'] if home else row['Away_lineup']
```

```python
    # Count the number of times each player has been the first to score in the last x games
    scorer_counts = last_games['Player_first_score'].value_counts()
    # Compute the scoring percentage for each player in the lineup based on their scoring history in the last x games
    scorer_percentages = [round(scorer_counts.get(player, 0) / (len(last_games) - 1), 2)
                          for player in ast.literal_eval(team_lineup)]
    return scorer_percentages
```

compute_lineup_american_odds

Objective: Calculate the American betting odds for players in a team's lineup based on their scoring likelihood.

```python
def compute_lineup_american_odds(row, home=True):
    """
    Calculate the American betting odds for players in a team's lineup based on their scoring likelihood.
    Args:
    - row: A data row containing information about a match.
    - home (bool): If True, calculations are based on the home team. Otherwise, they are based on the away team.
    Returns:
    - list: A list of American betting odds for each player in the lineup.
    """
    def custom_sort(item):
        """
        Rank players who have the same score. This method ranks duplicated ranks by player positions.
        """
        value, index = item
        # Ranking in order of likelihood to score: F1, F2, C, G1, G2
        # where F is forward, C is center, and G is guard.
        priority = [1, 2, 0, 3, 4]
        return (value, priority[index])

    def clamp(val, min_val, max_val):
        """
        Set a boundary for a value based on provided minimum and maximum values.
        """
        return max(min_val, min(max_val, val))

    def rank_list(lst):
        """
        Rank players based on their scores.
        """
        indexed_lst = [(value, index) for index, value in enumerate(lst)]
        sorted_list = sorted(indexed_lst, key=custom_sort)
        ranks = [0] * len(lst)
        for rank, (_, index) in enumerate(sorted_list):
            ranks[index] = rank
        return ranks

     # Extract the lineup for the team (home or away).
    team_lineup = row['Home_lineup'] if home else row['Away_lineup']
    team_scores = row['home_percentages'] if home else row['home_percentages']

    # Rank each player based on their score.
    sorted_scores = sorted(team_scores, reverse=True)
    rank_lineup_raw = [sorted_scores.index(i) for i in team_scores]
    rank_lineup = rank_list(rank_lineup_raw)

    odds = []
    # Calculate American odds for each player based on their rank and position.
    for x in range(len(rank_lineup)):
        if x < 2:  # Forwards
            pos = 0
            odd = clamp(750 + (rank_lineup[x] * 100) + pos, 480, 900)
        elif x == 2:  # Center
            pos = -200
            odd = clamp(750 + (rank_lineup[x] * 100) + pos, 400, 600)
        else:  # Guards
            pos = 200
            odd = clamp(750 + (rank_lineup[x] * 100) + pos, 500, 1100)

        odds.append(odd)

    return odds
```

adjust_odds_optimization

Objective: Adjust player odds uniformly to ensure implied probability is within bounds.

```python
def adjust_betting_odds(row):
    """
    Adjust player odds uniformly to ensure implied probability is within bounds.

    Args:
    - row: A data row containing Home_odds and Away_odds

    Returns:
    - tuple: Two lists of adjusted odds split between home and away.
    """
    # Importing required libraries for optimization and numerical operations
    from scipy.optimize import minimize
    import numpy as np

    def objective(delta, original_odds):
        """Objective to minimize the squared difference from original odds based on uniform reduction."""
        adjusted = original_odds - delta
        return np.sum((adjusted - original_odds)**2)

    # The ida behind these constraints is to ensure that the adjusted betting odds produce
    # implied probabilities within acceptable limits (110 to 120%) and are within the allowed betting range.
    def constraint1(delta, original_odds):
        """Ensure implied probability does not exceed 120% after uniform reduction."""
        adjusted = original_odds - delta
        implied_probs = 100 / (adjusted + 100)
        return 1.2 - np.sum(implied_probs)

    def constraint2(delta, original_odds):
        """Ensure implied probability is at least 110% after uniform reduction."""
        adjusted = original_odds - delta
        implied_probs = 100 / (adjusted + 100)
        return np.sum(implied_probs) - 1.1

    def constraint3(delta, original_odds):
        """Ensure no adjusted odd is less than 400."""
        adjusted = original_odds - delta
        return 400 - np.min(adjusted)

    def constraint4(delta, original_odds):
        """Ensure no adjusted odd is more than 1500."""
        adjusted = original_odds - delta
        return np.max(adjusted) - 1500

    def round_to_tenth(num):
        """Round a number to its nearest tenth."""
        return round(num, -1)

    # Combine home and away odds for unified processing
    original_odds = np.concatenate([row['Home_odds'], row['Away_odds']])

    # Setting up constraints for the optimization process
    cons = [{'type': 'ineq', 'fun': constraint1, 'args': [original_odds]},
            {'type': 'ineq', 'fun': constraint2, 'args': [original_odds]},
            {'type': 'ineq', 'fun': constraint3, 'args': [original_odds]},
            {'type': 'ineq', 'fun': constraint4, 'args': [original_odds]}]

    # Performing optimization to find the best adjustment (delta) that meets our criteria
    result = minimize(objective, [100], args=(original_odds,), constraints=cons)

    # Adjusting the original odds by the optimized delta value to achieve the desired properties
    delta_optimized = result.x[0]
    adjusted_all = original_odds - delta_optimized
    adjusted_odds = [round_to_tenth(odd) for odd in adjusted_all]

    # Splitting the adjusted odds into home and away segments and returning
    mid = len(adjusted_odds) // 2
    return adjusted_odds[:mid], adjusted_odds[mid:]
```

Betting Backtest Strategy Functions (permutations only)

Betting_backtest_permutations

Objective: Backtest parlay betting strategy based on either player positions or their score ranking

```python

def Betting_backtest_permutations(df, position_range=None, rank_range=None,
```

```python
random_dates=2,combinations_n=2):
    """
    This function backtests of parlay betting strategy based on either player positions or their score rankings.

    Parameters:
    - df: DataFrame containing the game data.
    - position_range: Tuple indicating the range of player positions to consider.
    - rank_range: Rank of the players to consider (e.g. top 5 players by score).
    - random_dates: Number of random dates to backtest on.
    - combinations_n: Number of game combinations to test.

    Returns:
    - bank_account: Final value in the bank account after backtesting.
    - win_loss_ratio: Proportion of successful bets made.
    """

    import pandas as pd
    import random
    import ast
    import numpy as np
    #Extracting number of bets per team
    if rank_range is not None:
        n = rank_range
    else:
        n = len(list(range(position_range[0],position_range[1])))
    # This function is used for sorting players based on their likelihood to score, given their position.
    def custom_sort(item):
        """Sort function to rank players by their likelihood to score."""
        value, index = item
        # Ranking in order of more likely to score F1,F2,C,G1,G2
        # where F is forward, C center and G Guard.
        priority = [1, 2, 0, 3, 4]
        return (value, priority[index])

    # This function clamps a value between a minimum and maximum limit.
    def clamp(val, min_val, max_val):
        """Clamp function to set max and min boundaries for each player position."""
        return max(min_val, min(max_val, val))

    # This function generates a rank list based on the custom_sort criteria.
    def rank_list(lst):
        """Generate a rank list based on custom_sort criteria."""
        indexed_lst = [(value, index) for index, value in enumerate(lst)]
        sorted_list = sorted(indexed_lst, key=custom_sort)
        ranks = [0] * len(lst)
        for rank, (_, index) in enumerate(sorted_list):
            ranks[index] = rank
        return ranks

    # This function reorders a list of player names and their odds based on the given rank.
    def reorder_list(names, odds, rank):
        """Reorder names and odds based on given ranks."""
        reordered_names = [name for _, name in sorted(zip(rank, names))]
        reordered_odds = [odd for _, odd in sorted(zip(rank, odds))]
        return reordered_names, reordered_odds

    # This function extracts the starting lineups and odds for a particular game based on score ranking.
    def extract_data_based_on_rank(row, rank_range):
        """Extract starting lineups and odds based on score rank."""
        home_players, home_odds = process_team_ranking(row, 'Home', rank_range)
        away_players, away_odds = process_team_ranking(row, 'Away', rank_range)
        return home_players[:rank_range] + away_players[:rank_range], home_odds[:rank_range] + away_odds[:rank_range], ast.literal_eval(row['Home_odds'])+ast.literal_eval(row['Away_odds'])

    # This function processes and reorders players and odds based on the scoring rank within a team.
    def process_team_ranking(row, team_side, rank_range):
        """Compute player rankings and reorder players and odds."""
        players = ast.literal_eval(row[f'{team_side}_lineup'])
        odds = ast.literal_eval(row[f'{team_side}_odds'])
        scores = ast.literal_eval(row[f'{team_side.lower()}_percentages'])
        rank_lineup = rank_list(scores)
```

```
    players, odds = reorder_list(players, odds, rank_lineup)
    return players, odds

# This function extracts the starting lineups and odds for a particular game based on player positions.
def extract_data_based_on_position(row, position_range):
    """Extract starting lineups and odds based on player positions."""
    home_players = ast.literal_eval(row['Home_lineup'])[position_range[0]:position_range[1]]
    away_players = ast.literal_eval(row['Away_lineup'])[position_range[0]:position_range[1]]
    home_odds = ast.literal_eval(row['Home_odds'])[position_range[0]:position_range[1]]
    away_odds = ast.literal_eval(row['Away_odds'])[position_range[0]:position_range[1]]
    return home_players + away_players, home_odds + away_odds, ast.literal_eval(row['Home_odds'])+ast.literal_eval(row['Away_odds'])

# This function converts odds values between American and decimal formats.
def odds_conversion(value, american_odds=True):
    """Convert american probability to decimal probability and viceversa"""
    if american_odds:
        if value > 0:
            return 100 / (value + 100)
        elif value < 0:
            return -value / (value - 100)
    else:
        if 0 < value < 1:
            if value > 0.5:
                return round(- (value / (1 - value)) * 100,2)
            else:
                return round((1 - value) / value * 100,2)
        else:
            raise ValueError("Probability should be decimal")

# This function calculates combined odds given a list of individual odds.
def calculate_odds(odds_list, american_odds=True):
    total = 1
    for odd in odds_list:
        total *= odds_conversion(odd, american_odds)
    return odds_conversion(total, american_odds=False)


# The following section conducts the main backtesting, where we iterate through each date,
# select games, compute odds, place bets, and then calculate the win/loss ratio.
# Initializing the bank account with a starting balance.
bank_account = 1000
# Counter for the number of successful bets made.
successful_bets = 0
# we'll extract player data, odds, and scoring players for each selected game on a given date.
dates = random.sample(list(df['Game_date']), random_dates)
for date in dates:
    games_selected = random.sample(list(games_df[games_df['Game_date'] == date].index), combinations_n)
    scoring_players = []
    selected_players = []
    selected_odds = []
    all_odds = []
    for gameid in games_selected:
        row = df.loc[gameid]
        if position_range:
            players, odds, all_game_odds = extract_data_based_on_position(row, position_range)
        else:
            players, odds, all_game_odds = extract_data_based_on_rank(row, rank_range)
        scoring_players.append(row['Player_first_score'])
        selected_players.append(players)
        selected_odds.append(odds)
        all_odds.append(all_game_odds)

    # Next, we'll compute all possible player combinations for betting and calculate
    # the expected payout for each combination based on the odds.
    team_indexes = [list(range(0,n*2)) for x in range(0, combinations_n)]
```

```python
        combinations = list(product(*team_indexes))
        total_bet = 0.02 * bank_account
        try:
            index_scorers = tuple([g.index(x) for g, x in zip(selected_players, scoring_players)])
            combinations_odds = []
            for combination in combinations:
                odds_list = [selected_odds[i][combination[i]] for i in range(combinations_n)]
                combinations_odds.append([combination, calculate_odds(odds_list)])
            odds = [x[1] for x in combinations_odds]
            bet_size = total_bet / len(combinations)

            # Based on the actual scoring players, we'll calculate our win or loss for the given date.
            result = [item[1] for item in combinations_odds if item[0] == index_scorers] #Identify if any combination we bet has the actual scoring combination
            if len(result) > 0:
                win_loss = (bet_size * ((result[0]) / 100)) + bet_size
                successful_bets += 1
            else:
                win_loss = 0
            bank_account += win_loss - total_bet
            if bank_account <= 0:
                break
        except:
            bank_account += 0 - total_bet
            if bank_account <= 0:
                break

    win_loss_ratio = successful_bets / len(dates)
    # The function returns the final bank account balance and the win/loss ratio after the backtest.
    return bank_account, win_loss_ratio
```

Bibliography

T. J. Moskowitz. Asset pricing and sports betting. *Journal of Finance*, 76(6):3153–3209, 2021. URL https://doi.org/10.1111/jofi.13082.