Blackjack Dice Final Report

Andrew Siu

April, 30, 2018

1 Abstract

It is impossible to tell whether a coin or dice is fair with a limited amount of flips (or rolls) in a sequence. According to the Central Limit Theorem, this only becomes possible once one reaches an ∞ amount of trials. This along with the fact that it is not possible to create a truly fair coin or dice (since to do so would require having a perfectly distributed atomic weight for each side of the object) is the reason why it is impossible to ever create a fair coin or dice. The exploration of this claim included the generation of sequences of flips and rolls for both a computationally fair/unfair coin and dice, the generation of histograms for these sequences, and a linear regression to test for approximate fairness. This was all accomplished using the libraries Numpy, MatPlotLib, SciPy, and Scikit-learn in Python 3.6.

Computer randomization was also explored and was found out that computers are pseudo-random machines. Computers, being Deterministic Finite Automaton (DFA) with billions of states, cannot be inherently random because of how DFA's work. Programming languages like Python have *pseudo-randomization* in which mathematical "tricks" are used to create random-like results [1].

These pseudo-random numbers can be used for most practical purposes as they are extremely close to simulating perfect randomness, and as a result, computers can be assumed to be random for most purposes. A game was created called Blackjack Dice and three strategies were tested to see if it were possible to get the house-edge to be negative. The three strategies are as follows:

- Strategy 1: Continue to *hit* if the score is ≤ 11
- Strategy 2: Continue to *hit* if the score is ≤ 12
- Strategy 3: Continue to *hit* if the score is ≤ 13

Based on a thousand simulations of a thousand games per strategy, the proportion of player wins in each simulation was recorded. The estimated mean of the proportions and the 95% confidence intervals for each of them strategies are:

- Strategy 1: 0.476024 = 47.6024% — (0.4745,0.4765)
- Strategy 2: 0.491394 = 49.1394% — (0.4906,0.4926)

- Strategy 3: 0.482299 = 48.2299% — (0.4796,0.4816)

It was observed that it seems like none of the strategies will result in a player-edge greater than the house-edge. Thus, this game seems to be a casino game that would benefit the house in the long-run.

2 Introduction

Are coins fair? Are dice fair? Can one really say with 100% confidence that a coin or dice is fair? Is it possible to have a truly random generator on a computer? These are questions that were investigated by extending the Casino Lab. The extension includes testing to see whether it is truly impossible to guess, with one-hundred percent certainty, the fairness of a coin, whether computers are actually random, and lastly, the creation of Blackjack Dice, a modified game of Blackjack. This was computed and explored by using Python's Random, Numpy, Scipy, MatPlotLib, and Scikit-learn libraries as well as theoretical statistics, the MIT School of Engineering's public forums, and Wizard of Odds.

3 Methods

Coin Flip Simulation

A simple coin flip simulator was created to look into probabilities of fair and unfair coins. This simulator takes in the probability of landing on a head and also the amount of flips that the user chooses as the parameters. For the fair coin, the probability of landing on a head was set to 0.5, and for the unfair coin, the probability of landing on a head was set to 0.35. The following definitions were used to create sets of coin flips with fair and unfair coins. Histograms were generated to see results of these simulations.

```
def coinflip(p, trials):
    coin = [0,1]
    seq = np.random.binomial(1, p, trials)
    flips = []
    for i in seq:
        if i == 0:
            flips.append('T')
        elif i == 1:
            flips.append('H')
```

```
    return flips
def nHeads(flips):
    n = 0
    for i in flips:
        if i == 'H':
            n += 1
    return n

def nTails(flips):
        return(len(flips)-nHeads(flips))

def set_seq():
        fair_seq = []
    biased_seq = []
    rand = float(random.randint(1,4)/10)
    count = 0

    while count < 10:
        x = coinflip(.5, 10)
        y = coinflip((rand),10)
        count += 1
        fair_seq.append(x)
        biased_seq.append(y)
    return fair_seq, biased_seq

def heads_seq(lst):
    n_heads = []
    n_tails = []
    for i in lst:
        n_heads.append(nHeads(i))
        n_tails.append(nTails(i))
    return n_heads, n_tails
```

Dice Roll Simulation

A function was created to simulate the rolling of two six-sided dice and also report the probability of each outcome, assuming both dice are fair. The theoretical probabilities of each possible outcome are shown in Figure 1.

```
def dice_prob(sides,result):
    total_outcomes = (2*sides) - 1
    prob = 0
    midpoint = 0
    outcomes = {}
    dice = []
    count = 0
    outcome_counter = 2
    for j in range(sides):
        dice.append(j+1)
```

```
    midpoint = dice[sides-1] + 1

    for x in range(total_outcomes):
        outcomes[outcome_counter] = 0
        outcome_counter += 1
    outcome_counter = 0
    for x in outcomes:
        if(x <= midpoint):
            outcomes[x] = outcome_counter + 1
            outcome_counter += 1
        else:
            outcomes[x] = outcome_counter - 1
            outcome_counter -= 1

    prob = outcomes[result]/(sides**2)
    return prob
```

The dice_prob function takes in the amount of sides for each dice and the desired outcome for the sum of the two dice. It uses these inputs to compute the probability of the specified outcome and reports it as a decimal value.

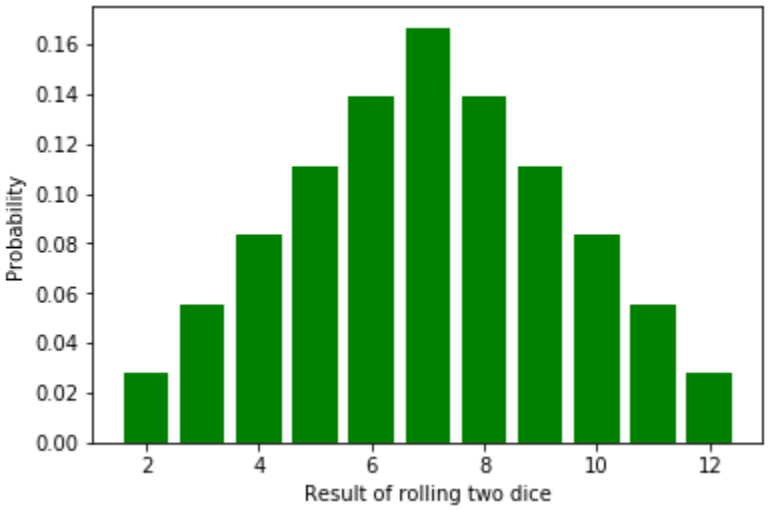


Figure 1: Theoretical Probabilities for Two Six-Sided Dice

Blackjack Dice Simulation

Blackjack dice is a simple dice game similar to blackjack, played between one to three players and the house. The objective of the game is to beat the house by either getting a cumulative score of

18, or by letting the house roll dice until their score exceeds 18. Each player starts with a wallet with the player's money included (in the source code a wallet of $100 was used for simplicity), and must bet a minimum of $10 per game. In each game, players will roll two dice, and are allowed to roll the two dice again to improve their score as much as desired (called a *hit*). If the player scores more than 18 in a game, they will automatically *bust* and lose their bet and the house will not roll against that player. Conversely, if the player scores exactly 18, they will automatically win and get back 1.5x their original bet and the house will not roll against that player. If players decide to stop rolling before they exceed 18, the house will play against each of those players individually. The house will continue rolling until it either *busts* by scoring over 18, or wins by scoring higher than the player without exceeding 18.

Three similar strategies for playing this game were developed, and tested for probability of winning, resulting in a estimated mean $\hat{\mu}$ and 95% confidence interval. The house and player edge were determined for each of the tested strategies once the $\hat{\mu}$'s were calculated. To simplify the simulation, wallets and bets were disregarded, while a constant of one player was used in the simulation of the game. The strategies are as follows:

- Strategy 1: Continue to *hit* if the score is ≤ 11
- Strategy 2: Continue to *hit* if the score is ≤ 12
- Strategy 3: Continue to *hit* if the score is ≤ 13

It was hypothesized that Strategy 1 would provide the highest $\hat{\mu}$, given that the player has the highest probability of rolling the score of 7, creating the best chance of reaching the winning score of 18 and the highest probability of not exceeding a score of 18 in a roll of two dice.

4 Results

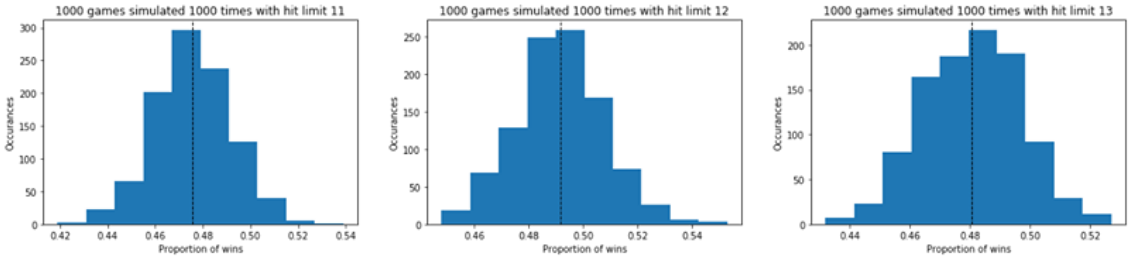
Coin Flip Simulation

The results of the coin flip simulation showed that the true probability of the coin became clearer as the number of trials increased. Based on the observations, the hypothesis that it's impossible to create a truly random coin remains unchanged.

Blackjack Dice Simulation

After simulating the three strategies proposed, it has been concluded that Strategy 2 gave the highest

win percentage when simulated for one-thousand trials of one-thousand games.



(a) Strategy 1 (b) Strategy 2 (c) Strategy 3

Figure 2: Winning Proportions For Each Strategy

- Strategy 1: $0.476024 = 47.6024\%$ — (0.4745,0.4765)
- Strategy 2: $0.491394 = 49.1394\%$ — (0.4906,0.4926)
- Strategy 3: $0.482299 = 48.2299\%$ — (0.4796,0.4816)

As the $\hat{\mu}$'s for strategies 1 and 2 are inside the confidence intervals, it can be said with 95% confidence that those are sufficient estimations for the true μ of the strategies. The opposite goes for the third strategy since its μ is outside the confidence interval, but the true mean seems to be close to 48% which is still less than the estimated mean for Strategy 2.

The house-edge for each of the strategies was calculated like so:

Assuming a \$10 bet for a game

- Strategy 1: $0.476024(5) + (1 - 0.476024)(-10) = \frac{-2.85964}{10}$ [\$10 bet] $= -0.285964$
- Strategy 2: $0.491394(5) + (1 - 0.491394)(-10) = \frac{-2.62909}{10}$ [\$10 bet] $= -0.262909$
- Strategy 3: $0.480000(5) + (1 - 0.480000)(-10) = \frac{-2.80000}{10}$ [\$10 bet] $= -0.280000$

This results in a lowest house-edge for Strategy 2 equaling a loss of $\approx$ 26¢ per game on average [2].

Computer Randomization

The question of whether computers can be random has haunted computer scientists for decades. It was relatively recently that computers were able to simulate randomized outputs such as a random number generator. There are "true" random generators that computers can use as well as "pseudo"-random generators that are close to random.

The reason as to why computers cannot be inherently random machines is because they are all deterministic. Determinism is defined as future events being predetermined by past events; if a machine is deterministic then it will, on a fixed input, output the same output every time. It can be easier to think of a computer as a Deterministic Finite Automaton which is a graph of what a computer does in a certain "state" and to which state it transitions to on a binary input. When one asks a computer do give it a random number, the machine is in a certain state and will read the same input (asking for a random number) and will respond by giving back the same number every time, unless a seed is unspecified. When no seed is specified, the computer will do some "math tricks" or a special algorithm that will take the input state and will pick a seed that is hidden in the algorithm's source code that will simulate a random output. If an user had access to what state you were in when the command asking for the random number, then that person can say what number will be in the output, hence why it's pseudo-random.

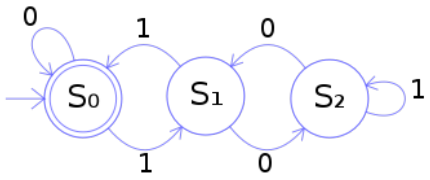


Figure 3: DFA

Computers can simulate "real" randomness with help from naturally occurring atmospheric events or an atomic clock, which most computers do not have. An example of these naturally occurring atmospheric events are a computer's fan temperature at a certain time and a mouse click that was captured at a certain time before asking for a random number. Since an outside source is random, the number generated via this algorithm will be "truly" random/irreproducible [1, 3].

5 Summary

Based on the results, one can not truly have 100% confidence about the fairness of a coin or dice, as it is impossible to reach an infinite amount of trials. One can be truly certain about the fairness of a coin or dice only as the number of trials reaches infinity, revealing the true probability of each outcome. The coin flip simulator demonstrated this idea, as it could not provide a "fair"/even amount of heads and tails outcome when the probability for each outcome was set to .5.

In addition, three strategies were developed and simulated in order to acquire the best strategy with the highest profit for Blackjack Dice in the long-run. It was concluded that Strategy 2 is the most optimal out of the three because it had the lowest house-edge after one million simulations. This is contrary to the initial hypothesis that Strategy 1 was the most optimal. This could be due to Python's pseudo-random Random package.

References

[1] Rubin, Jason M. Can a Computer Generate a Truly Random Number? *MIT Engineering*, 1 Nov. 2001, engineering.mit.edu/engage/ask-an-engineer/can-a-computer-generate-a-truly-random-number/.

[2] Shackleford, Michael. How the House Edge for Each Bet Is Derived. *Blackjack Strategy*, 15 Sept. 2016, wizardofodds.com/games/craps/appendix/1/.

[3] Anonymous. "Deterministic Finite Automaton." *Wikipedia*,Wikimedia Foundation, 17 Apr. 2018, en.wikipedia.org/wiki/Deterministic_finite_automaton.