ROULETTE BETTING ANALYSIS

ILINA SHAH*, ANNA BARRY†, AND SURBHI BECTOR‡

Abstract. Should a player bet a certain number of times in order to maximize profits in the game of roulette? This paper explores this question through iterative algorithms involving Monte Carlo simulations. Keeping the amount a player bets and the number of times they bet constant, we run 5000 simulations to see if, for a particular betting option, the play at which a player's money is at its peak converges to a specific trial number. We find that that the peak play consistently converges to a range within 3 numbers. Additionally, the peak play is higher for betting options with lower probabilities of winning than ones with higher probabilities of winning.

Key words. roulette, Monte Carlo simulation, gambling strategy, probability

1. Introduction.

As an extension of the casino lab, we decided to explore various betting strategies in the game of roulette. The roulette wheel has red and black slots ranging from 1-36, with two additional slots, 00 and 0, that solely add to the house advantage. Each number has properties that players can bet on, and the properties correspond with the different probabilities of the roulette ball landing on said property. These probabilities represent the number of slots which meet the given property out of the total 38 slots, which will be referred to as w for the probability of a player winning a round. Another set of probabilities represent the number of slots which meet the given property divided by 36 so that the 0s are ignored, which will be referred to as r. If a player succeeds in landing on the option they bet on, the house returns $1/r$ multiplied with the amount of money they bet $0 < w < r < 1$ [2].

Research on roulette mechanics and strategies shows that casinos "know how to work probabilities"[3]. This inspired interest in finding out if players can similarly work probabilities. We hypothesized that there may be a peak number of plays at which a player obtains the maximum profit on average for each betting option, referred to as the "peak play." This would also indicate the point at which the casino

*Department of Natural Sciences, The University of Massachusetts Amherst, Amherst, Ma 01003 (ishah@umass.edu),
†Department of Natural Sciences, The University of Massachusetts Amherst, Amherst, Ma 01003 (annabarry@umass.edu),
‡Department of Natural Sciences, The University of Massachusetts Amherst, Amherst, Ma 01003 (sbector@umass.edu)

would want to entice the player to keep playing in order for the house to gain profit. Additionally, we hypothesized that if such a peak play number exists, it would be higher for the single number betting option because it has the lowest probability, and is thus less likely to be landed upon earlier. Monte Carlo simulations were used to generate data on the multiple betting options to explore these hypotheses, keeping the starting player money, amount of money being bet, and the number of plays constant for each simulation.

2. Methods. While the project utilizes many different methods, the main technique uses algorithms that performed iterations as well as Monte Carlo simulations. Alongside this main technique, data structures are used to store the simulated data, visualization to illustrate the data, and theorems, such as the law of large numbers, to explain why these algorithms worked the way they do.

The play function acts as the foundation for this research. This function simulates a game of roulette in which the player starts with $5,000, and allows the player to decide how much money to bet, which betting option they want to play, and how many times to play using those parameters. The betting options are chosen by the player from a dictionary where the keys are the betting options and the values are the corresponding number of slots on the wheel for said betting option. In order to simulate a roulette spin, an integer is randomly generated from -1 to 36, where -1 and 0 represent the 00 and 0. If the number generated is both greater than 0 and less than or equal to the betting option value, the algorithm counts the play as a success. It then adds the earnings, $1/r$ multiplied by the betting amount, to the player's pool of money in addition to subtracting the betting amount. If the trial is not a success, the function only subtracts the betting amount. The function produces the returning the amount of money the player is left with after all the plays and the peak play in which the player had the most money. Another version of this function is amended to return a list of the player's money at each play. For all analyses, the betting amount is kept constant at $5, as the amount of money bet does not affect the patterns themselves as long as it is kept constant. The number of plays is also kept constant at 100 because it is low enough to be reasonable for a real player to play and high enough so that

patterns can be observed.

For the first part of the analysis, we looked at what amount of money the player is left with on average for each play. 5000 simulations of the game are tested using the play function, in which the player bet $5 100 times in a row. A list is then produced containing 100 numbers that represent the player's money at each play averaged over all the simulations. This was done for all of the different betting options.

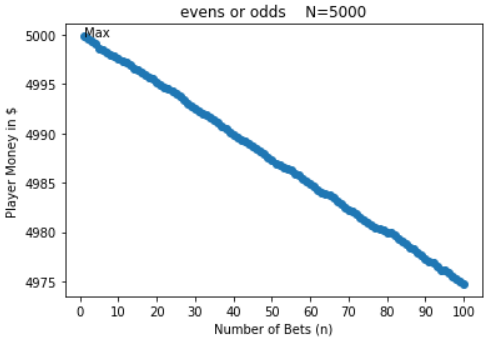
The second part of the analysis explores the main hypotheses through the law of large numbers. The law of large numbers is a theorem which states, as the number of identically distributed and randomly generated variables increases, their sample mean or probability will approach their theoretical mean or probability [4]. The observed model is based off of a typical coin toss simulation, in which 1 represents heads and 0 represents tails. In the coin toss example, the x axis is the number of simulations and the y axis is the average of the current peak trial simulation and the previous peak trial simulations. This method demonstrates the law of large numbers because over several simulations, the line plotted converges to .5, which is the theoretical probability of obtaining heads or tails. Similarly, in order to explore whether there is a peak play, the play function is simulated 5000 times to focus on which trial number is considered the peak play for each simulation. This data is visualized in a graph to emulate the coin toss example, where the x axis is the number of simulations and the y axis is the average of the simulated peak play and all the peak plays from the previous simulations. These results determine whether the peak play converges to a particular trial number for each betting option.

Finally, if the simulated plots did show peak play values converging to a specific number, we want to see how that converged number differs when betting on options with very low probabilities, and thus higher winnings during a successful play, or high probabilities, and thus lower winnings during a successful play. A bar plot was created to show which peak play each betting option converges to using the last plot point of the respective convergence plots. The betting options are ordered from least probability of success to the greatest probability of success on this bar plot.

3. Results.

3.1. Player Money over Simulations. This plot shows the player money at each play averaged over 5000 simulations for the betting options of "evens" or "odds" which have a 18/38 probability of being landed upon . The results show that the average amount of money a player has decreases as they play more. This same pattern was observed for every single betting option.

FIG. 1.



3.2. Peak Trial. These plots show that the average peak play which different betting options converge to over 5000 simulations. Running the same simulation function within a particular betting option showed that the peak play converged to a number that was consistently within the range of a few numbers. Essentially, the peak play consistently converges to around a certain number.

Even, Odd, Red, Black, First Half and Last Half all have a 18/38 probability of coming up on the wheel.

FIG. 2.

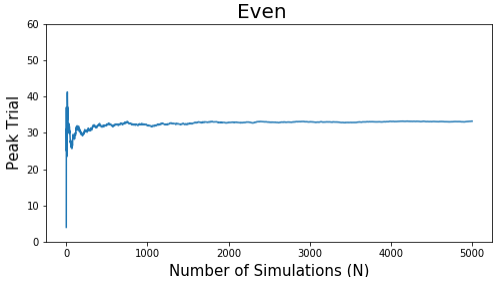


FIG. 3.

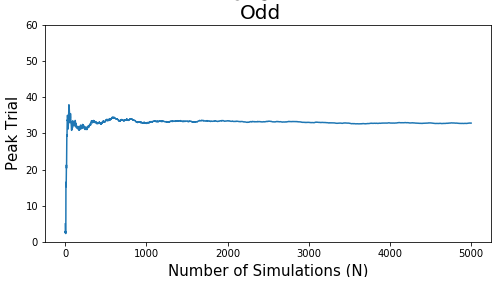


FIG. 4.

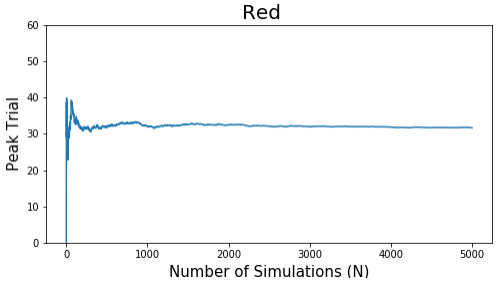


FIG. 5.

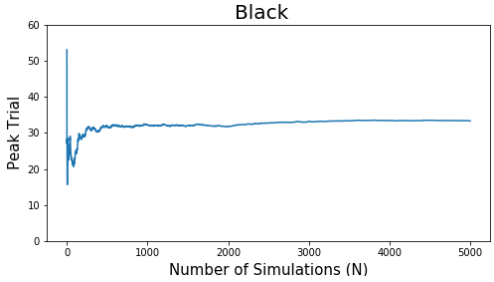


FIG. 6.

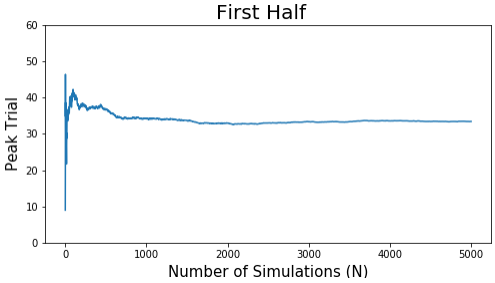
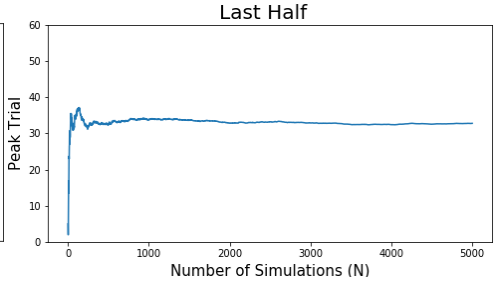


FIG. 7.



First Thirds, Second Thirds, Last Thirds and 2 to 1 all have a 12/38 probability of coming up on the roulette wheel.

FIG. 8.

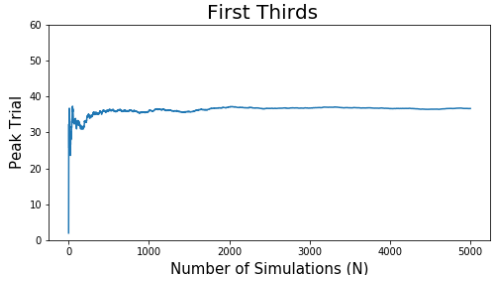


FIG. 9.

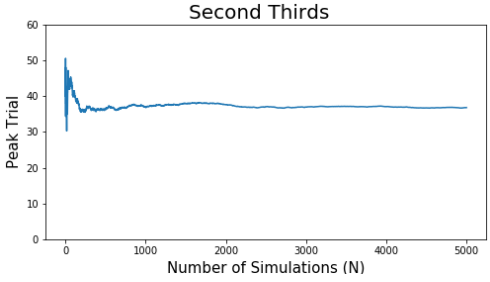


FIG. 10.

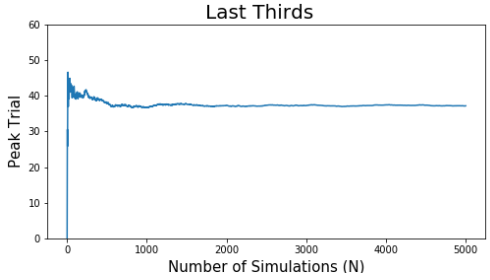
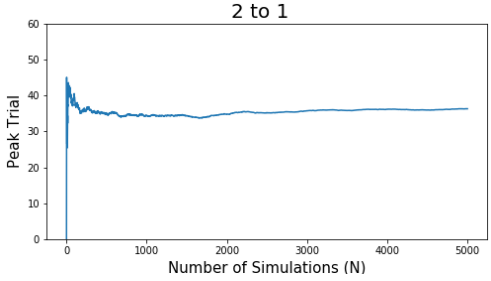
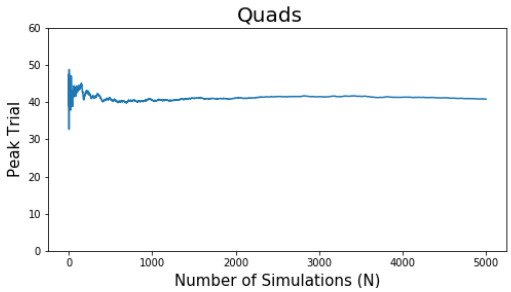


FIG. 11.



Quad is the only betting option with a 4/38 probability of showing up on the roulette wheel.

FIG. 12.



Pairs and Greens (0 and 00) are the options with a 2/38 probability.

FIG. 13.

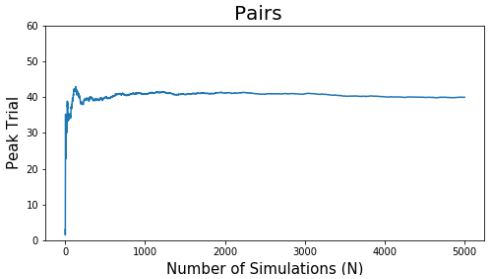
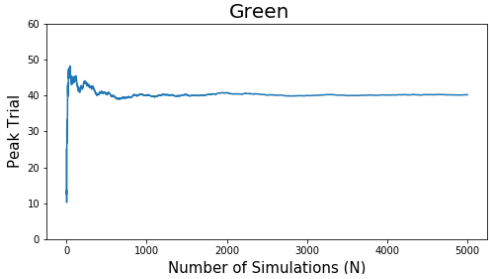


FIG. 14.



The last betting option is a Single Number which has a probability of 1/38 of showing up on the roulette wheel.

FIG. 15.

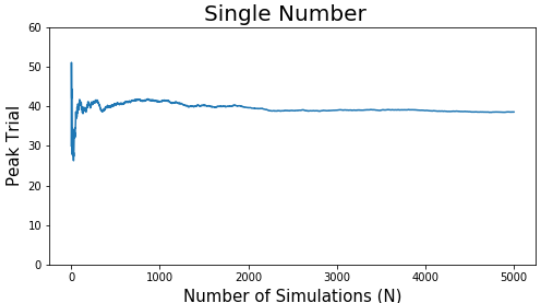
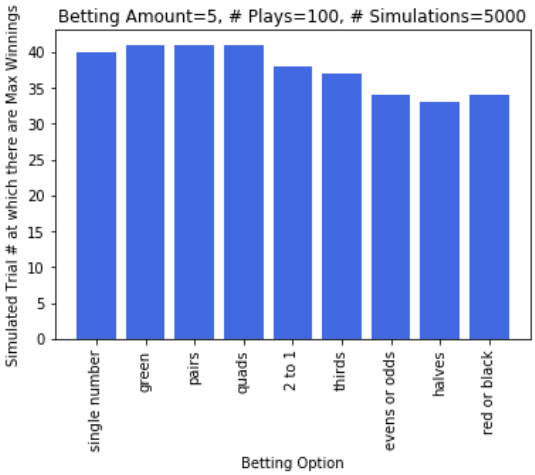


FIG. 16.



The above bar plot shows the rough number of peak play which each betting option converges to through the previous analysis. Re-running the function which produced this plot produces consistent results every time. Betting options with lower probabilities tend to have slightly higher peak plays than those with greater probabilities.

4. Summary. The results indicate that our hypotheses were correct. If a player is to bet a certain amount of money on one betting option 100 times in a row, the peak play on average does converge to a specific number. In essence, there is a specific number of plays that on average produces maximum winnings for a player betting using this strategy of repeat betting. Additionally, the betting options with lower probabilities like "single number" indeed had higher peak plays than those with higher probabilities. However, does this mean that players can "work probabilities" using these findings?

In this experiment involving Monte Carlo simulations, it is important to note the "Monte Carlo fallacy", otherwise known as the "gambler's fallacy" [1]. This is the mistaken belief that in a game of chance such as roulette, if a certain outcome has not happened in a while, it is more likely to occur in the future. For instance, if a coin is tossed a million times, roughly half of those tosses are heads. However, tossing 10 coins in a row and only receiving tails does not mean that heads is due. The graphs

produced demonstrate this idea as well. Though the line eventually converges to a single number after thousands of simulations, the first few simulations produce peak plays which vary greatly.

As to whether a player should use these findings as a betting strategy and play until the peak play number for a particular betting option - more research is needed. However, what is clear is the casino should encourage players to simply play as much as they can. The results of the first graph shows that on average, player money decreases as they continue to play. The casino may also develop a marketing strategy to keep players going once they reach the peak play numbers so that leaving aside the Monte Carlo fallacy, they are doing all they can to get as much of the player's money as possible. In conclusion, the results indicate that the hypotheses were correct, but whether they allow the players to work probabilities requires further analysis.

Appendix A. .

```
import random
import pdb
import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn.linear_model import LogisticRegression
import pandas as pd
import statistics as stats
from sklearn.model_selection import train_test_split
from sklearn import metrics

##pdb.set_trace()

# Functions Directly Used Report

## Play Function Version 1

def play(amount, choice, n):
    """
    This simulates a game of roulette in which the player starts with
↪   5,000 dollars, and allows the player
    to decide how much money to bet (amount), which betting option
↪   they want to play (choice),
    and how many times to play using those parameters (n). It returns
↪   the amount of money the
    player is left with after all the plays and the peak play in
↪   which the player had the most
    money.
    """
```

```
    options = {'even':18, 'odd':18, 'quads':4, 'pairs':2, 'first
    ↪  thirds':12, 'second thirds':12, 'last thirds':12, 'first
    ↪  half':18, 'last half':18, 'single number':1, 'green':2,
    ↪  'red':18, 'black':18, '2 to 1':12}

    value = options.get(choice)

    prob = value/36
    winnings = 1/prob
    i = 0
    money = 5000
    most_money=5000
    best_step = 0

    while i < n:
        i = i+1
        number = random.randint(-1,36) ##-1 represents 00
        if 0 < number <= value :
            money = money + (amount*winnings) - amount  ## you still
            ↪  have to pay for your bet
            if money > most_money:
                most_money = money
                best_step = i
        else:
            money = money - amount
    return money, best_step

## Play Function Version 2
```

```
def play(amount, choice, n):
    """
    This simulates a game of roulette in which the player starts with
    5,000 dollars, and allows the player
    to decide how much money to bet (amount), which betting option
    they want to play (choice),
    and how many times to play using those parameters (n). This
    returns a list of the player's
    money at each play.
    """

    options = {'single number':1, 'green': 2, 'pairs':2,  'quads':4,
    '2 to 1':12, 'thirds':12, 'evens or odds':18, 'halves':18,
    'red or black':18}

    value = options.get(choice)

    odds = value/36
    inverseodds = 1/odds
    i = 0
    money = 5000
    moneylist = []

    while i < n:
        i = i+1
        number = random.randint(-1,36) ##-1 represents 00
        if 0< number <= value :
            money = money + (amount*inverseodds) - amount  ## you
            still have to pay for your bet
        else:
```

```
            money = money - amount
        moneylist.append(money)
    return moneylist

def play_simulate(amount, choice, n, N):
    """
This function simulates N games of the play function with the
↪ parameters amount, choice, and n. It returns a list containing
↪ 100 numbers that represent the player's money at each play
↪ averaged over all the simulations.
    """
    i = 0
    simlist = []
    while i < N:
        simlist.append(play(amount, choice, n))
        i+=1
    avglist = [float(sum(col))/len(col) for col in zip(*simlist)]
    return avglist

def plot_simulate(amount, choice, n, N):
    """
    This function simply plots the list generated from
↪ play_simulated(amount, choicen, n, N).
    """

    x = range(1, (n+1))
    y = play_simulate(amount, choice, n, N)
    max_money = max(y)
    max_trial = y.index(max_money)+1
    plot = plt.plot(x, y, marker = 'o')
```

```
    plt.xticks(range(min(x)-1, max(x)+1, 10))
    title = choice + "    N=" + str(N)
    plt.title(title)
    plt.annotate('Max', xy=(max_trial, max_money))
    plt.xlabel("Number of Bets (n)")
    plt.ylabel("Player Money in $")
    return plot

def plot_optionbestplaynumber(choice, N):
    '''
This function conducts N simulations of play(5, choice, 100) and
↪  creates a list where each value is the average of the simulated
↪  peak play and all the peak plays from the previous simulations.
↪  It then plots a graph to emulate the coin toss example, where the
↪  x axis is the number of simulations and the y axis is the average
↪  of the simulated peak play and all the peak plays from the
↪  previous simulations.
    '''
    data = []
    meanlist = []
    for i in range(N):
        _, playnum = play(5, choice, 100)
        meanlist.append(playnum)
        ave = stats.mean(meanlist)
        data.append(ave)
    x = range(N)
    y = data
    plt.figure(figsize=(10,5))
    plt.plot(x,y)
    plt.xlabel("Number of Simulations (N)", fontsize = 15)
```

```
    plt.ylabel("Number Trial", fontsize = 15)
    plt.figtext(.5, .9, "Averaging of " + str(choice) + " betting
    ↪  option over N simulations", fontsize = 20, ha = "center")
    plt.show()
    return stats.mean(data)

def besttrial_final(amount, choice, n, N):
    """
    This function extracts the peak play from play(amount, choice, n)
↪   for N simulations, and creates a list where each value is the
↪   average of the simulated peak play and all the peak plays from
↪   the previous simulations. It returns this list of length N.
    """

    data = []
    meanlist = []
    for i in range(N):
        _, playnum = play(amount, choice, N)
        meanlist.append(playnum)
        ave = stat.mean(meanlist)
        data.append(ave)
    return(data)

def trialresults(amount, n, N):
    """
    This function extracts the very last value of the list produced
↪   by besttrial_final(amount, choice, n, N) where choice is inputted
↪   as the various betting options through a forloop. This produces a
↪   dictionary in which the keys are different betting options, and
↪   the values are the last peak play number that was extracted.
```

```
    """
    choices = {'single number':1, 'green': 2, 'pairs':2,  'quads':4,
    ↪  '2 to 1':12, 'thirds':12, 'evens or odds':18, 'halves':18,
    ↪  'red or black':18}
    result = {}

    for i in choices.keys():
        triallist = besttrial_final(amount, i, n, N)
        peaktrial = round(triallist[N-1])
        result[i] = peaktrial

    return result

def plottrialresults(amount, n, N):
    """
    This function creates a barplot where each category is the
↪   betting option ordered from least probability of winning to
↪   highest probability of winning, and the y axis is the number of
↪   peak play extracted using trialresults(amount, n, N).
    """
    myDictionary = trialresults(amount, n, N)
    plt.bar(myDictionary.keys(), myDictionary.values(),
    ↪  color='royalblue')
    plt.ylabel("Simulated Trial # at which there are Max Winnings")
    plt.xlabel("Betting Option")
    plt.xticks(rotation=90)
    title = "Betting Amount=" + str(amount) + ", # Plays=" + str(n) +
    ↪  ", # Simulations=" + str(N)
    plt.title(title)
    plt.savefig('triresultsps.png')
```

```
# Functions Not Directly Used Report, but Used for Other Exploratory
↪ Analysis


def best_playnumber(N):
    '''
    This returns a list of the play number in each simulation that
↪ gives the maximum amount of money.
    It takes in the number of times you wish to simulate the options.
↪ It plots the list in a box plot
    for each simulation showing a distribution of the data.
    '''
    peak_play = {}
    sim = 0
    most_money = 5000
    best_number = []
    list_choices = ['even','odd','quads','pairs','first
    ↪ thirds','second thirds','last thirds','first half','last
    ↪ half','single number','green','red','black','2 to 1']
    for i in list_choices:
        while sim in range(N):
            sim += 1
            _, best_step = play(5, i, 100)
            best_number.append(best_step)
        peak_play[i] = best_number
        best_number = []
```

```
        sim = 0
    data = peak_play.values()
    labels = peak_play.keys()
    plt.figure(figsize=(20,10))
    plt.boxplot(data) ##plotting the median of the peak plays for
    ↪  each different betting option.
    plt.xticks(range(1, len(labels) + 1), labels)
    plt.figtext(.5,.93,"Boxplots of the best trial number per betting
    ↪  option", fontsize=30, ha='center')
    plt.figtext(.5,.9,"Each game is 100 rounds and the game is
    ↪  simulated 1000 times ",fontsize=15,ha='center')
    plt.ylabel("Trial Number", fontsize = 15)
    plt.xlabel("Betting Option", fontsize = 15)
    return peak_play
    plt.savefig('boxplot_roulette.png')

def plot_bestplaynumber_med(N):
    '''
    This function only takes in one parameter N, this is the number
↪   of Monte Carlo Simulations that
    will be done on each betting option and the number of rounds in
↪   each game.
    This function goes through each betting option and give a scatter
↪   plot of the median peak play number for each
    simulation for each betting option.
    '''
    data = best_playnumber(N)
    for i in data.keys():
        numbers = data[i]
        ave = stats.median(numbers)
```

```
        data[i] = ave
    x = data.keys()
    y = data.values()
    plt.figure(figsize=(20,10))
    plt.scatter(x,y)
    plt.title("Most Frequent Best Trial Number per each option")
    plt.ylabel("Trial Number")
    plt.xlabel("Betting Option")
    plt.show()

def store_data(N):
    '''
    This takes in the number of simulations you would like to do over
    the options. It returns a list
    for each option and the number of money that the person would
    leave the casino with for each simulation.

    '''
    data = {}
    sim = 0
    list_temp = []
    list_choices = ['even','odd','quads','pairs','first
    thirds','second thirds','last thirds','first half','last
    half','single number','green','red','black','2 to 1']
    for i in list_choices:
        while sim in range(N):
            sim += 1
            money, _ = play(5, i, N)
            list_temp.append(money)
        data[i] = list_temp
```

```
        list_temp = []
        sim = 0
    return(data)


def win_or_lose(amount, n, N):
    '''
    a function that should theortically predict if the betting option
↪   will win or lose and plot
    if the betting option actually wins or loses based on the amount
↪   of simulations. Ideally this function
    will also be able to use sckit learn to predict which function
↪   would win or lose over a number of simulations.
    '''
    total = {}
    win = {}
    winmoney = {}
    total = store_data(N)
    data = total.values()
    labels = total.keys() # using the store data data and plotting it
    ↪  below(that is also commented out)
    x = labels
    plt.figure(figsize=(15,9))
    plt.boxplot(data)
    plt.xticks(range(1, len(labels) + 1), labels)
    for f in labels: ##Looking to see if overall, that game strategy
    ↪  won money or lost money (making the data)
        yesorno = 0 #Assume they lost money
        money = total[f]
        final = stats.mean(money)
        if final > 5000:
```

```
            yesorno = 1 ##if they won money then change the value to 1
        winmoney[f] = yesorno


    for i in labels: #looking to see if each game was a win or lose
    ↪  (makingn the data)
        winlist = []
        for end in total[i]:
            if end >= 5000:
                winlist.append(1)
            else:
                winlist.append(0)
        winlist = stats.mean(winlist)
        win[i] = winlist
    windata = win.values()
    options = {'Names' : ['even', 'odd', 'quads', 'pairs', '1st
    ↪  thirds', '2nd thirds', '3rd thirds', 'first half', 'last
    ↪  half',  'single', '000', 'red', 'black', '2 to 1'],
    ↪  "Prob":[18/38, 18/38, 4/38, 2/38, 12/38, 12/38, 12/38, 18/38,
    ↪  18/38, 1/38, 2/38, 18/38, 18/38, 12/38]}
    df = pd.DataFrame(options) ## making a dataframe with the names
    ↪  and their porbablities.
    df['end money'] = data ##just the store data data
    df['percentwins'] = windata ## looks into each individual game
    ↪  whether you win or lose, calcs precent of win
    df['winmoney'] = winmoney.values() ## 0 or 1 depending on if you
    ↪  won money overall or not
    df = df.sort_values(by = ['Prob'], ascending = True) #sorting
    ↪  dataframe by smallest probability
    print(df)
```

```
x = df[["Prob", "percentwins"]] #trying to find the predictors
↪  for the log regression
y = df[['winmoney']]
#x = x.to_numpy(copy = True)
#y = y.to_numpy(copy = True)
X_train,X_test,y_train,y_test=train_test_split(x,y,test_size=0.3,
↪  random_state=0) #starting the log
↪  regression
##the line above this is what is giving me trouble, once this
↪  works the logistic regression should work too
logreg = LogisticRegression()
logreg.fit(X_train, np.ravel(y_train, order = 'C'))
predictions = logreg.predict(X_test)
score = logreg.score(X_test, np.ravel(y_test))

cnf_matrix = metrics.confusion_matrix(np.ravel(y_test),
↪  predictions)
print(score, predictions, cnf_matrix)
plt.figure(figsize=(15, 5))
plt.scatter(df.Names, df.winmoney)
plt.figtext(.5, .93, "Overall, does the strategy win or lose
↪  money", fontsize = 22, ha = 'center')
plt.figtext(.5, .89, "Average of the end money of every
↪  simulation, is it more than starting money?", fontsize = 10,
↪  ha= 'center')
plt.ylabel('Win or lose', fontsize = 15)
plt.xlabel('Betting Option', fontsize = 15)
plt.savefig('log reg roulette.png')
plt.show()
```

REFERENCES

[1] D. L. Chen, T. J. Moskowitz, and K. Shue, *Decision-making under the gamblers fallacy: Evidence from asylum judges, loan officers, and baseball umpires*, SSRN Electronic Journal, (2014), https://doi.org/10.2139/ssrn.2538147.

[2] L. E. Dubins, *A Simpler Proof of Smith's Roulette Theorem*, vol. 39, Institute of Mathematical Statistics, 1968, http://www.jstor.org/stable/2239033.

[3] A. O'Connor, *Real world stats: How (not) to win at roulette*, O'Connor Memory Judgements Lab, (2013), http://akiraoconnor.org/2013/04/25/real-world-stats-how-not-to-win-at-roulette/.

[4] R. Routledge, *Law of large numbers*, Encyclopædia Britannica, (2016), https://www.britannica.com/science/law-of-large-numbers.