Project 1 - “Stochastic Experiments”

CECS 381 - Sec 06

Dustin Martin - 015180085

Problem 1: Function for an N-Sided die

- **Introduction**
  - Problem 1 simply involves writing a function which takes in the probabilities of each side of an unfair die and returning the results of a single roll. To ensure the correctness of said function, repeated rolls will be made and the outcomes will be plotted on a stem plot graph.

    The probabilities were given as:

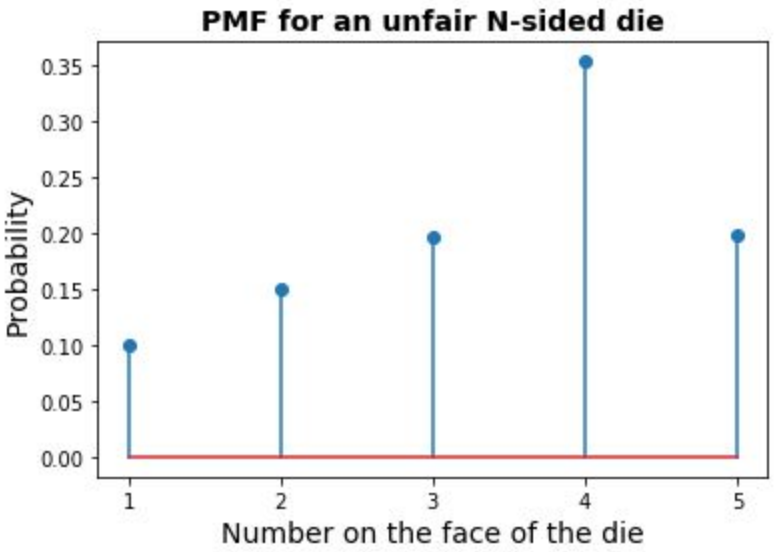
    p=[0.10, 0.15, 0.20, 0.35, 0.20]

    So, following the given probabilities of each side of the die, a stem plot which initially grows and then drops at the end is to be expected.

- **Methodology**
  - nSidedDie(p) takes in parameter `p`, the probability array which describes the probability of each side of an unfair die. The function may work with any size array as its length is determined within the function to perform the necessary calculations.

    Returning the outcome of a single roll is not enough to plot accurate information, thus the experiment is repeated 10,000 times in a for-loop, appending the results of each to an array. The results array is then passed into the plot() function which converts the calculated data into a graph.

- **Results**



  - As predicted, the stem plot initially grows to an apex of ~0.35 for side 4 before dropping down. All faces match their probabilities.

- ## Appendix/Code

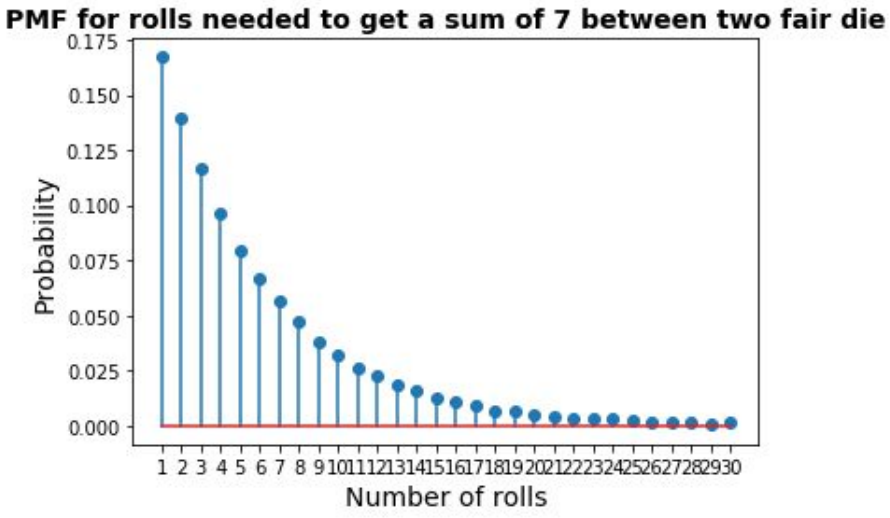
```
import numpy as np
import matplotlib.pyplot as plt

def nSidedDie(p): #flips the unfair die a single time and returns the result
    n = len(p)
    cs = np.cumsum(p)
    cp = np.append(0,cs)
    r = np.random.rand()
    for k in range(0,n):
        if r>cp[k] and r<=cp[k+1]:
            d=k+1
    return d


def plot(results, p, N): #plots the result and prints the PMF table
    b = range(1, len(p)+2)
    sb = np.size(b)
    h1, bin_edges = np.histogram(results,bins = b)
    b1 = bin_edges[0:sb-1]
    plt.close('all')
    prob = h1/N
    plt.stem(b1, prob)
    plt.title('PMF for an unfair N-sided die', fontsize = 14, fontweight = 'bold')
    plt.xlabel('Number on the face of the die',fontsize=14)
    plt.ylabel('Probability',fontsize=14,)
    plt.xticks(b1)


def main():
    N = 10000 #number of times to repeat the experiment
    p = np.array ([0.10,  0.15,  0.20,  0.35, 0.20]) #probability for each side of the unfair
die
    results = []
    for num in range(N): #runs experiment N amount of times
        results.append(nSidedDie(p))
    plot(results, p, N) #plots the result

main()
```

Problem 2: Rolls needed to get a “7” with two dice

- **Introduction**
  - This program runs simulations to test how many rolls of two die it takes until a sum of 7 is reached. To ensure accuracy, the test is repeated 100,000 times and the results are tracked so the results may be plotted.

- **Methodology**
  - The methodology for this experiment is relatively simple. Two die are flipped repeatedly until their sum is 7 and the number of flips needed is recorded. The experiment is repeated 100,000 times to ensure that the accuracy of the calculated probability for each number of rolls is sufficient.

- **Results**
  - After 100,000 repeats, the first roll is clearly the most probable event to roll a 7 at ~0.17. The probability then drops and approaches 0 as it goes onto infinity. The graph is limited to 30 rolls for viewing convenience.



- **Appendix/Code**

```
import numpy as np
import matplotlib.pyplot as plt

def plot(results, p, N): #function to plot the results
    b = range(1, len(p)+2)
    sb = np.size(b)
    h1, bin_edges = np.histogram(results,bins = b)
    b1 = bin_edges[0:sb-1]
    plt.close('all')
    prob = h1/N
    plt.stem(b1, prob)
    plt.title('PMF for rolls needed to get a sum of 7 between two fair die', fontsize
= 14, fontweight = 'bold')
    plt.xlabel('Number of rolls',fontsize=14)
    plt.ylabel('Probability',fontsize=14,)
    plt.xticks(b1)


def main():
    rolls = 100000 #number of times to repeat the experiment

    successes = []
    for num in range(rolls): # runs the experiment N amount of times
        sum = 0
        count = 0
        while sum !=7: #checks if the dice sum is 7
            sum = np.random.randint(1,7) + np.random.randint(1,7) #rolls the die and
checks the sum
            count+=1 #counts how many rolls
        successes.append(count) # records the number of rolls
    y=[]
    for num in range(30):
        y.append(num)

    plot(successes, y, rolls ) # plots the results


main()
```

Problem 3: Getting 50 heads when tossing 100 coins

- **Introduction**
  - This experiment calculates how likely it is to flip **exactly** 50 heads when flipping a coin 100 times.

- **Methodology**
  - A coin is flipped 100 times and the heads are counted. If the headcount is 50, it is deemed a success. If its not 50, it is a failure. This experiment is repeated 100,000 times so the probability that exactly 50 heads are rolled can be accurately calculated and plotted.

- **Results**
  - The probability of getting exactly 50 heads when flipping 100 coins is ~0. 08

```
The probability of getting exactly  50  heads is:
0.07955
```

- **Appendix/Code**

```
import numpy as np

c = 100 #number of coins flipped
succ = 50 #number of heads deemed a 'success'
N = 100000 #number of repeats
successes = 0 #initialized to 0
for i in range(N): #repeats the experiment
    headcount = 0
    for k in range(c):
        flip = np.random.randint(0,2) #0=heads, 1=tails
        if flip == 0: #if the flip is a head
            headcount+=1
    if headcount == succ: #if the number of heads meets success
        successes+=1
print("The probability of getting exactly ", succ, " heads is: ", successes/N)
```

Problem 4: The password hacking problem

- **Introduction**
  - This experiment tests the safety of a 4-letter password by putting it through a gauntlet of brute-force attempts. The attacker has a list of size (m) containing randomly generated 4-letter attempts to guess your word, some of which may be duplicate attempts. By running trials, we can calculate the probability that your password is on a list of m words.
    It would also be useful to know how big m needs to be before the probability of it having your password reaches 0.5.

- **Methodology**
  - There are a total of 26^4 (456,976) different 4-letter words able to be generated. We can assign each word a number instead of checking the entire word to save processing power. For my secret word, I chose 66.

    Experiment 4.1 checks the probability that a list of size 70,000 has your password on it. To do this, the code randomly picks numbers from 1 to 26^4 70,000 times to check if 66 is there. This experiment is repeated 1000 times to ensure accuracy. The probability that 66 is on the list is then calculated from the dataset.

    Experiment 4.2 is the same as 4.1, except the list size (70k) is multiplied by 7, giving us a list size of 490,000.

    Experiment 4.3 is similar to 4.1 and 4.2 except it continuously increases the size of m until the probability of 0.5 is reached. This is done by making a while-loop which continuously makes (k) bigger and bigger until the returned probability is greater than or equal to 0.5.

- **Results**
  - The probability of the secret word being in a list of 70k (m) randomly generated words is: `0.146`

  - The probability of the secret word being in a list of 490k (k*m) randomly generated words is `0.658`. You might think that because 490k exceeds the total possible combinations of 4-letter words (456,976) that the probability must be 1.0, but remember that duplicates are allowed when generating the random list and thus the probability is brought down considerably.

  - The **approximate** number of words on a list needed in order for the probability of the secret word being on the list is 0.5 is around ~`350000` words, which neatly fits between the number of words needed to get the probabilities from 4.1 and 4.2.

- **Appendix/Code**

```
import random

#my 'word'
my_word = 66

#variables given on beachboard
m = 70000
k = 7

N = 1000 # number of trials

def run(a, b, c):
    successes = 0 #counts the number of successful guesses
    for num in range(a): #number of times to run the experiment (1000)
        for k in range(b*c): #number of 'words' in each list to test
            if my_word == random.randint(0,26**4): #checks if my 'word' matches a
random
                successes+=1
                break
    return successes/N #calculates the probability that the word is on a list of m
words

p1 = run(N, 1, m) #runs the experiment with m words on the list
print(p1)

p2 = run(N, k, m) #runs the experiment with k*m words on the list
print(p2)

p3 = 0
i = 0
while p3 <= 0.5: #runs the experiment repeatedly, adding words to m until the
probability that the word is on the list is at least 0.5
    i+=1
    current = i*m
    p3 = run(N, i, m)

print(current) #prints approximately the number of words on the list needed
```