EE 381 Project 1: Random Numbers and Stochastic Experiments

by Alexandra McBride

September 18, 2019

California State University, Long Beach

Dr. Anastasios Chassiakos

Problem 1

Introduction

This problem involves a die that has n sides, or a variable amount of sides. Along with those sides are corresponding probabilities, which are variable probabilities that can be set. Those probabilities are represented as $p_1, p_2, \ldots, p_n$. This means that our die could be either fair, where the likelihood of the die landing on a side could be the same for all side, or unfair, in which the die landing on one side may be more likely or less likely than others. Through this experimentation, we are able to set the sides and probabilities of each side for a die, and we will see how they will affect the outcome of a die and see what the probabilities are for the results.

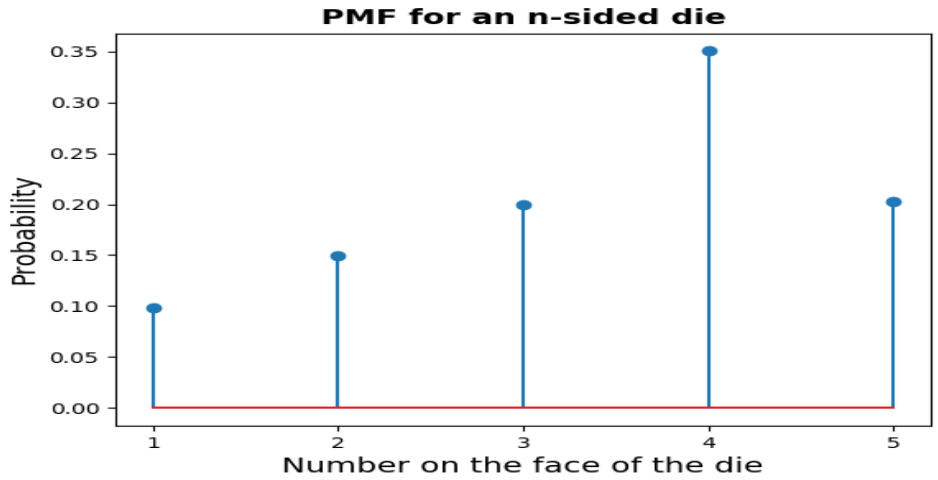
Methodology

For this problem, I will be using Python in the PyCharm IDE. The tools used will include methods from the `numpy` library and a file called `plottingP1`, which was made using the `matplotlib.pyplot` library. To execute this problem, I have created a method called `nSidedDie` that take an input p, which will be a list of probabilities of type double. The number of probabilities in the array, or the size of the array, will determine n, or the number of sides of the die. It will then roll the n-sided die and return the number of the face that it landed on. The program will call the function 100,00 times to simulate a dice roll, each time storing what face the die landed on. From this, we will gather our results for the probabilities using a stem plot as our PMF plot. This plot will be the output from this method

We will be using a test case in order to gather our result, so these results will be specific to this test case. Our test case will be the list [0.10, 0.15, 0.20, 0.35,0.20] as the argument for p.

Result and Conclusion

Here is the PMF plot, which shows the probability of each face of the n sided die:



From this plot, we see that our results for the probabilities of thee die landing on each face match the probabilities given in the test case list that we used. For this particular case, face 4 has the highest probability of 0.35, and we see that the face 4 was landed on the most in this experiment. In conclusion, the probability of each face will determine how often it is landed on in a single die roll, and we are able through this experiment to set the probabilities and see it reflect in our result.

Appendix

Here is the program used for this problem:

```
import numpy as np
import plottingP1 as plt1

#######PROBLEM 1########

# rolls the n sided die
def nSidedDie(p):

    #Getting n (number of sides)
    n=len(p)

    # random number between 0 and 1
    randNum = np.random.rand()
    return randNum




# method to perform the experiment
def exp():
    #Setting list for test case
    pList=[0.10,  0.15,  0.20,  0.35, 0.20]

    # Setting N (number of rolls of n-sided dice)
    N = 100000
```

```
    outcomes = np.zeros((N, 1))  # N zero arrays with 1 zero element

    cumulProbs = np.cumsum(pList)  # array of sums
    cumulProbs0 = np.append(0, cumulProbs)  # array with 0 added

    for i in range(0, N):
        dieRoll = nSidedDie(pList)
        for j in range(0, len(pList)):
            #checks for the range that the dieRoll is in and get the probability for
that range
            if dieRoll > cumulProbs0[j] and dieRoll<= cumulProbs0[j + 1]:
                side = j + 1
                outcomes[i] = side # stores the outcome

    plt1.plotting(len(pList),outcomes,100000) #plotting the PMF

exp()
```

Here is `plotting.py`:

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

def plotting(endpoint,outcomes,N,):
    # Setting the x values
    xRange = range(1, endpoint + 2)
    xSize = np.size(xRange)  # number of x values

    # Histogram to find final probabilities of each outcome
    hist, bin_edges = np.histogram(outcomes, bins=xRange)
    ticks = bin_edges[0:xSize - 1]
    plt.close('all')
    prob = hist / N

    # Plotting stem plot for PMF
    plt.stem(ticks, prob, use_line_collection=True)  # stem plot (x,y,...)

    # Labels for the plot
    plt.title('PMF for an 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(ticks)
    filename=input("Enter a name and extension (.pdf) to save the file as :")
    plt.savefig(filename)
```

Problem 2

Introduction

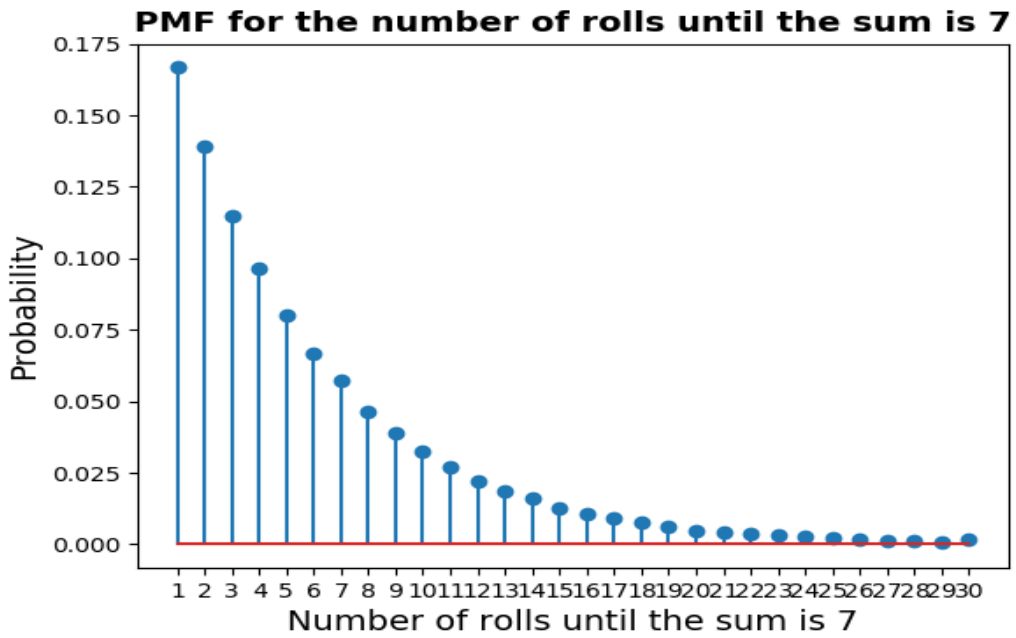
This problem involves a pair of fair 6-sided dice, and it involves getting a certain sum from the pair of dice in a certain number of rolls. For this experiment to be a success, the roll of the two dice should sum up to seven, and we are wanting to see the number of rolls that it takes until the experiment is successful. Through this experimentation, we will see the probability of success for n rolls, where the n values we will look at will range from one to thirty in our results.

Methodology

For this problem, I will be using Python in the PyCharm IDE. The tools used will include methods from the `numpy` library and a file called `plottingP1`, which was made using the `matplotlib.pyplot` library. In the program, we are going to simulate the roll pair of dice until sum is seven, keeping up with the number of rolls. Once we are successful, we will store the number of rolls it took. We will repeat these 100,000 times and store the number of rolls for each repetition. From this, we will gather our results for the probabilities using a stem plot as our PMF plot. This plot will show the probabilities for the number of rolls where the range will be from one to thirty.

Results and Conclusion

Here is the PMF plot for this problem:



From this plot, we see that the probability of the sum of the pair of rolled dice being seven on the first roll is the highest, estimating it to be about 0.17. And we see that as the number of rolls increases, the lower the probability of the sum being 7 on that roll. So, the number of rolls with the lowest probability is 30 rolls. In conclusion, the roll of two fair pair of dice summing up to seven is most likely to happen on the first roll.

Appendix

Here is the program used for this problem

```
import numpy as np
import plottingP1 as plt1
######PROBLEM 2#########

#rolls a fair die
def diceRoll():
    dice=np.random.randint(1,7) #dice 1
    return dice


#method to perform the experiment
def exp():
    #Setting N
    N=100000

    outcomes=np.zeros((N,1))

    for i in range (0,N):
        numOfRolls=0 #keep track of the number of rolls before successful experiment
        while True:
          dice1= diceRoll() #dice 1
          dice2= diceRoll() #dice 2
          sumOfDice= dice1+dice2        #sum of the two dice
          if sumOfDice==7:
             numOfRolls+=1
             outcomes[i]=numOfRolls     #assigns results into outcomes
             break
          else:
             numOfRolls+=1

#Plotting
    plt1.plotting(30,outcomes,100000)

exp()
```

Here is `plotting.py`:

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

def plotting(endpoint,outcomes,N,):
    # Setting the x values
    xRange = range(1, endpoint + 2)
    xSize = np.size(xRange)  # number of x values
```

```
    # Histogram to find final probabilities of each outcome
    hist, bin_edges = np.histogram(outcomes, bins=xRange)
    ticks = bin_edges[0:xSize - 1]
    plt.close('all')
    prob = hist / N

    # Plotting stem plot for PMF
    plt.stem(ticks, prob, use_line_collection=True)  # stem plot (x,y,...)

    # Labels for the plot
    plt.title('PMF for an 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(ticks)
    filename=input("Enter a name and extension (.pdf) to save the file as :")
    plt.savefig(filename)
```

Problem 3

Introduction

This problem involves the traditional coin toss, with two sides whose sides are heads and tails. However, in this experiment, we want to toss 100 coins at the same time. From this, we want to find the probability of exactly 50 coins landing as heads. This experiment will give us the probability of this event occurring in this type of coin toss.

Methodology

For this problem, I will be using Python in the PyCharm IDE. The tools used will include methods from the `numpy` library. In the program, it simulates 100 coins tossed at once, then sees if the sum of the number of coins that are heads equals 50. If the sum is 50, then we will add one to a counter called `numOfSuccesses`. We will repeat the toss 100,000 times, then we will take the number of successes and divide it by 100,000 to get our results.

Results and Conclusion

After running the program to simulate this coin toss, we get the following numerical result:

Probability of 50 heads in tossing 100 fair coins	
Answer :	**p = 0.07825**

From these results, we see that the percentage of exactly 50 coins landing on heads is about 0.08, which means there is a small chance of this happening when doing a roll of 100 coins. In conclusion, it is not very likely that this will happen in a single 100-coin toss if done.

Appendix

Here is the program used for this problem:

```
import numpy as np
#### PROBLEM 3 #####

#Number of tosses where number of heads is exactly 50
numOfSuccesses=0

#Setting N
N=100000

for i in range (0,N):
  coinToss= np.random.randint(0,2,100) #tosses 100 coins (0= tails, 1=heads)
```

```
  total= sum(coinToss) #sum of numbers
  if total==50:  #if total is 50, add one to numOfSuccesses
      numOfSuccesses+=1

#Calculate the probability
prob= numOfSuccesses / N
print("Probability of 50 heads:", prob)
```

Problem 4

Introduction

For this problem, the scenario is that you have created a four-letter password as part of a login which consists of lowercase letters of the alphabet only. A hacker creates a list of m random four-letter words, where m is a number value, to try and figure out if any of those words match your password. For this, we want to see what the probability is of one of the words matching your password. Next, the hacker creates a longer list of k * m random words. We again want to see the probability of one of the words matching your password. Finally, we want to see how many four-letter words should be generated for the probability of finding a match to be around 0.5. Through experimentation, we will gather the numerical results for each of these three scenarios and analyze those results in order to give a general explanation for this sort of problem.

Methodology

For this problem, I will be using Python in the PyCharm IDE. The tools used will include methods from the `numpy` library. For explanation of our program, the number of possible four-letter words that can be generated is 26^4. We can think of each possible word being assigned a number, so the first word generated will be 0, then the second word will be 1, and so forth. In this program, we utilize this by generating a random number from 0 to 26^4 for the password. We will also generate random numbers between this range to create each of the hacker lists. Each of the three scenarios mentioned will be repeated 1000 times, and from this we will gather our results for the probabilities as well as for the desired m where the probability is 0.5.

Results and Conclusion

After running the program to simulate this, we get the following numerical result:

Hacker creates m words Prob. that at least one of the words matches the password	**p=0.141**
Hacker creates k*m words Prob. that at least one of the words matches the password	**p= 0.76**
p=0.5 Approximate number of words in the list	**m= 350,000**

From these results, we see that as the number of words generated by the hacker increases, so does the probability of finding a match to the password generated. We also see that to have a probability of 0.5, the number of passwords generated should be around 350,000. In conclusion, we see that the relationship between the number of words generated and the probability of finding a match to the generated password is one that is direct.

Appendix

Here is the program used for this problem:

```
import numpy as np

#Setting the constant values
N=1000  #number of experiments
m=70000
k=9
n=26**4  #number of possible 4 letter words

#Part 1- Hacker generates m passwords
numOfSuccessP1=0
for i in range(N):
    password = np.random.randint(0,n)

    hackerList=np.random.randint(0,n,m)

    if password in hackerList:
            numOfSuccessP1+=1

print("Probability of successful attempt with m passwords:", numOfSuccessP1/N)

#Part 2- Hacker generates k*m passwords
numOfSuccessP2=0
for i in range(N):
    password = np.random.randint(0,n)

    hackerList=np.random.randint(0,n,k*m)

    if password in hackerList:
            numOfSuccessP2+=1

print("Probability of successful attempt with k*m passwords:", numOfSuccessP2/N)

#Part 3 - Find m where the probability of success is 0.5
numOfSuccessP3=0
m2=349000
for i in range(N):
    password = np.random.randint(0,n)

    hackerList=np.random.randint(0,n,m2)

    if password in hackerList:
            numOfSuccessP3+=1

print("Probability of successful attempt:", numOfSuccessP3/N)
```