Kevin Long

Professor Chassiakos

EE 381

13 January 2019

Project 1: Random Numbers and Stochastic Experiments

1) Function for an n-sided die

a) Introduction

This is a function that simulates a single roll of an "n-sided" die. The function uses the probability vector as testing: p = [0.10, 0.15, 0.20, 0.05, 0.30, 0.10, 0.10]

b) Methodology

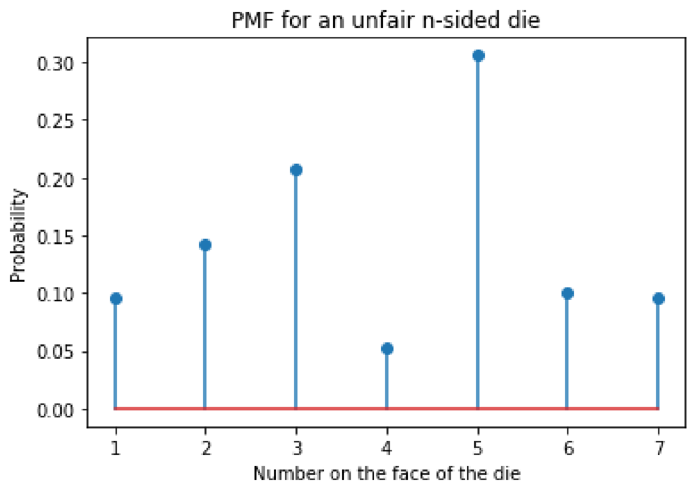
Input: The probabilities for each side, given as a vector p = [p_1, p_2, … p_n]
Output: The number on the face of the die after a single roll, i.e. one number from the set of integers {1, 2, ... n}

The function first checks to see whether the total sum of the probability array is not equal to 1.0, it prints an error message. The sum of "p" must be equal to 1.0, otherwise the probability values passed to the function are incorrect.

If the total sum of the probability array is equal to 1.0, the function will conduct 10,000 tests (this value is stored in the variable "N"). For these tests, an array is created. For every iteration in the test "N", count how many times that number is rolled and add it to the array. The number is counted by a comparison to a randomly generated number via numpy.

c) Result(s) and Conclusion(s)

Based on the vector: p = [0.10, 0.15, 0.20, 0.05, 0.30, 0.10, 0.10]
The results are:



d) **Source Code**

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

"""
1)
Function that simulates a single roll of an n-sided die.
Inputs: The probabilities for each side, given as a vector p = [p_1,p_2,...p_n]
Outputs: The number on the face of the die after a single roll,
i.e. one number from the set of integers {1,2,...n}
Note: The sum of p must be equal to 1.0, otherwise the probability values
are incorrect.
"""
def nSidedDie(p):
    n = len(p) #Number of elements in 'p'

    pSum = 0
    for k in range(n):
        pSum += p[k]

    if pSum > 1.0: #Checking for correct probability total
        print("Cumulative sum of p cannot be greater than 1.0")
    else:
        N = 10000 #Number of tests

        sampleSpace = np.zeros((N,1)) #Return a new array of given shape and
type, filled with zeros.
        cs = np.cumsum(p) #Return the cumulative sum of the elements along a
given axis.
        cp = np.append(0, cs) #Append values to the end of an array.

        for i in range(0, N): #For every iteration in N
            r = np.random.rand()
            for j in range(0, n): #count how frequent that number is rolled
                if r > cp[j] and r <= cp[j+1]:
                    d = j + 1
            sampleSpace[i] = d

        # Plotting
        b=range(1,n+2)
        sb=np.size(b)
        h1, bin_edges=np.histogram(sampleSpace, bins=b)
        b1=bin_edges[0:sb-1]
        plt.close('all')
        prob=h1/N
```

```
                    plt.stem(b1,prob)
                    # Graph labels
                    plt.title('PMF for an unfair n-sided die')
                    plt.xlabel('Number on the face of the die')
                    plt.ylabel('Probability')
                    plt.xticks(b1)
                    plt.show()

                    return d
```

2) <u>Number of rolls needed to get a “7” with two dice</u>

a) Introduction

Roll a pair of fair dice and calculate the sum of the faces. Count the number of rolls it takes until you get a sum of "7". The first time you get a "7" the experiment is considered a "success". You record the number of rolls and you stop the experiment.

The experiment is repeated 100,000 times. Each time, keep track of the number of rolls it takes to have a "success".

b) Methodology

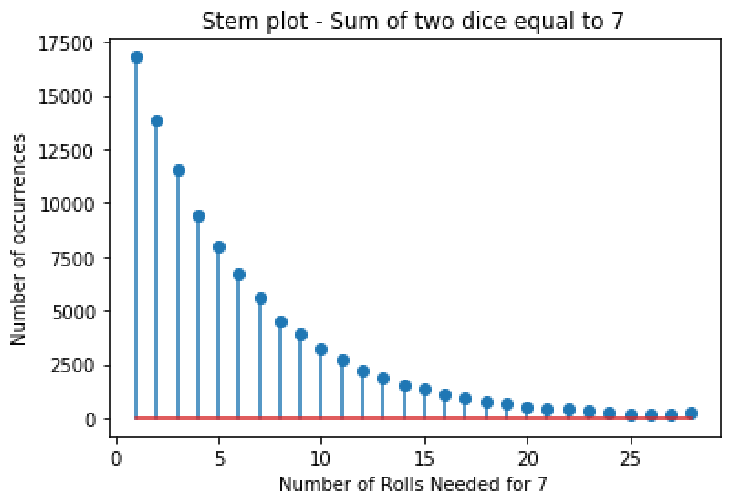
Input: None
Output: None

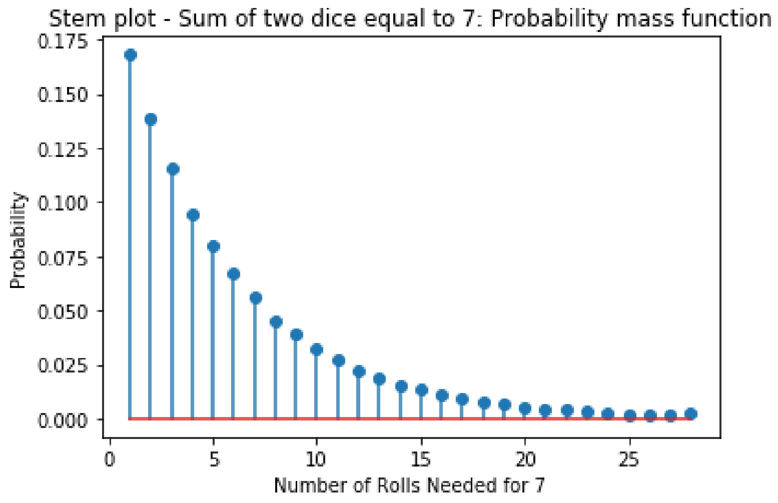
Two counters are created: timesRolled and timesGotSeven. A null array, timesGotArr, of “N” elements is also created (in this case 10,000 for the number of tests).

For each “test,” roll two dice until a 7 is rolled for that particular “test.” If a 7 is rolled, add the number of rolls it took to obtain the 7 into the array. The sentinel value, named ‘test,’ is triggered and we exit out of that “test” loop and move on into the next iteration of the entire experiment until “N” times is achieved.

c) Result(s) and Conclusion(s)

The probability of getting a certain sum from rolling two dice reduces as the sum increases.





d) **Source Code**

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

"""
2)
Number of rolls needed to get a "7" with two dice
Input: None
Output: None
"""
def numDiceRollsNeeded():
    N = 100000 #Number of tests

    timesRolled = 0
    timesGotSeven = 0
    timesGotArr = [None] * N
```

```
for i in range(0, N): #For each experiment
    test = True
    while test: #Conduct the test until a 7 is rolled
        timesRolled += 1
        d1 = np.random.randint(1,7) #Dice 1
        d2 = np.random.randint(1,7) #Dice 2
        dSum = d1 + d2
        if dSum == 7:
            timesGotSeven += 1
            timesGotArr[i] = timesRolled
            timesRolled = 0
            test = False #Exit test

b=range(1,30)
sb=np.size(b)
h1, bin_edges = np.histogram(timesGotArr,bins=b)
b1=bin_edges[0: sb-1]
plt.close('all')

fig1=plt.figure(1)
plt.stem(b1,h1)
plt.title('Stem plot - Sum of two dice equal to 7')
plt.xlabel('Sum of the two dice')
plt.ylabel('Number of occurrences')

fig2=plt.figure(2)
p1=h1/N
plt.stem(b1,p1)
plt.title('Stem plot - Sum of two dice equal to 7: Probability mass function')
plt.xlabel('Sum of the two dice')
plt.ylabel('Probability')
```

3) Getting 50 heads when tossing 100 coins

a) Introduction

Toss 100 fair coins and record the number of heads. This is considered a single experiment. If you get exactly 50 heads, this particular experiment is considered a "success.”

Repeat the experiment 100,000 times. After the “N” experiments are completed, count the total successes, and calculate the probability of getting exactly 50 heads in 100 fair coin tosses.

b) Methodology

Input: None

Output: The probability of success of getting exactly 50 heads

A counter for heads is created. The variables “n” is the number of coins used in the experiment while “N” is the number of tests conducted for the entire experiment.

A counter for the number of successful tests is created. The tests are conducted via a “for loop” and the coin tosses are simulated with the numpy function randint. If the total amount of heads equals 50 for the “test,” then it is counted and proceeds to the next test. When the final test is finished, the experiment is over and the calculated probability is returned.

c) **Result(s) and Conclusion(s)**

Probability of 50 heads in tossing 100 fair coins	
Ans.	***p* = 0.8**

d) **Source Code**

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

"""
3)
Calculates the probability of success of getting exactly 50 heads after
tossing 100 coins 100,000 times.
Input: None
Output: The probability of success of getting exactly 50 heads
"""
def fiftyHeadsPercent():
    heads = 0 #Counter for heads
    n = 100 #Number of coins
    N = 100000 #Number of tests

    numSuccessfulTests = 0
    for i in range(0, N):
        coin = np.random.randint(0,2,n) #Tossing 100 coins
        heads = sum(coin)
        if heads == 50:
            numSuccessfulTests += 1

return numSuccessfulTests/N
```

4) <u>The Password Hacking Problem</u>

a) **Introduction**

A computer system uses a 4-letter password for login. The password is restricted to lower case letters of the alphabet only.

A hacker creates a list of random 4-letter words, as candidates for matching the password. It is possible that some of the words may be duplicates.

The hacker first creates an "m" long list of passwords and then a "k*m" long list. The algorithm checks to see if there is a password in that list that matches the system's password. If there is a word in the list that matches, then the experiment is successful. Both password list experiments are ran 1000 times.

Finally, find the approximate number (m) of words that must be contained in the hacker's list so that the probability of at least one word matching the password is p = 0.5.

The list starts with m = 80,000 and k = 7.

b) **Methodology**

There are 3 functions for this particular problem.

In passwordHacking(), counters and password lists are created. The password list is filled with a helper function called pwGen() that returns a 4 character long lowercase string that will be used as a comparison to the guessed password by the user called hackPw (also generated by the function).

The function pwHacking() then takes in two paramaters, k and m, that specify how long the list will be. The function iterates N times and increments the counter 'count' if the hackPw is found in the pwList. The function returns the amount of successes over N.

The function probPwHack() does not take or return any data. It simply prints out the statistics involved with the problem. It prints out the data in order of the result(s) table as illustrated below.

c) **Result(s) and Conclusion(s)**

Hacker creates ***m*** words. Probability that at least one of the words matches the password.	***p* = 0.17**
Hacker creates ***k*m*** words. Probability that at least one of the words matches the password.	***p* = 0.7**
p = 0.5, Approximate the number of words in the list.	***m* = 320,000**

d) **Source Code**

```
import random
import string

"""
Input: None
Output: None
```

```
Function calculates the probability of guessing the correct password in a
randomly generated password list of 4 lowercase English characters.
The print statements are for the probability of 'm' words, 'k*m' words, and
number of words needed to achieve a p = 0.5
"""
def pwGen():
    return ''.join(random.choice(string.ascii_lowercase) for x in range(4))

"""
4)
Input: k, m integers
Output: probability of hacking password
Function calculates the probability of guessing the correct password in a
randomly generated password list of 4 lowercase English characters.
The print statements are for the probability of 'm' words, 'k*m' words.
"""
def passwordHacking(k, m):
    n = 26**4
    N = 1000

    #m or k*m
    count = 0
    pwList = list() #Generate a list to store passwords
    for i in range(0, k*m): #Fills list with 'k*m' 4char pw's with function
pwGen()
        pwList.append(pwGen())
    for j in range(0, N): #Check the list to see if our guessed pw is in the list
        hackPw = pwGen()
        if hackPw in pwList:
            count += 1 #Increment if found
    result = count/N
    return result

"""
Input: None
Output: None
Prints out vital information about the probability of the password
hacking experiment. Also prints out the words needed to achieve
p = 0.5 in a list of 'm' passwords.
"""
def probPwHack():
    k = 7
    m = 80000

    result1 = passwordHacking(1, m)
    print("Probability for m words:", result1)
```

```
        result2 =passwordHacking(k, m)
        print("Probability for k*m words:", result2)

        while not (result1 > 0.49 and result1 < 0.51):
            if result1 < 0.5:
                m += 10000
                result1 = passwordHacking(1, m)
            elif result1 > 0.5:
                m -= 10000
                result1 = passwordHacking(1, m)
    print("Words needed for p = 0.50:", m)
```

CODE FOR TESTING THE FUNCTIONS

```
if __name__ == "__main__":
    #1
    p = np.array([0.10, 0.15, 0.20, 0.05, 0.30, 0.10, 0.10])
    print("You rolled: " + str(nSidedDie(p)))
    #test for incorrect p value total sum
    p2 = np.array([0.10, 0.15, 0.20, 0.05, 0.30, 0.10, 0.20])
    print("You rolled: " + str(nSidedDie(p2)))
    print("-----")

    #2
    numDiceRollsNeeded()

    #3
    print("Probability of getting exactly 50 heads:", fiftyHeadsPercent())

    #4
    probPwHack()
```