CS5016: Computational Methods and Applications

Lab Report

111901030
Mayank Singla

Q1.

A straightforward approach to solve the given problem is to compute both LHS and RHS for all the values of n till 10^6 and plot them. But I observed that it is not possible to compute such a large number and the code will give an error. So, to reduce it, I take **log** on both the sides of the LHS and the RHS.

LHS reduces to `Σlog(i)`

RHS reduces to `((½) * log(2 * PI) + log(i)) + (i * (log(i) - 1))`

The **log** reduces the values of both LHS and RHS to lower values that were easily computable and hence they could be easily plotted.

I pre-computed the values of both LHS and RHS and stored them in a list and also created a **diff** list that represents the difference in the corresponding LHS and RHS values.

I am plotting two plots using the **matplotlib** library.

The first plot is for Displaying the values of both LHS and the RHS in a single graph and it was observed that both the plots overlap indicating that Stirling's approximation is valid.

In the second plot, I am plotting the **diff** list which shows the difference in the values obtained. It was observed that the values of the difference were less than 0.08 for n till 10^6. This again clearly shows that Stirling's approximation is valid.

Q2.

First of all, we need to handle exceptions in this question. So to avoid repetition of code, I made a Decorator Factory function for class methods, that simply wraps the class method in a `try-except` block and handle and prints the error as mentioned in the question. This decorator forwards all the arguments it receives to the actual method so the functionality of the function doesn't break.

Next, I created the `Dice` class in which I created some helper functions to validate the input (`numSides`, `probabilityDistribution`) obtained from the user and raise an exception in case of invalid input. I created a helper function to compute the list of CDF intervals as we are doing in the trick in the theory lectures.

Next, for the class object to be printable, I created a dunder method and return the string in the required format to print that object.

For the `roll` function, I am using the same method as discussed in theory to sample discrete distribution using the continuous random variable in-built in python using `random.random()`. I generated a random variable U in [0, 1) using this function and then find in which interval U lies and increment the count of the corresponding value of the discrete random variable in a list. This will give me the actual occurrences of the discrete values.
The expected occurrences will be *(number of throws of dice) * (prob. of each face)*
I observed that for large values of n, the expected and actual values are nearly equal in the bar graph.

Q3.
First of all, I am creating a function `isInsideCircle` which takes as an argument the center and the radius of the circle and coordinates of some other point and checks whether that point lies inside the circle or not.
The equation of circle is $(x - x_0)^2 - (y - y_0)^2 = r^2$
For a point to lie inside the circle, put the coordinates of the point in the above equation and then LHS <= RHS
Then I created a function `estimatePi` which is to estimate the value of `PI` using the Monte Carlo Method. It takes as argument **n** which tells the number of random points generated. I am generating **x** and **y** coordinates of the point independently using uniform continuous distribution and then storing the total points generated (which are also equal to total points in the square) and the number of points inside the circle in a list and then plotting those two lists.
I observed that initially for small **n**, the values keep fluctuating, but for larger **n**, the value keeps going closer to the value of `math.pi` indicating the correctness of the Monte Carlo Method.

Q4.
I am again using the same decorator factory function that I created in **Q2** to handle the error this time also.
I created a helper function that will compute the frequency distribution of the next word for some pair of words. I am storing the frequency of occurrence of the next word for a pair in an array and then finally storing the list of unique next words and their frequency distribution for all the pairs of words.
Then I created the `assimilateText` function that reads an input file. Then I split the contents of the file into words using the python's in-built function `str.split()` to extract all the different words separated by whitespace and the newline. Then I am creating a prefix dictionary that maps a pair (2-tuple) of words to a list of words that follow that pair in the text using simple loops and update logic.

At last, I am implementing the `generateText` function that generates a random text of the given number of words as input. First, I am storing all the pairs of words from the dictionary in a list using `dict.keys()`. Then, if the starting word is not given, I am choosing a starting pair of words randomly from the list of pairs of words. If the starting word is given, I am choosing a random starting pair from the list of pairs of words for which the given starting word is the first member.
Next, I am running a loop keeping track of the number of words added to the randomly generated text. If the last pair of words is present in the original text, then I am choosing the next word based on the frequency distribution of the next words created for that pair and choosing randomly from them based on their frequencies as their weights. I am doing this with the help of python's in-built `random.choices()` method. And, if the last pair of words are not present in the original text (that could only be possible for the last pair of words), then I am generating a new fresh pair of words from the list of pair of words and appending them to the text and continuing the algorithm.