Zhuo Wei Li

1530467

Teacher: George Fleischer

Probability Distributions Package

420-LCW-MS

Advanced Programming

User Manual

**********VERY IMPORTANT**********

The package requires the installation of a python graphing library called matplotlib. It can be installed by entering the 2 following lines into the command line on a windows pc:

```
python -m pip install -U pip setuptools
python -m pip install matplotlib
```

More information can be found here https://matplotlib.org/users/installing.html

****

The code is located in the Distributions.py file, all of the other files are for testing. The package has three main objectives:

1. Allow the user to calculate probabilities of discrete probability distributions (uniform, binomial, hypergeometric, geometric, negative binomial and poisson).
2. Allow the user to simulate samples from a discrete probability distribution and show a histogram of the results
3. Allow the user to calculate probabilities of continuous probability distributions (uniform, exponential).

1. Discrete Probability Calculations
Test cases can be found in the DiscreteDistributionsTest.py file

Relevant functions:

- discrete.uniform(c, d)(*x)
  - c and d define the bounds of the uniform probability distribution
  Returns the probability of the event landing happening at the x value

- discrete.binomial(n, p)(*x)
  - n is the number of trials
  - p is the probability of a success occurring in one trial
  - x is the number of successes in n trials
  Returns the probability of having x successes in n trials

- discrete.hypergeometric(N, n, k)(*x)
  - N is the total number of elements
  - n is the number of elements chosen in the sample
  - k is the number of successes in those N elements
  - x is the number of successes in those n chosen elements
  Returns the probability of having x successes in the sample

- discrete.geometric(p)(*x)
  - p is the probability of a success occurring in one trial
  - x is the trial where the first success happens
  Returns the probability of having the first success on the xth trial

- discrete.negativebinomial(k, p)(*x)
  - k is the number of successes
  - p is the probability of a success occurring in one trial
  - x is the trial where the kth success happens
  Returns the probability of the kth success happening on the xth trial

- discrete.poisson(l)(*x)
  - l (lambda) is the parameter of the poisson distribution
  - x is the number of occurrences in the interval with mean l
  Returns the probability of having x occurrences in the interval

Every relevant function to this section is organized into the discrete class in the Distribution.py file. These functions are used in two steps:

First, calling a function with specific arguments returns a probability distribution function specific to the type of distribution and arguments inputted.

Example:

b = discrete.binomial(5, 0.8) #b is a function that maps values to probabilities

Then, this probability distribution function can take one or two x values as an argument to return a probability. If one x value is inputted, it will return the probability of that specific x value to occur. If two x values are inputted, it will return the probability of that interval of x values to occur, where the first value is the start of the interval and the second value is the end of the interval.

Example:

b(4) #returns 0.4096
b(2, 4) #returns 0.6656, same as doing b(2) + b(3) + b(4)
b(0, 5) #returns 1.0

More Examples:

discrete.uniform(5, 12)(10) #returns 0.125
discrete.uniform(5, 12)(5, 12) #returns 1.0
discrete.hypergeometric(10, 5, 5)(4) #returns 0.0992063492063492
discrete.geometric(0.6)(5) #returns 0.01536
discrete.negativebinomial(5, 0.6)(8) #returns 0.1741824
discrete.poisson(5)(3) #returns 0.14037389581428056

2. Discrete Probability Simulations
Test cases can be found in the DiscreteSimulations.py file

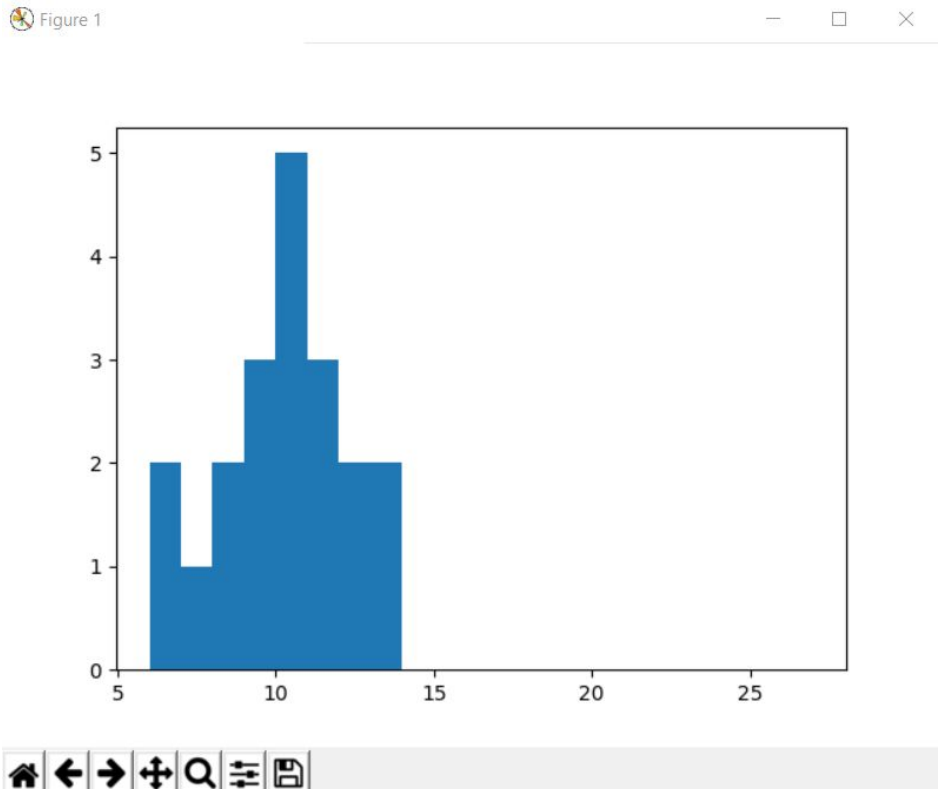
Relevant functions:

- discrete.generate_data(pdf, num_samples, sample_size = 1)
  - pdf is a probability distribution function returned by the other discrete functions such as discrete.uniform(5, 15), discrete.binomial(20, 0.7), etc
  - num_samples is the number of samples you want to take from the pdf
  - sample_size is the number of points inside of an individual sample, a sample size higher than one would make each one of your samples the sum of all the points in the sample
- discrete.histogram(data, num_intervals = 20)
  - data is a list of integers representing the x values in the histogram (like the data generated by discrete.generate_data)
  - num_intervals is the number of intervals (or classes) you want in your histogram, more intervals means that each bar on the histogram is thinner

Example:

```
prob_function = discrete.binomial(20, 0.5)
data1 = discrete.generate_data(prob_function, 20)
# data1 = [6, 6, 7, 8, 8, 9, 9, 9, 10, 10, 10, 10, 10, 11, 11, 11, 12, 12, 13, 13]
discrete.histogram(data1)
```

A frequency histogram would appear that represents your random simulation. It might look something like this:



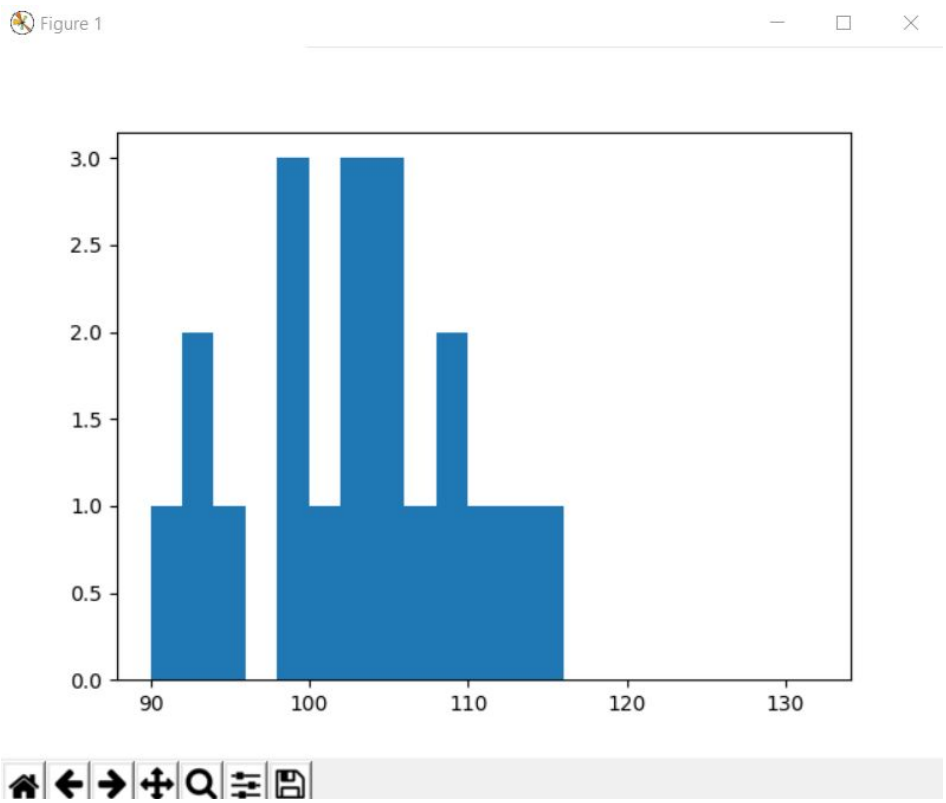
As you can see by adding up all the y-values, 20 samples of sample size 1 were taken from the binomial probability distribution. The width of the histogram is separated into 20 intervals, and it so happens that 1 interval has a width of one in this case. Note that the intervals include the lower bound and exclude the upper bound, so all of the values that ended up at 6 go into the bar of height 2 that looks like it is between 6 and 7. So, the histogram shows that in 20 samples, there was 2 samples of 6 successes, 1 sample of 7 successes, etc.

If we increase the sample size while keeping the same pdf,

Example:

```
data2 = discrete.generate_data(prob_function, 20, 10)
#data2 = [90, 92, 92, 95, 98, 98, 99, 100, 102, 103, 103, 105, 105, 105, 106, 108, 108, 110, 113, 114]
discrete.histogram(data2)
```

We might obtain something like this:

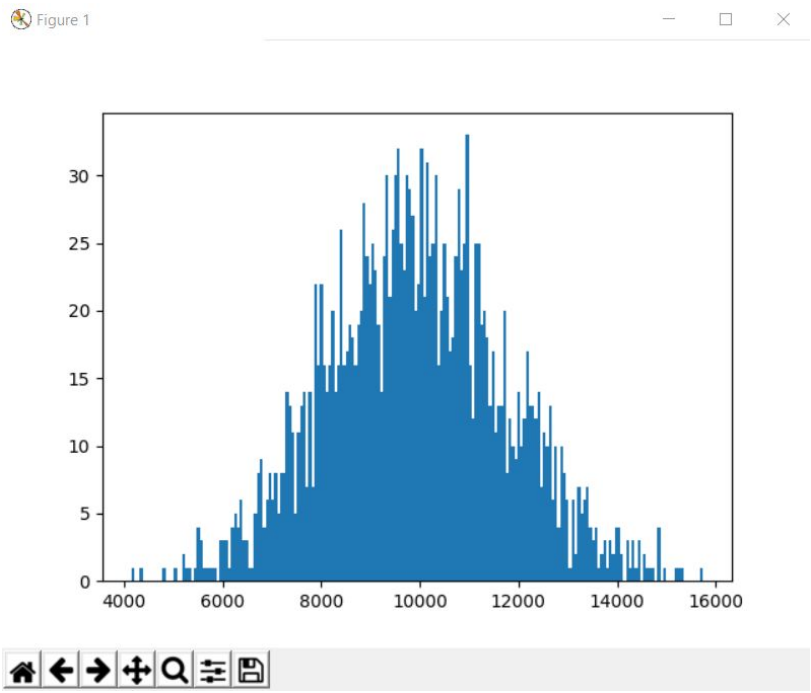


Here, each sample is the sum of 10 x-values randomly chosen from the binomial distribution, so the numbers in the x-axis are much larger. Also, since the range of values is larger but the number of intervals is still set to 20, each bar now has a width of 2.

For larger simulations, we could increase the number of intervals for a more accurate picture

Example:

```
pdf = discrete.uniform(0, 2000)
data3 = discrete.generate_data(pdf, 2000, 10)
discrete.simulate(data3, 200)
```



Now each individual bar becomes tiny (and notice how it approaches a normal distribution with large samples and sample size)

3. Continuous Probability Calculations
Test cases can be found in the ContinuousDistributionsTest.py file

Relevant functions:

- continuous.uniform(c, d)(start, end)
  - c and d define the bounds of the uniform probability distribution
  - start and end are the bounds of the interval for which you want to calculate the probability

- continuous.exponential(l)(start, end)
  The exponential function serves to calculate the waiting time until the first occurrence for a poisson process with mean lambda
  -l is the lambda parameter of the poisson process
  -start and end are the bounds of the interval of time for which you want to calculate the probability

Examples:

```
u = continuous.uniform(0, 10)
u(2, 5) #returns 0.3

e = continuous.exponential(0.5)
e(1,3) #returns 0.38340049956420363
```

External Documentation

Data structures

Overall, the package did not use any specialized data structures. Data was stored only using lists in most cases.

Algorithms

One algorithm worth noting is the algorithm used for mapping random numbers to x-values for the random generation of points.

The general idea behind it is that the random() function produces numbers from 0 to 1, and the cumulative probability function of any probability distribution function also goes from 0 to 1. As such, the random numbers can be mapped to the x-value of the smallest cumulative probability that is greater than the random number.

For example, if the cumulative probability function, F(x) had values F(3) = 0.6 and F(4) = 0.9, The random number 0.7 would map to 4. This way, we get to generate random samples based on the probability distribution function used.

The process involves sorting the list of random numbers and initializing a cache list to store the cumulative probabilities, only containing F(0) to start off. Then, a for loop is run through the random number list. For every number, it checks whether or not the it is smaller than the largest cumulative probability in the cache (the last element of cache). If it is smaller, than it would map to the x-value of the last element in the cache. Else, bigger cumulative probabilities are generated and appended to the cache (F(1), F(2), etc) until the most recent cumulative probability is bigger than the random number, then we can map the random number to that x-value. This algorithm works because the list is sorted, so an element will never map to a lower x-value than the previous element, ensuring that the last element in cache is either the correct cumulative probability or smaller than the random number.

Math Corner
(A few interesting observations confirming theoretical concepts, for fun!)

Central Limit Theorem

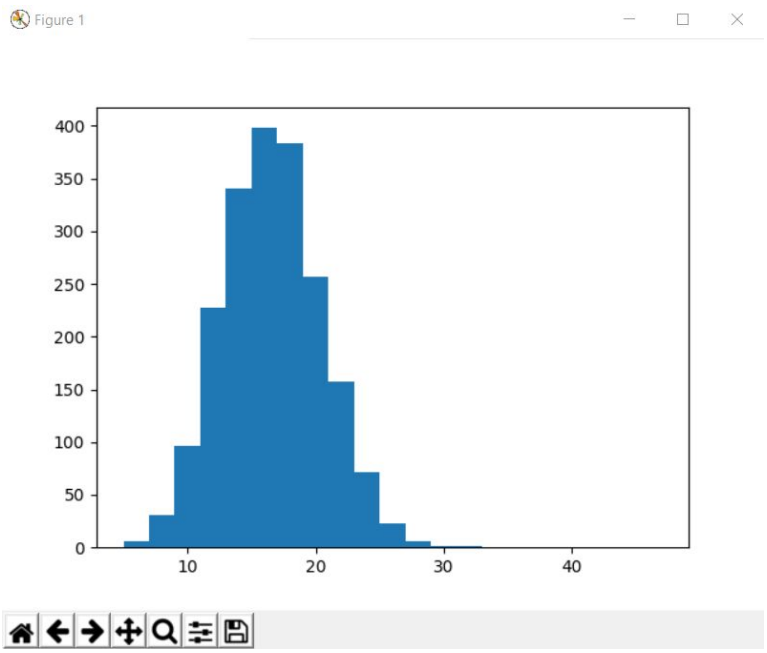
- Any distribution with a large number of samples and a large sample size begins to look like a normal distribution. (See DiscreteSimulations.py)

The relationship between poisson and binomial

- The poisson distribution with lambda = np can be used to approximate the binomial distribution if n is large and p is small
- Example:

```
b = discrete.binomial(200, 0.8)
Data1 = discrete.generate_data(b, 2000)
p = discrete.poisson(200*0.8)
Data2 = discrete.generate_data(p, 2000)
discrete.histogram(Data1)
discrete.histogram(Data2)
```

The histogram for the binomial distribution looks like this:



While the poisson histogram is very similar: