Stephen Murphy

Probability and Applied Statistics

Project 2

March 2022

***Distributions:***

For the distributions I rewrote the factorial class as a completely new class called DistFac which takes the two factorials needed for combinations and subtracts them because they cancel eachother out anyway. This makes it easier for the DistFac to be run so that it doesn’t overload the IDE outputting a 1 or NaN.

**Binomial:**

For the Binomial Distribution I called the combinations class and took each individual part of the equation and broke it apart to make it easier to understand and to make a simpler equation to be returned. In this case you take the Combination of the *number of trials* and *number of successes = c*. Then by raising the *probability of success* to the *y* power(*y = number of successes*) = *j*. Then l is the probability of failure raised to the power of failures out of the total trials (*trials-successes = k)= l*. Then by multiplying these together (*c\*j\*l*), we get the solution for the Binomial Distribution which is a percent chance that a success will happen this many times out of a specified number of trials.

We use Binomial Distribution when there are two outcomes of a trial either success or fail. Some examples of this are flipping a coin, rolling greater than 3 on a die, or pulling a diamond out of a deck of cards. The probability of the event doesn't matter as long as the only outcomes are success and failure.

**Geometric:**

For the Geometric Dist there is no need to call the combinations formula because the formula is very simple and doesn’t use it.With 3 parameters, (*p,q,y*) *p* is the *probability of success*. *q* is 1-*p* to the power of the *number of trails-1*. Which when I wrote in the program I just took *q* to the power of *y-1 = j* and then returned (*j\*p*). This distribution is used to determine the likelihood of success within a specified number of trials.

We use Geometric Distribution to find the probability of finding a success within a specified number of trials. For example, the odds it would take to find someone who likes chocolate over vanilla when we know the probability of someone liking chocolate over vanilla is something specific.

**Hypergeometric:**

The Hypergeometric Dist calls the combinations class, and by using the combinations class make it simpler to do the calculations of the Dist. This class has a method with 4 parameters *N, n, r, y,* where *N = total, n= how many taken from total, r = amount out of n selection group we want,*  and *y = amount of successes.* In this distribution we don’t just use the variables we had, we have to perform simple math to get exactly the right numbers to use. So in this case *N-r=q, n-y = w.* Declaring these makes it simpler to input into the Combination class.

The Hypergeometric Dist is used when we want to find the probability of a specific number of successes out of a sample size without replacing any of the pieces taken from the sample size. An example of this is the odds of pulling 4 kings from a deck of cards without putting the kings back in the deck after you pull them.

**Poisson:**

For the Poisson Dist I made it have 3 parameters, these parameters include *K = Total number of events, n = number of units, y = number of trials.* This class calls the Factorial class. *K/n = λ* which in this case is *l*, to the power of *y* and taking the factorial of the *number of trials, y* we divide these and multiply by *e^(K/n).*

This distribution is used to find a rare occurrence within an extremely large population. Some examples of this are finding someone who likes pens more than pencils at a convention for people who like pencils more. Or finding the number of meteorites that hit Earth in a year that are larger than 1 meter.

***Function Plotter/Salter/Smoother:***

**Plotter:**

For the plotter of this function, we call an equation class which holds the equation with a parameter on what the x-input is. In this case the equation I chose was *3x^2 + 2x - 1.* Having a parameter made it easy to call the class and use a for loop to input the values of x up to a specific value. For the plotter it does just that. It calls the equation class and uses a try and catch method to make and write to a .csv file called *eqouts.csv* this method also has a parameter which runs the for loop as many times as the number that is input. The plot method then writes out the number that is input into the equation, then whatever the output of the equation is and proceeds to skip a line. This runs through however many times as the parameter says. I chose 45 as an example.

**Salter:**

The Salter class initializes an ArrayList called *salt.* The Salt method has two parameters, one to say how many inputs there are as well as the value that should be salted into the data. This class also uses the random class and the equation class. Using a try and catch to write the file, a for loop will initialize an array called *change* that has two elements *{-b,b}.* *b* is the secondary parameter used in the Salt method. This will either add or subtract the value specified as *b.* Whether *b* is added or subtracted is determined by the *pick* variable which uses the random class. It picks a random position in the *change[]*  array and that's what will be added to the equation. The *x input* as well as the *y-value* are written into the .csv file but the *y-value* is also added to whichever element of the *change[]* array was chosen. But every time the *x input* is changed it randomly selects an element from the array. Then each time this for loop is run the salted *y-value* is added to the ArrayList *salt*. The output .csv file is named *salted.csv.*

**Smoother:**

The Smoother class has a parameter that says how many times the for loop will run, basically how *x inputs* we want it to go through starting at 1. In order to smooth the data we take the average of the datapoint before the current, and the datapoint after the current. For example if we have *x=3,* we average *x=2,* and *x=4.* The way the smoother does this is by calling the *salt* ArrayList. It runs through the elements of the array and as long as the lower and upper bounds of the averaging are within the ArrayLists’ bounds then it will find the average for that point. So we have (ArrayList *salt* element 1 + element 3) / 2 = element 2 ±3. The smooth method goes through the entire list until it reaches the second to last number because there is nothing to average. It then writes these averaged numbers as well as the *x inputs* in a .csv file called *smoothed.csv.*

**Graphs:**

Overall the graphs of the original, the salted and the smoothed are similar because the range is so big. If I had chosen a smaller number like 20 for the amount of inputs, there might have been more skewed data points. Even so, there is a clear difference between the three graphs. The original has all its data points where they should be so it is the cleanest graph, the salted is where you can see the bumps and the dips, and with the smoothed graph you can see many of the bumps and the dips have been removed and it gets closer to the original function though it is not perfect. These graphs are at the end of the paper as well as in the excel sheets included within the Github project.

**Poker Hand:**

Using the Deck class I declared another ArrayList called *shuffled*. This stores the shuffled deck of cards after the ArrayList *deck* is put through the imported *util*.*Collections* class. Once the deck is shuffled the 0 element of the *deck* ArrayList represents the top of the deck of cards and it is taken out by the *drawCard()* method. This method also adds that same top card to a separate ArrayList called *hand* that is initialized in the *Hand Evaluator* class.

