The first code that I did for project 2 is a modification on the CSV writer that was created for project 1. The file writer method is still the same as before were it writes to a csv file and returns an error if it cannot write to said file. The difference lies in the outputSingleLine method. The single line method now will output data based on the equation “x2”, salted data from that equation, and smoothed data from the salted data list. The method creates a new buffer reader and a new smoothed number array list to be used later in the code. The top of the code also specifies the upper boundary and the lower boundary of the salter that is contained in the method. As for the salting part of the method, the x squared equation is run through again only this time a value between -100 and 100 is selected to be added to each value on the table. This in turn will “salt” the data and will make the graph of the equation seem more jagged. The last new feature of the outputSingleLine method is the smoother. This part will take all of the numbers in the salted column and add them into an array called smoothedNumber. The code will then take a number in the index and add that to the previous number in the index. Once it accomplishes this, it divides it by two to get the average of the numbers. This average is then presented as the smoothed data all the way down the list. This continues for the whole list except the first value in the list. The code will simply put this value back as it is in the smoothed data. An example for this would be if we had three numbers: (10, 20, 30). The first number in the smoothed data will be 10 as it is the first in the list and it follows the rule specified earlier. The next number in the list will be 10 plus 20 divided by 2 which gives 15. Finally, the last number on the list would be 20 plus 30 divided by 2 which would be 25.

The next code that I worked on for this project is the poker hand analyzer. I worked on this code alongside Joe DeLizzo. The code is a collection of three classes and a tester. These are the Card class, the Deck class, the HandEvaluator class, and the Poker Hand Tester. Within the Card class lies the program’s getter and setter methods. These are for getting all of the numerical values and the suites for each of the cards that are generated in one deck. The last part of the Card class is a compareTo method. This allows the code to compare two cards in a set of five cards by number.

The second class is the Deck class. Inside of the Deck class lies the deck generator and the array to get the cards generated. The generator has a string array with all of the suites in a deck of cards (spades, clubs, diamonds, and hearts). The generator will generate 13 numerical values for each suite (ace, jack, queen, king is considered 1, 11, 12, 13 respectively) making the total number of cards per deck 52. The class has a method that will draw cards from a created deck. Once a card is selected from the deck, it removes it so it may not be selected again on accident.

The third class is where all of the code is resided in. A deck is created along with a hand of cards. An array of five cards is created that acts like a hand in poker which runs the drawCard method in deck 5 times. A method called removePair is also in the third class that will take any pair that the hand has and put it in another array for analysis. This is used for later methods to determine different winning scenarios. Once the hand has been created, the hand goes through many Boolean methods to determine what type of winning options the hand has. The first method is the check pair method. This will check the numerical value of each card in the hand for any possible matching numbers. The method will comb through the hand by comparing the hand with each card individually. If at one point the numerical value of one card matches another, then the method returns true. If there are no matching numbers, the method comes back false. The second method is the checkThree method. This will remove the first pair that the code sees and looks through the remaining hand. If any card inside the remaining hand has a matching numerical value to the pair that was extracted, the method returns true. If there is no card that matches the pair extracted or there is no pair at all, the method returns false. The third method the hand of cards goes through is the checkFullHouse method. The method will attempt to find a pair to extract. If there is no pair in the hand it comes back false. If there a pair that has been removed, the method will run the checkThree method inside of the checkFullHouse method. If that comes back as true, then the whole method returns true. If checkThree comes back false, the method will return false. The next method is the checkFour method. This will attempt to extract two pairs from the hand. If there is not a second pair to begin with, the method will return false. Two integers are made that hold the numerical value of both of the pairs. If the value for one pair the same as the other pair, the method returns true. Otherwise, the method will return false. The fifth method in the HandEvaluator is the checkTwoPair method. It is set up similar to the checkFour method only this time extracting one pair and not two. If the code again cannot get a pair from the hand, it returns false. If the method does receive a pair to extract, then the code runs the check pair on the remaining hand. If the remaining hand has a pair in it, the checkTwoPair method comes back true. Otherwise, the method comes back false. Both the checkFlush and the CheckStraight methods are handled the same way only what they look for is different. Both will go through each card in the hand and look for similar suites for flush and similar numerical value for straight. If at any point in the search there is another suite or number value, the checkFlush or checkStraight will come back false. If every card has the same suite or the same number, then checkFlush or checkStraight comes back as true. The final method is a way to keep track of all of the instances where “true” came back for any of the previous methods. This is called trials and will run based on the number of times that the user specifies in the tester. There is a counter in every if statement that goes up if there is a true value for the method every time its run. After each method is checked for a true value, the deck is reshuffled, and 5 new cards are distributed. This is to give the methods all five cards again after some methods remove pairs of cards. Once the number of trials is complete and all of the counters are tallied, the percentages of each method’s number of true values are printed out rounded to four decimal places.

The last code that I worked on this project was a modification of the PermAndCombo code that was submitted in project 1. The difference now there are all of the probability distributions are coded in. These include the binomial method, the geometric method, the hypergeometric method, and the Poisson method. The binomial method takes the combination of an integer *n* and an integer *r* and multiplies it by the percentage of success *p* raised to *r.* It is then multiplied again by the probability of failure *q* raised to the power of (*n –* the amount of something desired *y*). The second method is the geometric method. This takes the value of *q* and raises it to the power of (*r-*1*)*. Once that is done, the method will multiply that value by the value of *p*. The third method is called the hypergeometric method. The hypergeometric method takes 3 combinations and will multiply or divide them. The first combination of r and y is multiplied by the combination of (N-r) and (n-y). Once that is done, it is divided by the combination of N and n to get the final answer. The final method is called the Poisson method. The Poisson method will take the "a" value which represents lambda is raised to the power of y. After, it is divided by y factorial. The answer of that is then multiplied by mathematical variable e to the power of negative lambda.

The key to knowing which probability distribution to use is figuring out what needs to be figured out in the problem. For example, binomial distribution should only be used if a problem arises with only one of two outcomes can occur per trial. The cases are on a per trial basis meaning each trial is independent of each other. In order to use geometric distribution, the problem must be trying to ask how many trials something takes in order to achieve one of two outcomes. This does not mean the amount of that one outcome rather, the number of trials or time it takes to get to that outcome. In order to use hypergeometric distribution, the question must be asking probability of a certain number of successes in a certain number of trials. This comes without substituting anything about the original problem. For example, if someone wanted to get one red marble out of a bag with 4 red marbles and 5 blue marbles in 3 tries. Without replacing any marbles in the bag, what is the probability of this even occurring in the amount of tries specified. Finally, in order to use Poisson distribution, the problem must be asking to predict something happening in a given time interval. This means that the problem gives you the probability of something happening. Then the problem asks the amount of time or tries that the trial has. From there it will ask the probability of the event happening a certain number of times in that period of tries or time.