Bryan Xu
bryanxu@ucsb.edu
6849707

Note: value = obese, sex, type, diabetes, etc. category = dead/alive

My code consists of a main function that calls two other functions. The first function reads in the input from the CSV file, and stores the values in their corresponding variables. Using a series of if statements, I was able to determine which variable each of the values belonged to. I also included an "unknown" variable for every value, and left all of the "unknown" variables as their own categories. This is what I usually had some variables: dtubed1, atubed1, dtubed2, atubed2, dtubed3, atubed3, which was repeated for all of the variables. The first character "d" or "a" denotes if the statistic belongs to a dead or alive person, respectively. The next characters denote the original category of the variable, and the last number, denotes the type, where 1 corresponds to 1, 2 corresponds to 2, and anything else corresponds to 3.

My first function, countAllVariables(string filename), reads through filename, and stores them into the 120+ variables that I created at initialization. For dates, I completely ignored both the entry_date and date_symptoms category because it was too much of a wildcard to parse.

My second function, willDie(int sex, int type… int icu) computes the probability that a person will die with the given variables. This is where the naive bayes computation is done. I used a summation of the logs of the probabilities in order to determine which outcome was more likely. In addition to that, I also did a little smoothing by starting each value off at 1, which would avoid any possible log(0) cases. At the end of the function, I simply compared the probabilities and returned the one that was greatest.

In my last function, printAllTestCases(string filename), I opened the given filename, and stored the values in each line to a corresponding variable. This included things like int type = 1, if the number in the "type" value was 1. At the end of each line, I made a call to willDie(), and printed out the output.

In order to represent the document, I created all of the variables beforehand, for each possible case of the variable. This resulted in having over 120 variables, and is not scalable for bigger projects. As mentioned earlier, since cleaning the dates for entry_date and date_symptoms were too troublesome, I ignored them completely. For the age values, I split the age ranges into two, the first from less than 50, and the second from greater than 50. The number 50 was a pretty arbitrary number, in that it was the first value I chose, and seemed to have good results. After each test case was printed, I overwrote the variables with new ones, which saved some space.

In order to train my naive bayes, my method was to go through the entire csv file, count the number of each value, and then do the appropriate naive bayes calculation. The calculation involved adding up all of the log values of each probability to both the death and surival probabilities (for example, adding dsex1/dcount to the probability of death, and adding asex1/acount to the probability of surviving, if the sample had sex = 1 as the parameter). By simultaneously calculating the probability of survival and death at once, there was some running time saved. Then, the probability that ended up with the higher value became the more likely outcome.

My results for the provided datasets was 100% accuracy, with a run time of 1.09375 seconds. I believe that my code runs at O($N^2$), because my function runs through each value exactly once. In order to increase my accuracy, I took the time to submit to the auto-grader many times. Each time, I would change the weights of one of my variables inside my willDie() function, and recorded the changes. Eventually, I settled on several variables that I deemed were more important to increasing the overall accuracy of my program. I increased the weights of: age, diabetes, contact_other, intubed, obese, tobacco, covid_res, and, pneumonia. I decreased or entirely removed the weights of cardio, copd, asthma, sex, type, other, renal, hyper, icu, and, preg. My method for finding my weighting was entirely guess and check; I ended up taking a long time before I finally settled on the variables. A lot of my testing can be found directly in my nbc.h file. While I cannot say which 10 variables were the most important, I found that increasing the weights of age and diabetes had the most dramatic improvement in the early stages of the testing.

The biggest challenge for me was finding out a way to accurately update the variables all in one pass. This was difficult because the class label wasn't the first or last label of the dataset, so I had to figure out a workaround so that I wouldn't have to parse backwards. My solution instead became to increment the counters of both alive and death labels, and decrement from them once I determined the correct class label. For example, I would increment both dsex1 and asex1, but when I reached the class label of alive, I would decrement dsex1, thus cancelling out the inaccurate prior increment. Besides that, the most tedious part was finding a way to represent all of the variables. I ended up hard coding the variables, and it worked, but in the future, I would like to find and implement a data structure that can do it more efficiently.

The biggest weakness in my method was that it was horribly inefficient to increase the accuracy or the weighting. I had to manually comment out the block of code, or edit 6 different lines to change the weighting for one variable. Another weakness is that my code is not flexible, and that it cannot be generalized to different datasets well. Since all of the variables are hardcoded, this leaves a lot of room for improvement. For fixing my weighting, I think that I could implement a helper function that could adjust the weighting of any specified variable, so that it would be easier to change during future testing. For the issue of hardcoding my variables, I aim to find a better representation of my data, namely finding a data structure that can support efficient memory storage, such as a map or dictionary. Overall, this MP taught me a lot about how AI actually works, and how different weights can be changed to achieve better results.