The first approach that I implemented was the naïve bayes approach. In this approach, I calculated the posterior using the prior times the likelihood over the evidence. No explanation of libraries is needed for this approach because no major libraries were used (coded from scratch). Because the features are binary and not continuous, probabilities are not difficult to calculate and thus methods such as Gaussian Naïve Bayes is not necessary. For the prior, because we were not given the distribution of the classification in the testing set, I decided to estimate the classification probability of the testing set by using the distribution of the training set instead of assuming equiprobability of either class. Later, I tested found that doing so resulted in a slightly higher rank.

In this Naïve Bayes approach, I counted the frequencies of each feature and added them to a frequency map of either an active feature map or inactive feature map. With this, I was able to calculate the “prior” and “likelihood” values. The prior value was simply two values (one for the inactive class and one for the active class) that represented the percentage of the number of records classified as one class over the total records. However, with the imbalance of the distribution of classifications within the training set, I faced a decision to either keep this prior value true to the general Naïve Bayes approach or to anticipate and change the prior value to better match the classification distribution of the testing set. I ended deciding to keep the prior value as the original percentage explained above because I felt that I did not have enough information to confidently assume that the testing set would have a different distribution from the training set. For the likelihood values (active class likelihood and inactive class likelihood), I took the product of the probability of each feature given a class (active or inactive). With these prior and likelihood values, I multiplied the prior and likelihood for each class and compared them against each other to determine which classification has a higher probability.

During implementation, I was getting a lot of zero probabilities during the algorithm’s classification phase. This caused a lot of ties in class 0 and class 1 probabilities, all with values of zero. I found that a lot of the probabilities calculated to 0 due to the sheer sparsity of the available data. Because of this, I had to determine how to implement feature selection to heavily limit the amount of 0s that the calculation would receive. The first feature selection technique I decided to use was a simple threshold against any shared features between the active and inactive input set. If there were about the same amount of a given active feature in the active set as there is in the inactive set, then the feature would be removed from consideration. When put to the test, this technique did not yield a much higher score as I anticipated. With this, I did not think to consider normalization for this technique as I did not think it would prove useful for receiving a higher score anyway.

Because of the disappointing results from the implementation of the first feature selection technique, the second feature selection technique I decided to use on the Naïve Bayes model was an extreme measure to combat the sparsity of the input data and thus the amount of zero probabilities calculated. I removed all features that exclusively resided in either the active set or the inactive set. This yielded in safer results with ~85% of the classifications residing in the inactive/zero class. However, even with this, I ended with my highest F1 score for my Naïve Bayes approach of 0.57.

The second approach I implemented was a Neural Network implementation. Instead of coding a multilayer perceptron from scratch, I decided to use a library called “Keras” to handle most of the lower-level model/layer compilation and training. This way, I can focus on what the structure of the multilayer perceptron looks like as well as the many different parameters which includes deciding which algorithms to utilize.

For the general model, I originally started the implementation with one input layer, one hidden layer, and one output layer. I used a standard sigmoid activation function for the output layer which can output a value from a range of 0 to 1. During this initial parameter tweaking process, I also tried using several different activation functions for the hidden layer such as selu, relu, tanh, and elu. I ended up sticking with the relu (rectified linear unit activation function) which just returns zero for any negative input and returns the same number as the input if the input is positive. This worked well for the assignment’s heavy utilization in probability due to it being a problem based in binary classification. Another thing I considered due to this being a binary classification problem, I used a binary accuracy metric that optimizes the neural network by calculating the accuracy against a reference testing set. This optimization occurs during the training phase where training and testing data needs to be fed to the network along with the associated classifications for each record. Because we were only given one set with the classifications associated with the records (the training set), cross validation was done by splitting the training set into ¾ training set and ¼ testing/validation set. With these two sets being fed into the network during data fitting, the network was able to train and optimize at the same time. For dealing with the input data class imbalance, I set a parameter called “stratify” to the set of classes that correspond to the given training set (0s and 1s). This is so that during this cross-validation data split, the testing/validation data set will have the same class ratio as the training set. The specific optimizer method used was the momentum based gradient descent (“sgd”) function (momentum meaning descent is accelerated and correction is dampened to prevent any major oscillations or overcorrections). During optimization, the different algorithms work together not only to maximize accuracy, but to minimize loss. In my implementation, I chose for loss to be calculated with the binary\_crossentropy which is a useful function specifically for binary classification problems. After some more tweaking, I decided upon 200 neurons for the hidden layer and 256 for the input layer to minimize loss. For this initial draft implementation of a neural network, my miner2 F1 score resulted in 0.52.

In the initial draft, I chose arbitrary training values during the model’s data fitting/training phase. This is worked well as a draft to determine the other important parameters (like the optimizer, loss, and activation functions), but it is most likely a source of some loss. The two main parameters to tweak are the number of epoch iterations and batch size. The model generally improves (less loss) with more epochs and smaller batch sizes can converge faster but larger batch sizes allow training as a whole to be faster. I also noticed that for homework 2’s specific training dataset on my model, loss and accuracy started to converge at 12 epochs but I wanted to be safe so I kept the number of epochs for the model at 24. For the batch size, since we are working with a relatively smaller dataset, I found that lower sizes resulted in slightly less loss, so I left it at 32 after tweaking. I knew that due to the sparse nature of the input data, overfitting was likely to occur during training. To further improve my score, I thought of different ways to combat this probable overfitting. I found out that Keras offered a dropout layer module and so I added one more hidden dropout layer between the first hidden layer and the input layer. What this dropout layer does is it takes in a percentage of (input) units to drop (for my implementation I chose 50%) and it randomly drops that percentage of units from the previous layer. In practice, dropout layers should regularize the network and prevent overfitting to a specific training set. This technique was useful as it bumped up my miner2 F1 score to 0.69. By the end, the running times for each epoch average at around 80 ms and the whole script took around 24 seconds to finish (with 24 epochs and a batch size of 32). Decreasing the batch size or increasing the number of epochs would increase the running time by a few of seconds each increment/decrement.