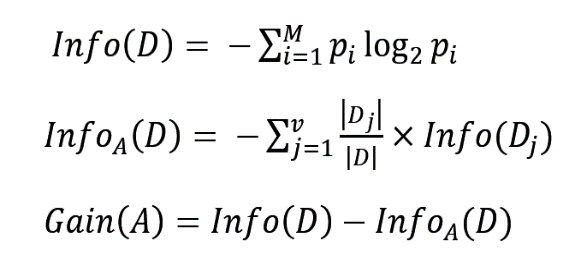
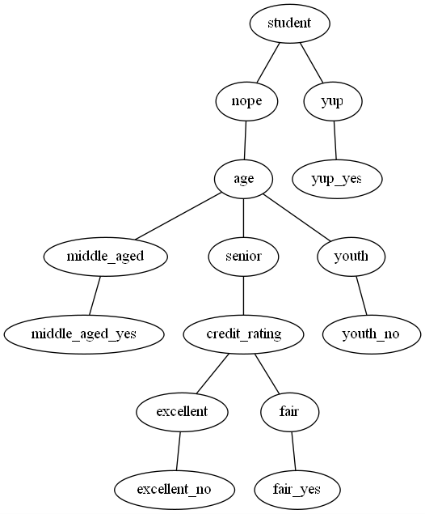
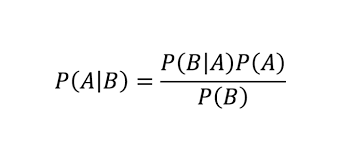
**Heart Disease Classifier**Team: Two Dudes

**Team Members:**  
Stephen Garcia, Marco Zamora

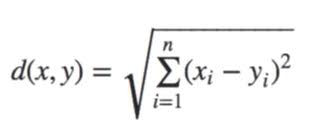
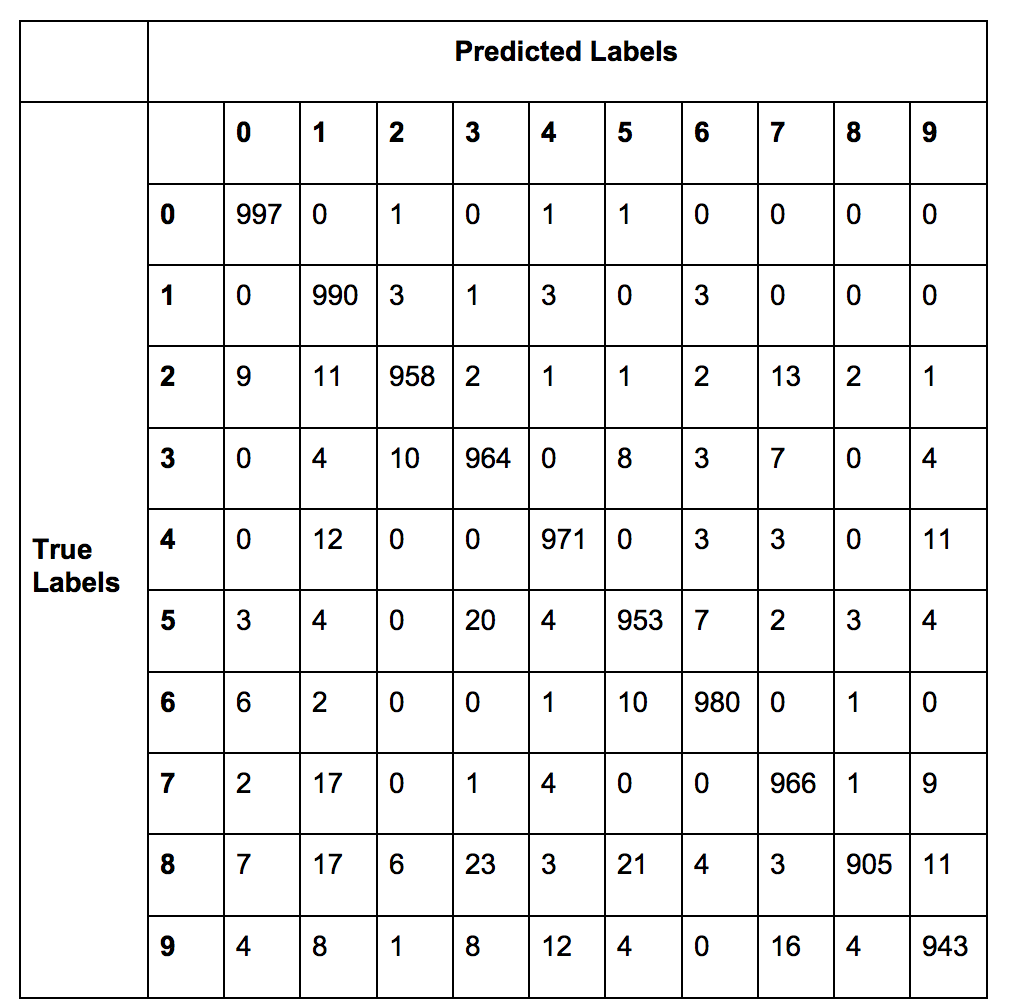
**Abstract:**Classification models are an important tool used for supervised learning of data. For our project we chose three classification algorithms to learn more about through implementation, which include a Decision Tree, Naïve Bayes, and k-Nearest Neighbor classifiers. Our project was initially meant to classify a specific heart disease dataset; however, we quickly realized that while the data was interesting and offered exciting options, it was also adding layers of complexity into the project that wasn’t conducive towards completing the programs in the time allotted. We subsequently worked on more generalized datasets, which were more conducive towards completing our task.

**Design, Implementation, and Testing:**The first step in the project revolved around preparing the data for classification. We provide to the user the option of selecting a specific csv. We perform some validation to ensure the file exists and then read it into a Pandas DataFrame. We give the user the choice to view the captured data and validate the proper data was loaded into the application. The application immediately splits the data into our training and testing sets with a 70/30 split. Once the data is loaded, we return the user to the main menu to proceed with selecting a classifier.

The first algorithm we worked towards completing was the decision tree. Constructing the functions to calculate the entropy and information gain were straight forward. We were able to leverage the work done in class and homework to verify that the measures were calculated correctly. The calculations for Entropy, Weighted Entropy, and Information Gain of *D* are:  
Outside of our novice experience using Python at a level required for constructing an application, the complexity revolved around constructing and displaying the tree. We decided to construct a nested dictionary using recursion, which could later be traversed in order to classify the data. In addition to the key-value paired text-based tree, wanted to give the user a graphical representation as well. This functionality took some time and research to try and implement. We decided to construct a graph using pydot in Graphviz. While constructing the tree we noticed that complex datasets produced duplicate edges, to work around this issue we introduced a list into the build to check to see if an edge had already been established; if so, we ignore the new connection and resume the drawing of the next node. There is also a quirk that creeps up when the nodes have the same name, in order to get around this we ensured that the possible data values for each feature had unique names. While this isn’t ideal for real world application, it’s something that was necessary to complete the project. Future iterations of the code could implement a richer set of graphical outputs. The final product isn’t perfect, but works well with smaller datasets cultivated to work around these two issues. The functionality of the decision tree also gives the user the option of inputting a tuple to classify; however, domain knowledge would be required in order to use this feature. We provide sample data for a specific dataset in the user manual. We tested the decision tree with the 30% cut from our data and



The second algorithm that we work on was the naïve bayes. To develop this algorithm, we used multiple sources to shape the final product that we are delivering. The way we were able to make a classification was to make a model that used a dictionary that had dictionaries for each class. Each of these class dictionaries held the features that would result in a certain target. After the dataset was separated by classes, the classes were run through a few functions that would calculate the probabilities of each class. Once the classes probabilities were calculated they were returned as model that could be used to make classification decisions. To do that we developed another function that would take the trained model and a test list that we would like a prediction for. This function would return the predict class and the probabilities for each class for the user to see. We tested the accuracy of our model against the built in and ours performed at 80% accuracy and the built-in performed at 66%. Comparing both models our model did ~13% better.

The third algorithm that we worked on was the K nearest neighbors. Several sources were used in the development of this algorithm. The overall final product was mostly successful except for one small situation that would not allow it to return a proper classification. We will mention that situation later in the paper. First, we would like to explain how the algorithm works on the higher level. This algorithm was made of three main functions. The first would take the dataset and test set and find the Euclidean distance of each point in every row of the dataset compared with every point in the test set. This would give me a model that contained all the Euclidean distance. If the dataset that was given was categorical, we need to encode that dataset into integers before passing the dataset to the functions that creates the model of Euclidean distances. After we had the model of Euclidean distances, we ordered the distances, and once they were sorted, we would return the targets that were the closest to the test set and it’s predicted class. This is the basic way this classifier works, and like mentioned before it worked most of the time except during the following instance. This instance that the algorithm cannot return a proper classification is when the user asks for large number of clusters. For some datasets asking large number of clusters returns an error that states that something is out of bounds of an index. Our thinking is that it could be possible that for smaller datasets there are not enough data to develop clusters, or some other reason that we are not aware of. We also tested the accuracy of our model compared to the built-in one. Our model got ~80% accuracy compared to ~73% accuracy with the built-in one. Our model did ~6.667% better than the built-in model.