Census Data Classification

Tytus Planck and Kyle Rossman

Data Mining

Homework 2 – September 29, 2017

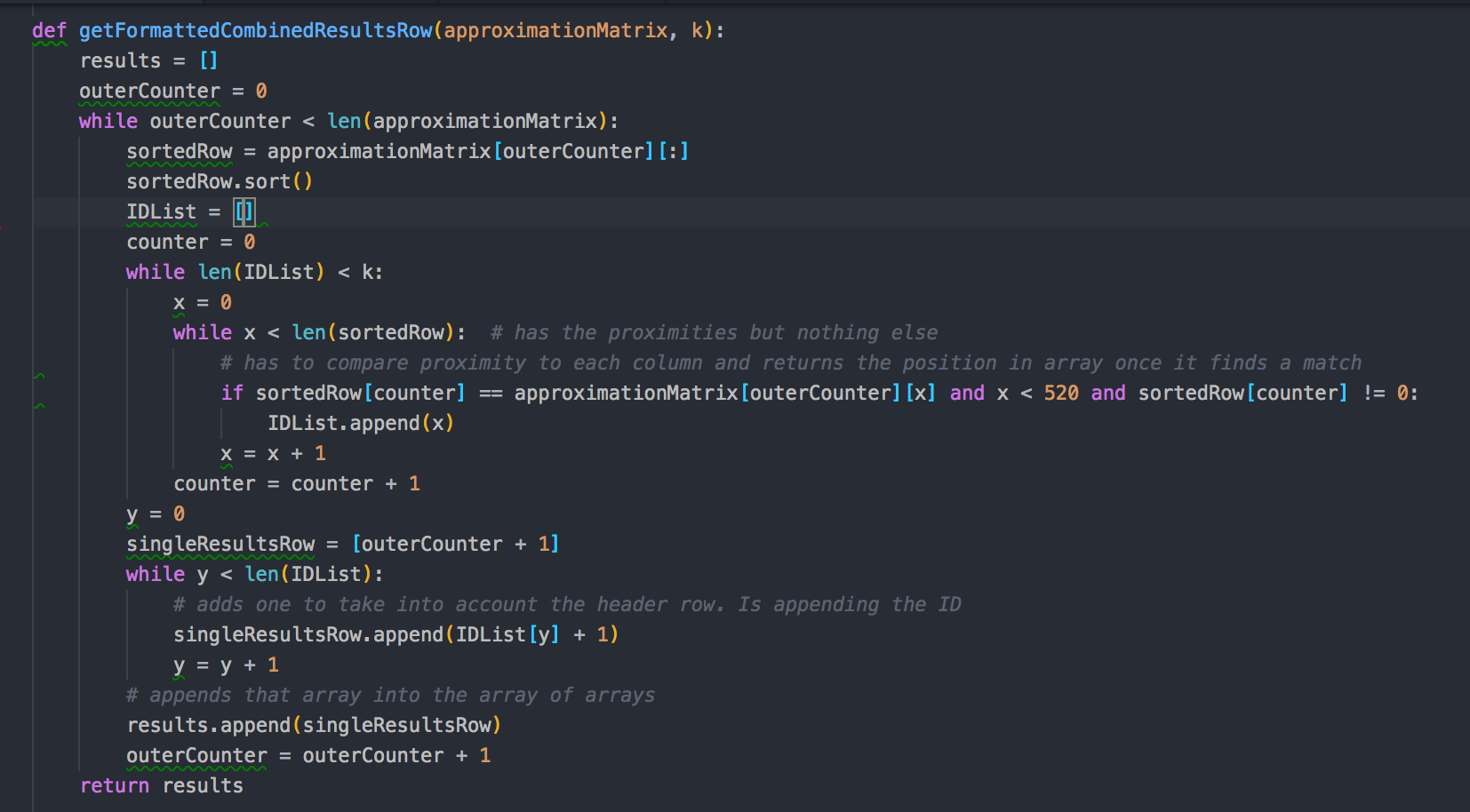
**Section 1 – Approach to Classification**

For this project, we combined the data analysis from the previous homework to then find a way to classify each row of a new data set. This classification was done by evaluating the “class” attribute in the training set to predict the class of instances of the test data set. The first step in approaching this task was determining what could be used of the already established code base, and what would have to be adapted in order for it to meet the new requirements. Next was grasping how to implement a logical solution to classifying the data through a kNN algorithm and applying Bayes theorem to calculate the posterior probability. Finally, we had to address the problems we were running into once we started actually writing code, and figure out the best way to combat these, not just from a correctness standpoint, but also in terms of efficiency of the program. The rest of the section will document each of these points in detail.

**Integrating Existing Code**

By already having the capability of using proximity measures to identify the closest instances to a given entry, part of the kNN problem is solved. We had a way to find the closest neighbors for entries in the training set. However, since what we need is to find the closest training set instances to an entry in the test set, we had to find a way to combine the two so that this calculation would be performed. The two options that were contemplated were either combine the two data sets into one matrix, or take the individual row from the test set and append it to the training set, then once we calculate its closest neighbors we remove it. The latter approach made the most sense, because we would not have to alter our proximity measure algorithm to exclude any of the test set as closest neighbors.

We were not sure what the cost would be to add an instance to the training set, run the proximity algorithm, and then delete the instance for all 288 of the test set instances, but since it caused us to modify the code the least we stuck with it at first. However, once we tried to implement this the program took approximately eighteen minutes to run. This forced us to backtrack and instead combine the two datasets into one matrix. This drastically reduced our run-time to around ten seconds, assumingly because our proximity algorithm only had to run once, versus 288 originally.

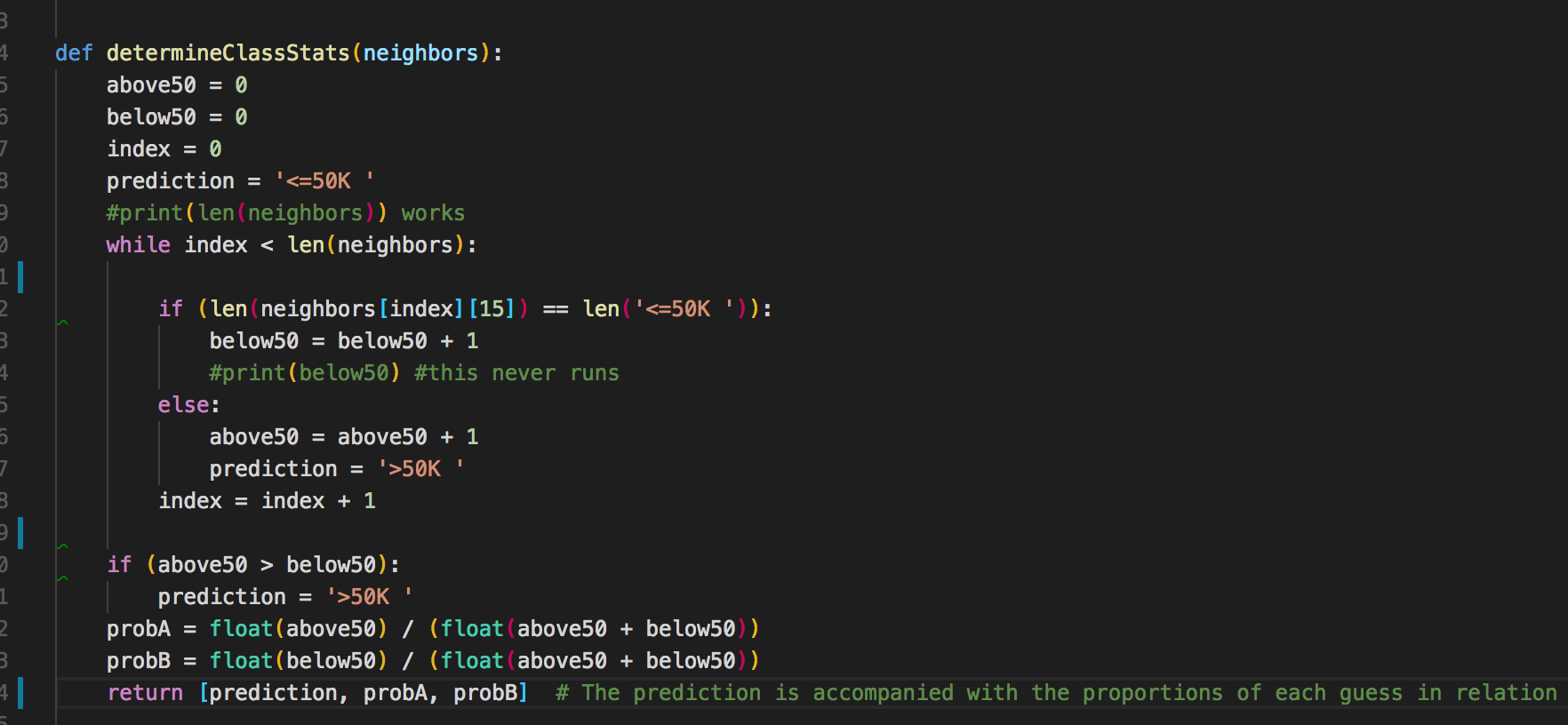


***Figure 1***

To adjust to combining our matrices some functions had to be modified to filter out proximity measurements for test set entries, but besides that the rest of the code did not change.

**Algorithm Implementation and Assumptions**

The majority of any new code that needed written to implement our algorithms was just to make a prediction and find the posterior proximity through Bayes Theorem for each record. For our prediction, we took the k-nearest neighbors subset and calculated which class was more prevalent in that subgroup. Then to implement Bayes Formula we divided it into it’s three parts and had a method for each.



***Figure 2***

A lot of thought went into how to calculate *p(x|+).* We ended up using the number of instances in the nearest neighbors set that belonged to the “+” class divided by the size of the nearest neighbor set. Since typically this would be the number of “true positive” predictions for a class divided by the total times that class was selected as the prediction, we had to adapt this to fit our kNN algorithm. Because we were basing our prediction off of the training set, we knew the class values of there would be no “false positives”. We felt that using the proportion of entries belonging to the class over the total entries was the best way to represent this, because it displayed our confidence that our guess was correct, under the assumption that if a majority of an instance’s neighbors belonged to one class we could me more confident that instance also belonged to the class. If a prediction was made but that class was barely above fifty percent of the neighbor set then our posterior probability would be lower, where as if the entire neighbors set was class “<50K” then the posterior probability would be high. This seemed to fit what Bayes’ Formula accomplishes so it made sense to find the posterior probability this way. The determineClassStats function from our code shows how we pulled the proportions of each class from the neighbors set and returned them with the prediction.

The other two functions involved in our posterior approximation was more straightforward. For p(+), or the probability of belonging to class “+”, we just calculated the number of training set records that were in that class divided by the total number of records. Lastly, for *p(x),* or the chance of getting any positive result, whether false or not, was calculated by taking the number of predictions for class “+” in the test set divided by the total number of records. Having three different methods to calculate this made the math straightforward. Consequently, with this posterior probability we could return whichever probability matched the predicted class and see how confident we could be in our predictions.

For Homework #2 we used almost the exact same assumptions as we did for Homework #1. Mostly that each row was completely independent of one another and that the data was relatively normal (i.e. not too many outliers). One additional assumption we had for this homework was the assumption that there were no duplicate rows in the testing data and training data sets. We actually specifically implemented this by eliminating any matches with a proximity measurement of zero indicating identical rows. We felt that duplicate rows could hinder the integrity of our analysis by potentially skewing classifications of individual rows of data.

**Troubleshooting and Data Issues**

Since a main theme in Computer Science and Data Analytics is that data is not always neat and tidy, it was not surprising that we ran into some issues that we had to traverse around. The biggest was that of our two CSV files, one read in data with quotations and the other did not. So, for example one record would have a class “<=50K”, and the other would just be <=50K. This was problematic because our entire proximity measure algorithm hinged on being able to match attribute values together, and the quotations made it appear as if two identical values were not the same. The reason for this problem was unknown since the exact same function was used to read in both CSV files, however it was fixed when the quotechar value for the reader function in python was modified to quotations.

The other big issue that was already mentioned was how slow our original program ran when we tried to implement our initial strategy of appending each test row to the training data, finding an approximation matrix, and then deleting the row. Although this allowed for us to have more concise data, it was actually extremely inefficient. It was taking almost twenty minutes for the entire approximation matrix to be built. We adjusted our strategy for producing this matrix and reduced our computation time to around 10 seconds. Once we switched to combining the two matrices together our main concern was containing data integrity. We had to make sure we removed the header from the second matrix, and go back and adjust all of our indices so that the data completely matched. So although combining our matrices complicated our analysis process at the end it allowed our program to run more efficiently, which we felt was a good trade-off.

Overall integrating our original code from Homework #1 turned out to be more difficult than anticipated. Unfortunately because we had two separate sets of data we had to adjust our code to compensate for the data to be combined for some operations and separated for others.

**Section 2 – Our Analysis**

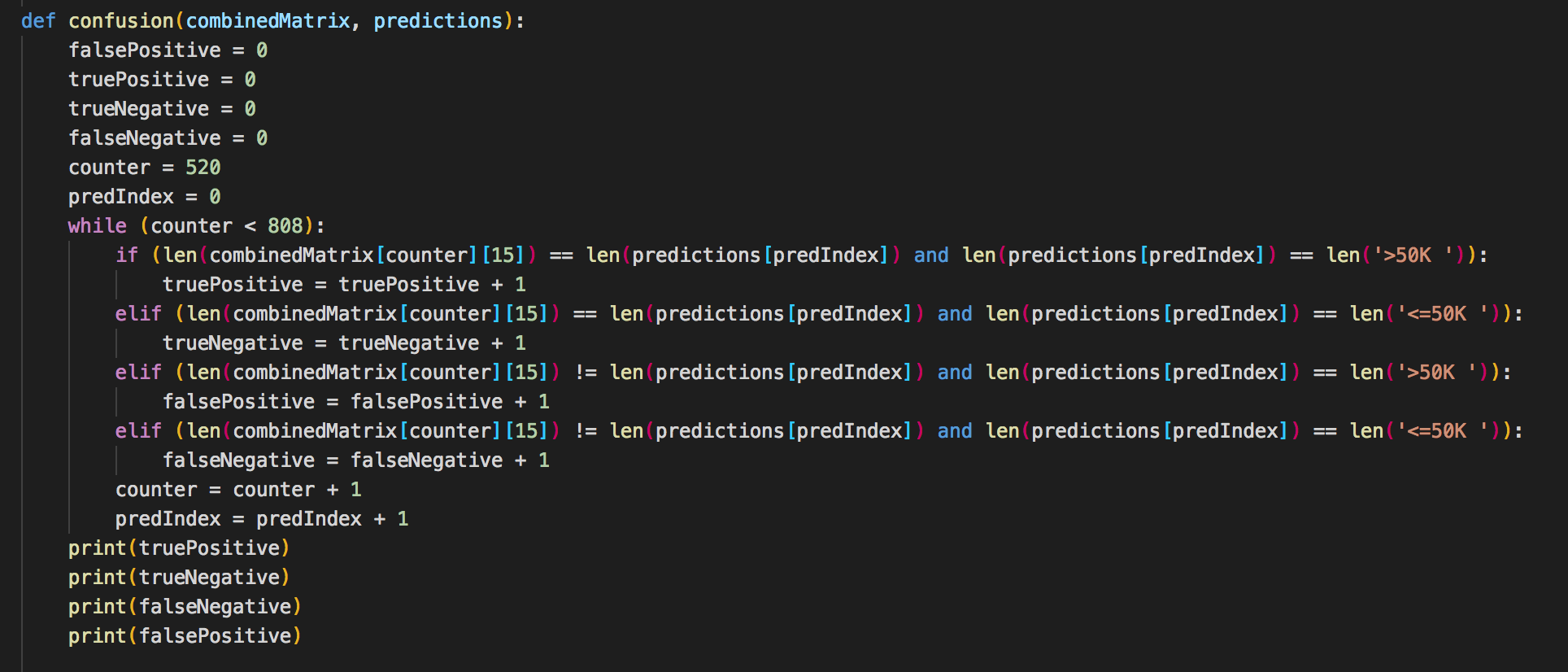
Our Python program generated an easy to read CSV file that allowed us to analysis our final results. From these results we were able to also develop some specific charts and other visuals to help further our understanding of these two data sets and our classification algorithm. This section will outline each one of these additional analytic items and provide additional insight into how our algorithm classified the test data set.

**Initial Result Observations**

Ultimately our assumptions we made about our kNN algorithm may have had an impact on the result. However, when running our program with a k value of 35 we correctly predicted 69.4% of all classifications. So, while our assumption may have provided us with less values to look at then what is typical the result was still relatively good. Also, our posterior probability calculation at k = 35 came out to an average of around 79%, with a minimum of 0.4905 and a maximum of 0.9976. which also is around what we expected. Particularly the minimum was encouraging since our selection method would always select the class that was most prevalent in the neighbors class, so a posterior probability lower than around 50% would not have made sense. Ultimately, these initial results were encouraging and convinced us our program exhibited success that could be evaluated even further after more analytics.

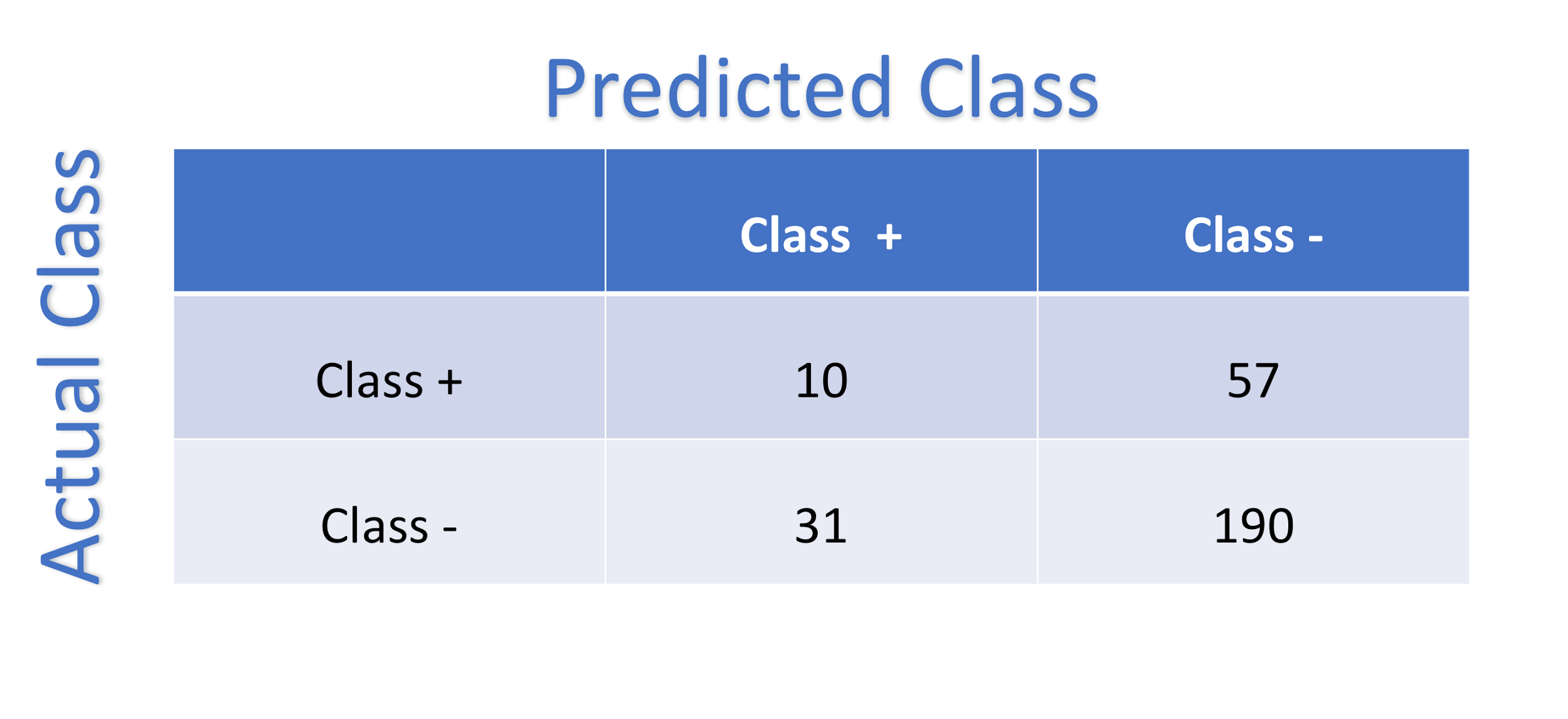
**Confusion and KNN Analysis**

After manipulating the output file a little to gain initial insight into how our results turned out, we also constructed more figures in order to get a visual of how our algorithm performed. To do that we created a function in our python codebase that took our combined matrix of both the training and test data set, and then our array of predicted classifications for each element. We then evaluated the prediction and actual “class” attribute for each instance of the test set to calculate the True Positive, True Negative, False Positive, and False Negative values.

****

***Figure 3***

This evaluation gave us the confusion matrix in the following figure. You can see in the code that the confusion matrix was built so that the “>50K” class was counted as the “+” class and “<=50K” was the “-“ class.

****

***Figure 4***

While it looks like from this figure our algorithm is way better at detecting true negatives than true positives, in reality there was just a significantly larger number of instances that fell into the “<=50K” class than the “>50K” class. So, it made sense to us that we would have more success when predicting something belonged in the “<=50K” because there is a greater possibility we would be right off of chance alone. In fact, 221 of the 288 test set instances belonged in this class, so the trend of our results being worse at correctly predicting the true positives (ten out of fifty-seven) fits what we would expect. The effect of this uneven distribution on our results can be shown through the drastic differences between our TP Rate and Our TN Rate. The TP Rate came out to 0.149, while the TN Rate was 0.860.

TN Rate = TN/(TN + FP)

TP Rate = TP/(TP + FN)

***Figure 5***

We also analyzed our algorithm by testing multiple sizes of values for k to see how having a larger subset of neighbors would impact our results. We tested k values of 11, 35, and 69 and there was a steady increase in the total number of correct classifications as the k value increased, but not as significantly as we had expected there to be considering it would have made sense for a larger subset of neighbors to help make the classification more certain. What we concluded might have caused there to not be as large of an increase was that there were no weights between the neighbors. Since the closest neighbor held the same impact as the 69th neighbor, the gain in accuracy we would get from including more of them would have been offset by the entries farthest away having the same leverage as the closest neighbor even though it has less in common with the test record. Therefore we concluded that having a large value for k would be much more useful for a weighted KNN algorithm compared to our unweighted one.

**Calculating Algorithm Error and Cost Effective Metrics**

Now in terms of an actual cost-effective algorithm, we wanted to look at our error as a whole, our precision, recall, and f-measure statistics. Before even performing the calculations, we realized there would be some information left out if we only used the confusion matrix from Figure 3. While counting one classification as the “+” class and the other as the “-“ class worked fine for general numbers about the accuracy, if we wanted the whole picture for the F-measure calculation we knew we would need an inverted Confusion Matrix where the class “<=50K” was counted as positive. By having two different confusion matrices we can see the difference in precision and recall rate when our algorithm was trying to correctly classify records that belonged to the majority class and the minority.

***Figure 6***

To get insight into the precision and accuracy of our algorithm to classify both classes we created a second confusion matrix with the same k value of 35 as the first so we could compare. The following proportions for precision, recall, and F-measure in order for the first confusion matrix, where the “+” class had a smaller number of values, was 0.2439, 0.1493, and 0.1852. These calculations for the second matrix was 0.7692, 0.8597, and 0.8119. These numbers reinforce the discovered pattern that it was much easier to be precise and have a better recall rate when you use KNN to classify entries when the class you are predicting it to be is a large majority of the training set.

**Section 3 – Comparing Our Classifier**

For our off the shelf implementation we decided to use a guide developed by Jason Brownlee on the scikit-learn Python library. This library would have been relatively complex to use with our dataset so after we initially attempted to implement it with our records but ran into complications we decided to instead use the guide as a basis to understand how other people use scikit-learn library to analyze their data. This allowed us to also learn about what quality values are for the proximity measurements and true/false positives.

**Our Algorithm vs. Brownlee’s Algorithm**

One of the largest differences between the two algorithms is how the training and test data were created. For Jason’s program he actually uses a specific ratio to split up his original data set into two new sets randomly. He uses a 67/33 (training/test) ratio which he claims is relatively standard when it comes to classification algorithms. For our algorithm we had a 520/289. Which gives slightly more weight to the training set compared to Jason’s ratio. We also were provided our data at separate times and are unsure the origin of either set.

Jason uses strictly Euclidean distance to find similarities in his data. This wouldn't really work for our set because we have many different data types within our sets. Also because we use other forms of similarity, we were able to place more weight on different attributes to give more clear analysis. We get the neighbors for given data points almost identically by simply using our similarity values to find the closest neighbors sets. Jason only using euclidean distances simply found the lowest distances values for each, while we had to use our proximity values that were calculated using a multitude of similarity calculations.

Once he had the neighbors calculated out, Brownlee was then able to simply use the scikit-learn components to then produce his results. The scikit kNN classifier is just a public library so we aren’t totally sure how it might be calculated. Although Brownlee uses very similar processes to us to get the data prepared for kNN calculations. We assume that the library uses Bayes Theorem which should produce values around the same range as ours.

**Our Results vs. Brownlee’s Results**

Brownlee was able to calculate his accuracy given the data he used the end of his program with the scikit-learn library. He found that his program actually found a 98% prediction accuracy. We had an average posterior probability of around 79%. So it appears that the combination of his data set and his program produced better results. Although it should be noted that his dataset was definitely smaller and much less complicated. Brownlee’s dataset only used simple integers which made the use of the Euclidean distance simplify the final kNN calculations.

So the difference in algorithms did seem to produce relatively different results. Overall we both performed pretty well but Brownlee’s was nearly perfect. This isn’t totally based on the design of our algorithm but really how our data was laid out and the manner it was provided to us. It also should be noted that this comparison could’ve potentially been more valuable had we figured out how to properly use the library to compare the results on the same data. Unfortunately because our data was so complicated it prevented us from having enough time to thoroughly make the scikit-learn library work. We still believe the above analysis is worthwhile because it allowed us to better understand the differences in how proximity measurements as well as the original data set can truly affect the results of the kNN classifier algorithm.

**Conclusion**

Overall our team was relatively pleased with the results that we found. We had many issues with simply getting the approximations (Homework #1) to be properly formatted and usable for our kNN classifier. Without these issues we could’ve possibly produced better results as well as more well thought out analysis. That being said, our program still does a relatively good job predicting the class given the data has so many different variables. We felt that our program from Homework #1 provided a good foundation for similarity approximations and our results give credence to the fact that it was well designed. The combination of the both our old code and new code fulfilled the requirements laid out by Homework #2 and we felt that our analysis proves that fact.