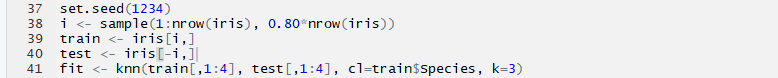
kNN Classicfication

For the first project of the semester, I was tasked with writing an implementation of kNN classification in c++. This algorithm works by finding the k nearest neighbors of some new point and classifying that point according to the classes of its neighbors. In my R implementation of the algorithm, I split the data between train and test data sets, and used the knn() command to run the algorithm on my test data.

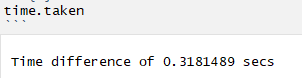
The algorithm classified 29 out of 30 observations correctly, an accuracy of 96.67%. The list of predicted classes is given below.

After some data exploration, I found that the 17th element was misclassified as virginica when it ought to have been classified as versicolor.

To capture the execution time of the R implementation, I made use of the sys.time() function to capture the time at the beginning of the program and at the end of the program. Subtracting the two yields the time in seconds of execution.



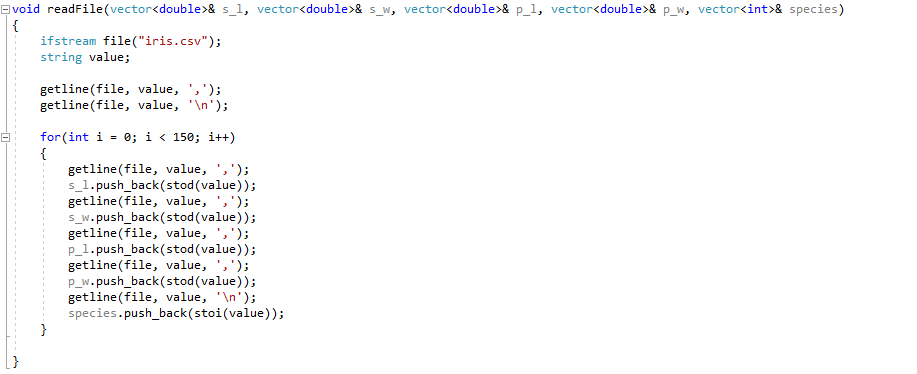




The average time of execution over 10 executions was around 0.288 seconds.

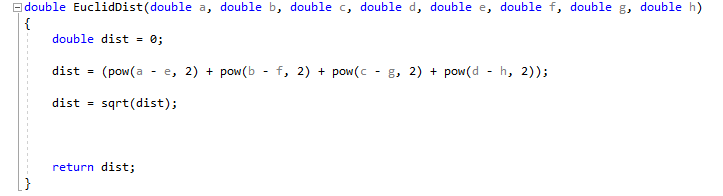
For the c++ implementation of this algorithm, I began by changing the value of class in the iris.csv file. Instead of working with strings, I changed all setosa classes to a 0, versicolor to a 1, and virginica to a 2. I did this to simplify some of the data analysis in my code.

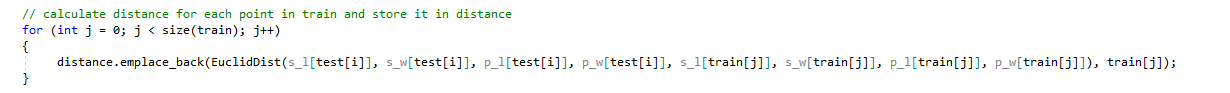
Next I read in the csv file and stored the columns in five different vectors. This was a simple enough process since the csv features are delimited by ‘,’. I passed those vectors into a kNN function which does most of the processing work for the algorithm.



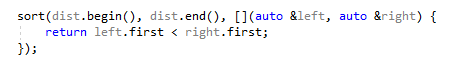
In the kNN function, I needed to separate my train and test data sets. I achieved this by looking up the indices of each in R, and making an array *train* and *test* which contained the indices of their respective data.

With the structures in place to separate the data, I needed to implement the actual algorithm. The first thing to do was to find the Euclidean distance from a test point to each train point. To achieve this, I set up a nested loop structure that stores the distance of a test point to each train point and stores that data in a vector called distance. This vector contains a pair: the distance from the test point and the index of the row which corresponds to the point. The distance was calculated using the standard formula for Euclidean distance, using the 4 features of the train and test point as predictors.

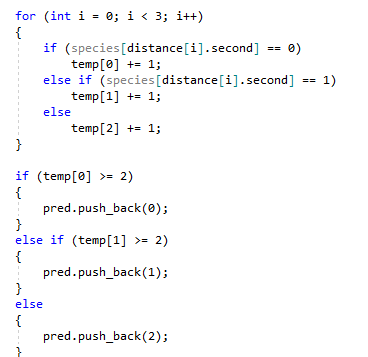




At this point we have a vector of length 120 that contains both the distance from the target test point, and the index of the row that corresponds to the point. We need to find the rows with the smallest distance. To do this, I sorted the vector by distance. The sorting algorithm I used is based on lambda calculus which I had written previously while learning about functional programming in another course.

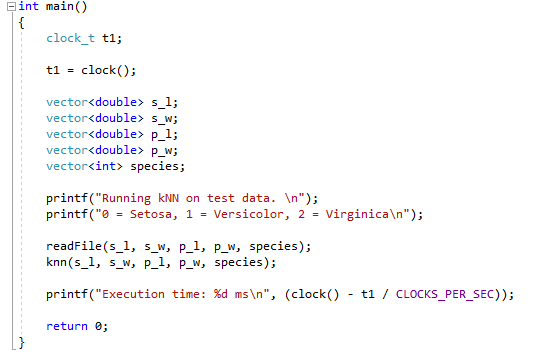


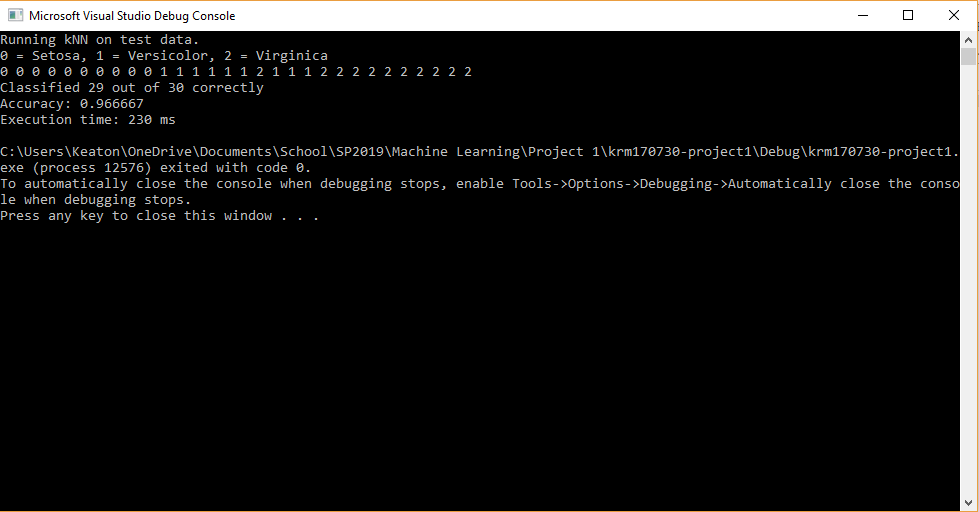
With the vector now sorted, I can check the type of the first three entries in *distance* (we use the first three because we are using k = 3). I use the 2nd value in each of the first three entries in *distance* to access the associated row of the *species* vector which contains the class variables. Depending on the class (0, 1, or 2) I incremented a corresponding index in an array *temp*. Then, depending on which index had two or more, I pushed a class to the *pred* vector which contains a final list of our predictions. As an example, if a test point has 2 neighbors with class 0 and one with class 1, the *temp* array would contain {2, 1, 0}. Then a 0 would be pushed to *pred*, indicating a prediction of setosa, because *temp* contains 2 or more setosa classes.



This looping structure continues until the end of the *test* array, indicating that we’ve made a prediction for each test point. To compute accuracy, I compared the classes stored in the *pred* vector with the corresponding indices in the *species* vector.

The final output gave the same predictions as our R implementation. You can see in the output that the 17th test point was incorrectly classified, just as in our R implementation. The run time of the program averaged 0.243 seconds over 10 executions, making it a bit faster on average than the R implementation. To measure execution time in my c++ program, I implemented the same technique conceptually as in the R code. That is, I took a time at the start at the program using the clock() function, and subtracted it from the time at the end of the program. To convert to seconds, I divided that value by a value clocks per second.





Ultimately my c++ implementation functions similarly to the R implementation for the given data. The output of the program was the same, and I feel confident that other train and test data sets would match the R implementation.