Task 1 – K Nearest Neighbours

2802 – INTELLLIGENT SYSTEMS

JESSY BARBER – S5138877

31/05/2021

Software Design

Initially the csv file is read into Spyder using the pandas module. This data is then split into three different species (Species1, Species2 and Species3), representing the three different species of Iris flowers. Scatter plots are then formulated to give a visual representation of each flower's sepal and petal characteristics. The driver code now takes a K range as input and a percentage in the form 0.X (e.g. 0.7). The program will run the training and testing data for this percentage split 3 times for each value of K, giving an average accuracy result for each run. The driver function will be explained in more depth later, but initially the split_data function is called with percentage as an argument.

Def split_data(percent):

The basic operation of the split_data function is to split the training and testing data as per the percentage parameter. Firstly, this function extracts all of the data columns and stores this in a variable called extract_data. The numpy random.permutation function is now called to randomly shuffle the order of the data. This is stored in a variable called random_data in which this data is then re-assigned to the extract_data variable. A training_size variable is initialized with the shape of the extract data multiplied by the parameter percentage. This assigns the size of the training data to the input percentage of the extracted data. This percentage of the data is now stored in a variable called training_df which represents the training data. The rest of the data is then stored in a variable called testing_df which represents the testing data. Variables train_data and test_data are now initialized with the values of training_df and testing_df. y_true and x_true are then initialized with the last column of the train_data and test_data which stores the corresponding flower type for each data point.

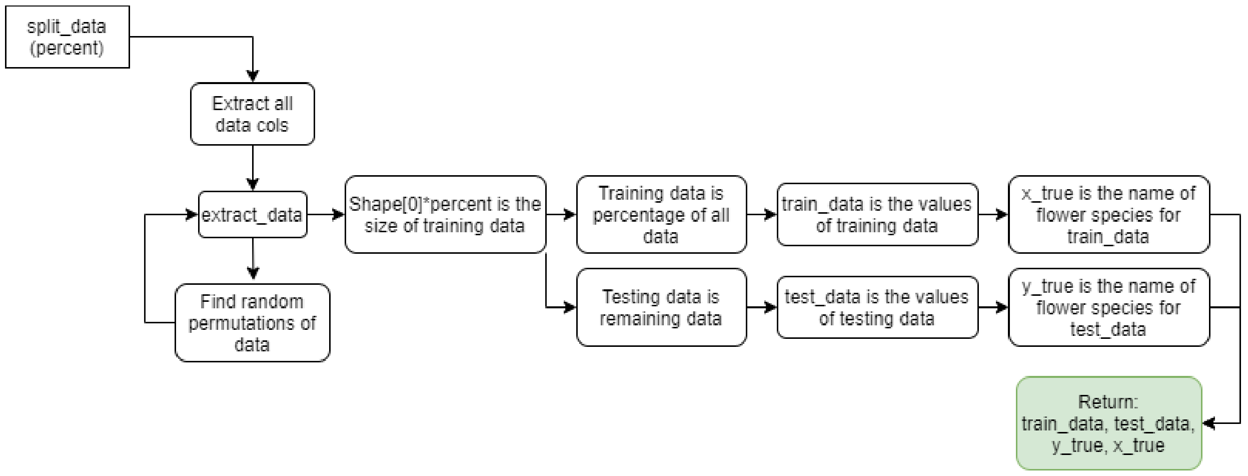
split_data (percent)
Extract all data cols
extract_data
Find random permutations of data
Shape[0]*percent is the size of training data
Training data is percentage of all data
Testing data is remaining data
train_data is the values of training data
test_data is the values of testing data
x_true is the name of flower species for train_data
y_true is the name of flower species for test_data
Return: train_data, test_data, y_true, x_true

Figure 1: Split_Data Software Design

Def knn(x_test, x_train, k, euc):

The knn function is called iteratively in the driver code for both the test and train data. The knn function is called with its current iteration, the train_data, value of K and a Boolean value called euc as arguments. The euc bool value indicates whether the Euclidean distance or Manhattan distance will be used. The knn function is called twice in each iteration with a True and False value, using both distance measuring metrics for each data point. The knn function initializes a list called flower_class to hold the results of calling the classification function. A variable called neighbours is used initially store the values of calling the classification function, which is iterated over and appended to the flower_class list. The flower_prediction variable stores the max value within the list, and this is returned.

Def classification(x_test, x_train, k, euc):

Within the knn function, the classification function is called with x_test, x_train, K and euc as arguments. Inside the classification function two lists called data_distance and data_values are initialized. If the Boolean euc is passed as True, the Euclidean distance function will be called iteratively for each value in x_train and the return value of this will be appended to the data_distance list. The current value of the iteration through x_train is appended to the data_values list. The same occurs if euc is false, except the Manhattan distance function will be called. Data_distance and data_values are re-assigned with the numpy array function. A variable called sort_indexes is created which stores an array of indices in the same shape of the data_distance array. The variable data_values is now re-assigned with the indexes of sort_indexes. The data values starting from the Kth element are returned.

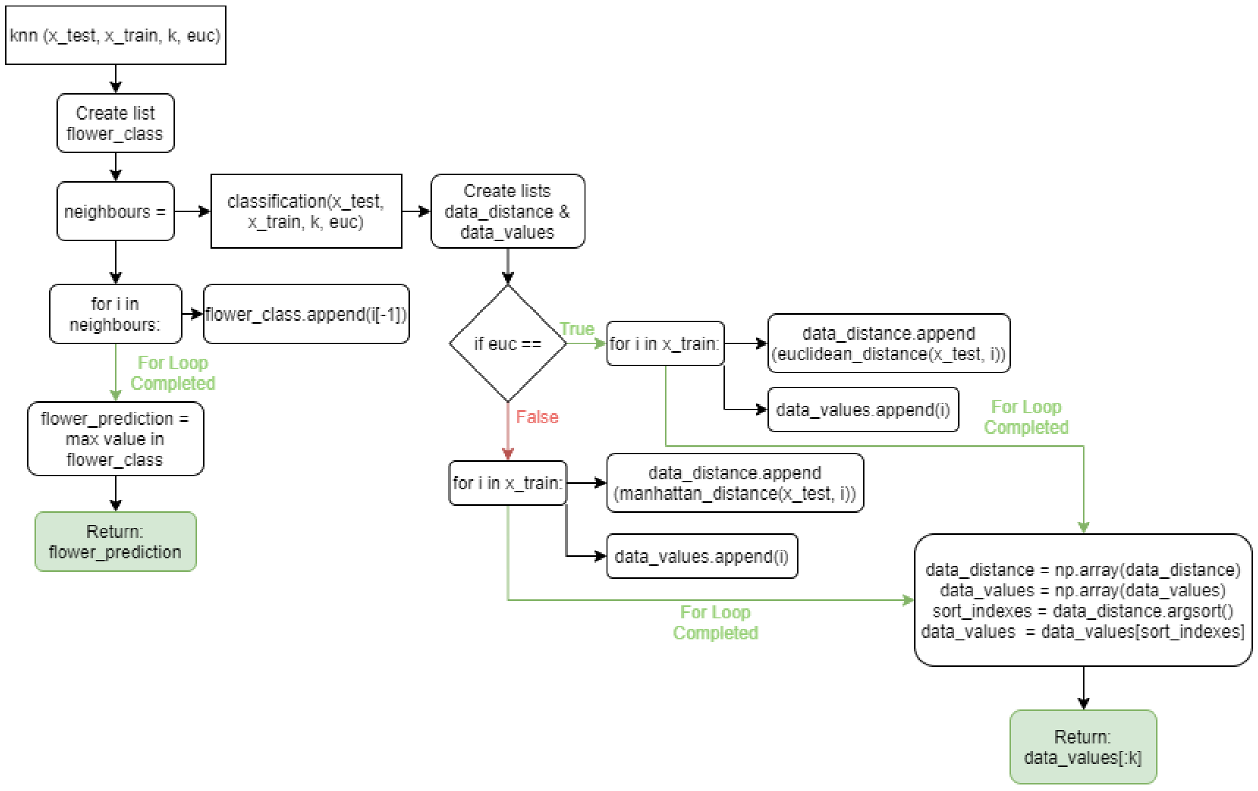
knn (x_test, x_train, k, euc)
Create list flower_class
neighbours =
classification(x_test, x_train, k, euc)
Create lists data_distance & data_values
for i in neighbours:
flower_class.append(i[-1])
For Loop Completed
flower_prediction = max value in flower_class
Return: flower_prediction
if euc ==
True
False
for i in x_train:
data_distance.append (euclidean_distance(x_test, i))
data_values.append(i)
For Loop Completed
for i in x_train:
data_distance.append (manhattan_distance(x_test, i))
data_values.append(i)
For Loop Completed
data_distance = np.array(data_distance)
data_values = np.array(data_values)
sort_indexes = data_distance.argsort()
data_values = data_values[sort_indexes]
Return: data_values[:k]

Figure 2: KNN and Classification Software Design

Def euclidean_distance(x_test, x_train):

The euclidean_distance function is called within the classification function and finds the Euclidean distance between data points in the test data and data points in the train data. Euclidean distance is defined as:

$$D_{Euclidean}(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \text{ (Wikipedia, 2021)}$$

Where p and q represent two points and the Euclidean distance is the square root of the summation of squared differences of subsequent points. A for loop iterates an index i from 0 to the length of the test data, summating the square differences between x_test and x_train points indexed by i. The square root of this value is returned by the function.

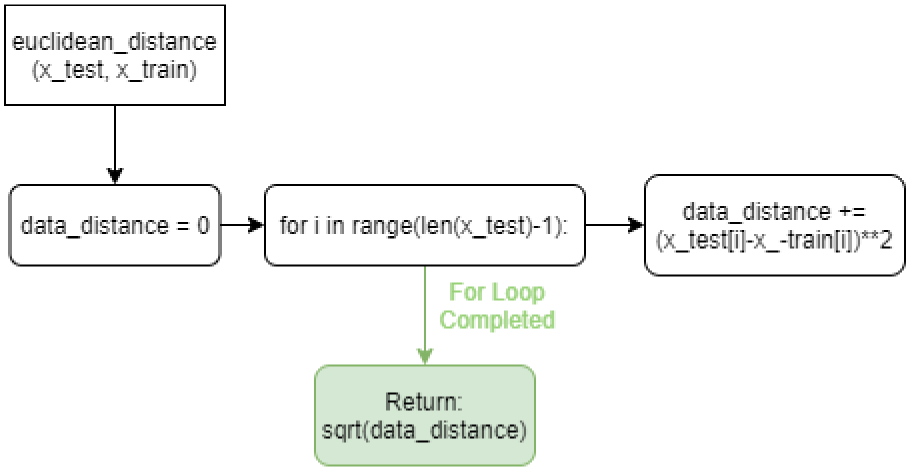
euclidean_distance (x_test, x_train)
data_distance = 0
for i in range(len(x_test)-1):
data_distance += (x_test[i]-x_-train[i])**2
For Loop Completed
Return: sqrt(data_distance)

Figure 3: Euclidean Distance Software Design

Def manhattan_distance(x_test, x_train):

The manhattan_distance function is quite similar to the euclidean_distance function. The difference is that it uses a slightly different metric of calculating the measurement between points. The Manhattan distance is defined as:

$$D_{Manhattan}(p, q) = \sum_{i=1}^{n} |p_i - q_i| \text{ (Wikipedia, 2021)}$$

The manhattan_distance function is similar to the euclidean_distance function in the way that it iterates the index i over the length of the test data. The data_distance is then defined as the summation of the absolute (magnitude) subtractions between subsequent x_test and x_train points indexed by i. This data_distance value is then returned.

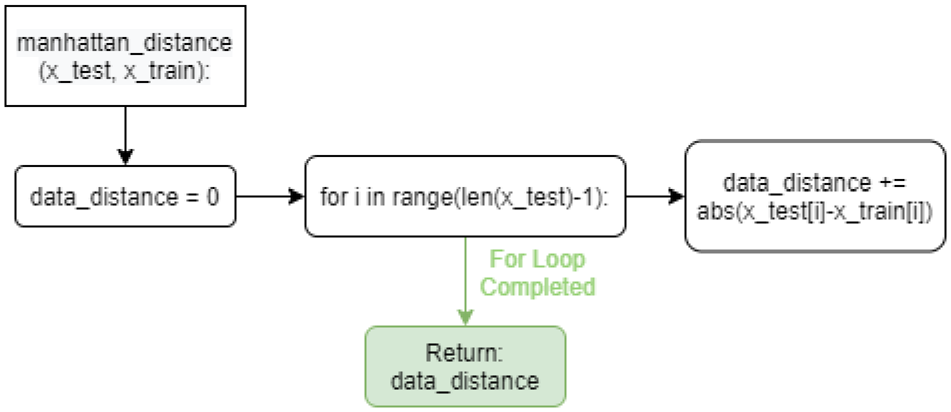
manhattan_distance (x_test, x_train):
data_distance = 0
for i in range(len(x_test)-1):
data_distance += abs(x_test[i]-x_train[i])
For Loop Completed
Return: data_distance

Figure 4: Manhattan Distance Software Design

Def accuracy(y_true, y_pred):

The last function is used to calculate the accuracy for each distance metric. As mentioned earlier, the split_data function returns a value of y_true and x_true. This is passed as the y_true parameter in the accuracy function and i iterates over the length of this list. In the driver code, 4 lists are created to represent the train and test data for the euclidean_distance and manhattan_distance functions, euc_x_pred, euc_y_pred, man_x_pred and man_y_pred respectively. These lists contain the predicted values returned by the knn and classification functions and are compared with the species names stored in the x_true and y_true lists. If the predicted and true values for the data points are the same, a count called correct_count is incremented. The accuracy is then defined as the count divided by the length of the true values and is returned by the function.

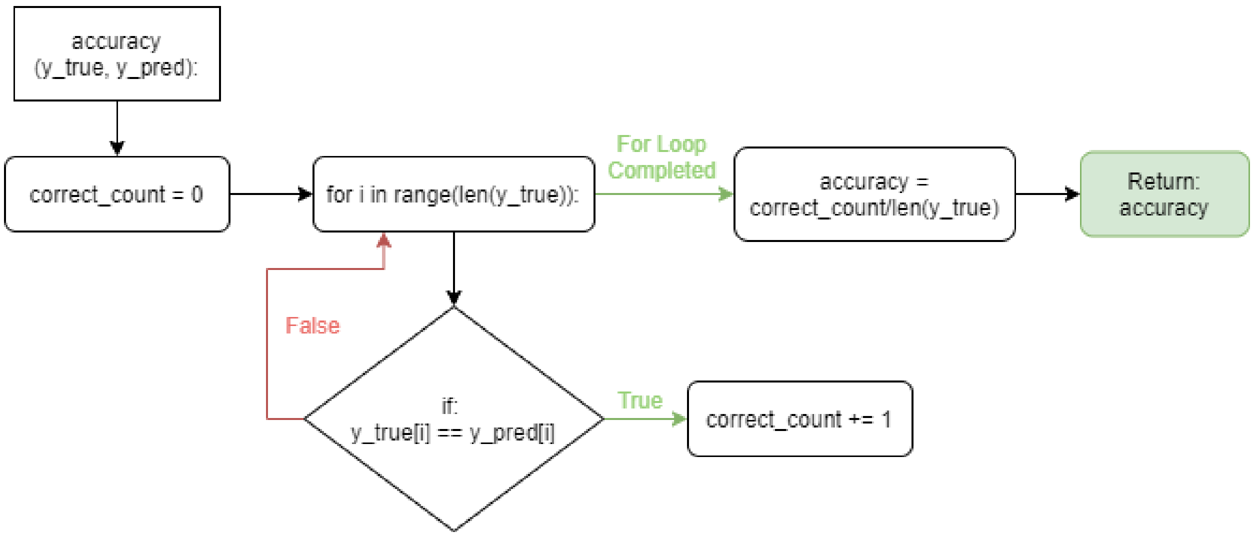
accuracy (y_true, y_pred):
correct_count = 0
for i in range(len(y_true)):
For Loop Completed
accuracy = correct_count/len(y_true)
Return: accuracy
False
if: y_true[i] == y_pred[i]
True
correct_count += 1

Figure 5: Accuracy Software Design

Driver Code:

As mentioned before, the driver code initially asks the user to input a range of k values (k_range) and a percentage (percent) to split the data. Hard coded is a value called run_times = 3 which determines the number of times the program is run for each value of k. This is to find the mean accuracy for each value of k over 3 runs. Lists are created to store these accuracy values which are defined under the run_times variable. A for loop now iterates over the range of 1 to k_range (skipping the k value of 0), with a nested for loop that iterates over the number of run times. The split data now called, meaning that for each 3 runs the data is randomly shuffled. Lists for the train and test data for each distance measurement are not initialized. Next is a for loop iterating over the test data, firstly setting the euc Boolean to True (finding the Euclidean distance) and appending the return value of the knn function to its corresponding storage (euc_y_pred), and then setting the euc Boolean to False (finding the Manhattan distance) and appending the return value of the knn function to its corresponding storage (man_y_pred). The same is repeated for the train data. The driver code will now start printing out the testing and training data as: "sample class = (true species), prediction class = (predicted species), prediction correct (True or False). The accuracy for each case is then printed and the mean of these three values are calculated and stored in test and train mean storages for both distance measurements. After the outer for loop has completed, the program will output two graphs showing the mean accuracy for the train and test data for each value of k in the specified range for Euclidean and Manhattan distance.

Results

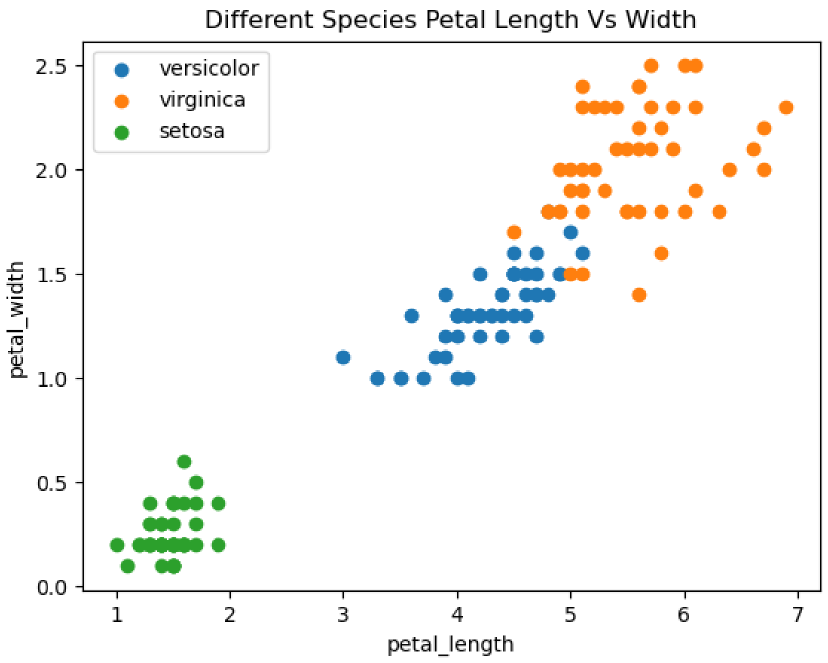

Figure 6: Species' Petal Characteristic Scatter Plot

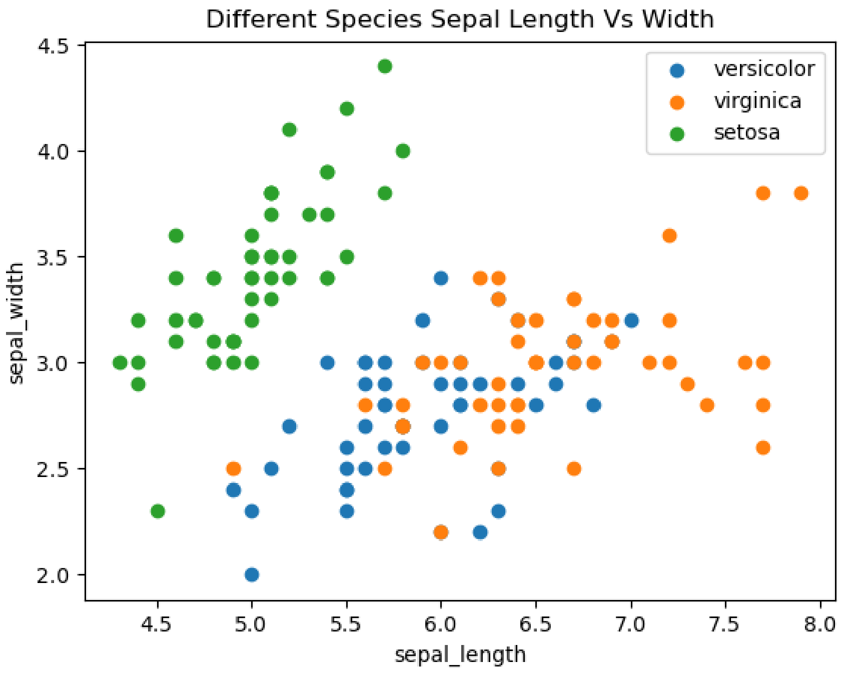

Figure 7: Species' Sepal Characteristic Scatter Plot

Displaying the Sepal and Petal characteristics of each species gives a great visual representation of the spread of the data. Visually it is clear that the most distinct characteristic between the species is their petal length and width.

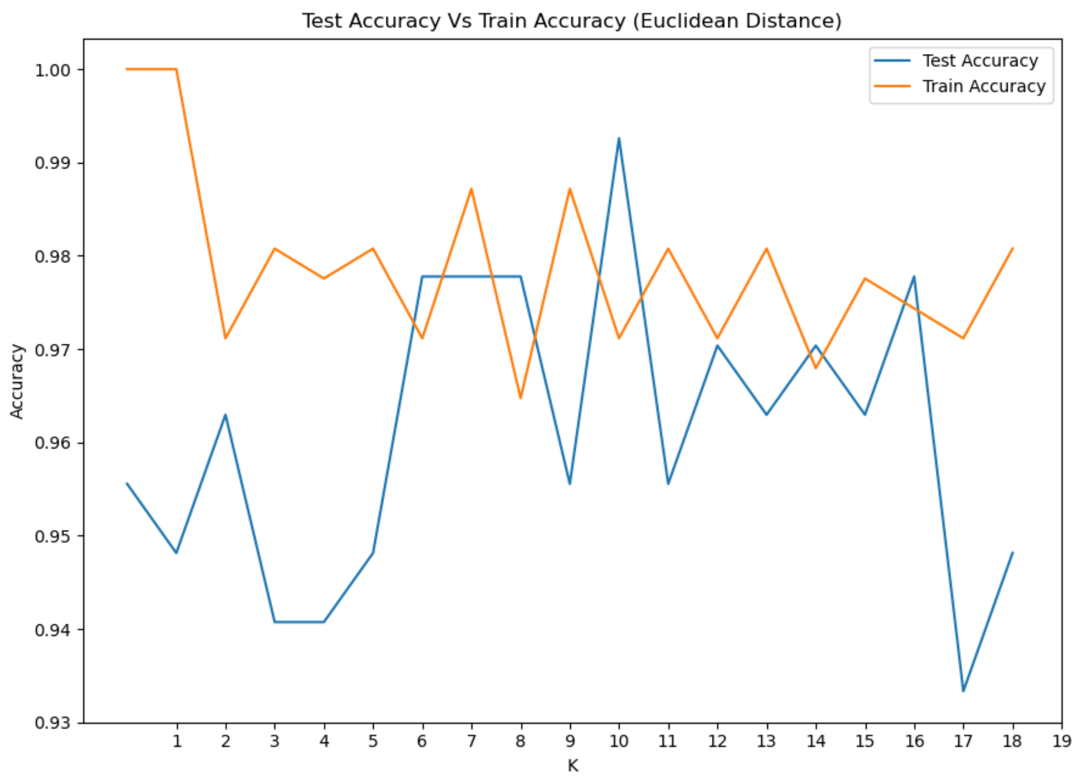

Figure 8: Test VS Train Accuracy Euclidean Distance

Mapping the testing and training accuracy for the Euclidean distance for a k range of 20, we can see how the value of k influences the accuracy of the data. Theoretically, an ideal value of k is usually around the square root of the test data, in this case for a 70% split data, an ideal value of k for the test data would be $\sqrt{105} = 10.25$. This is exactly what we see here as the test accuracy spikes to over 99% at a k value of 10 whilst the train accuracy drops to around 97%. It can be seen that the test data is overfitting between k values of 3 and 5, and dropy dramatically at a k value of 17. The train data remains quite constantly accuract, and the test data accuracy is greater then the train data in only a few spots such as the k value of 10 which is a good sign. Overall the Euclidean distance data is a pretty good combination of underfit and overfit for the test and train accuracy, and the learning behaves as expected with different values of k.

Mapping the testing and training accuracy for the Manhattan distance for a k range of 20 yield similar results to the euclidean distance. The Euclidean distance seems to give an overall better fit for the data sinnce the test accuracy looks to be slightly higher when using the Euclidean distance measurement. Nontheless, they are graphically similar and the Manhattan distance contains the expected behaviour of peak testing accuracy at a k value of 10. This peak results in an accuracy just under 99%, with a training accuracy of under 96%. Accuracy values when using the Manhattan are overall lower than the accuracy when using the Euclidean distance which immeditately suggests that the Euclidean distance measurement yields a more effective learning method. Since the accuracy when using the Manhattan distance is overall lower, the data is more underfit than the Euclidean distance results which supports the idea of Euclidean distance being the more effective method to use.

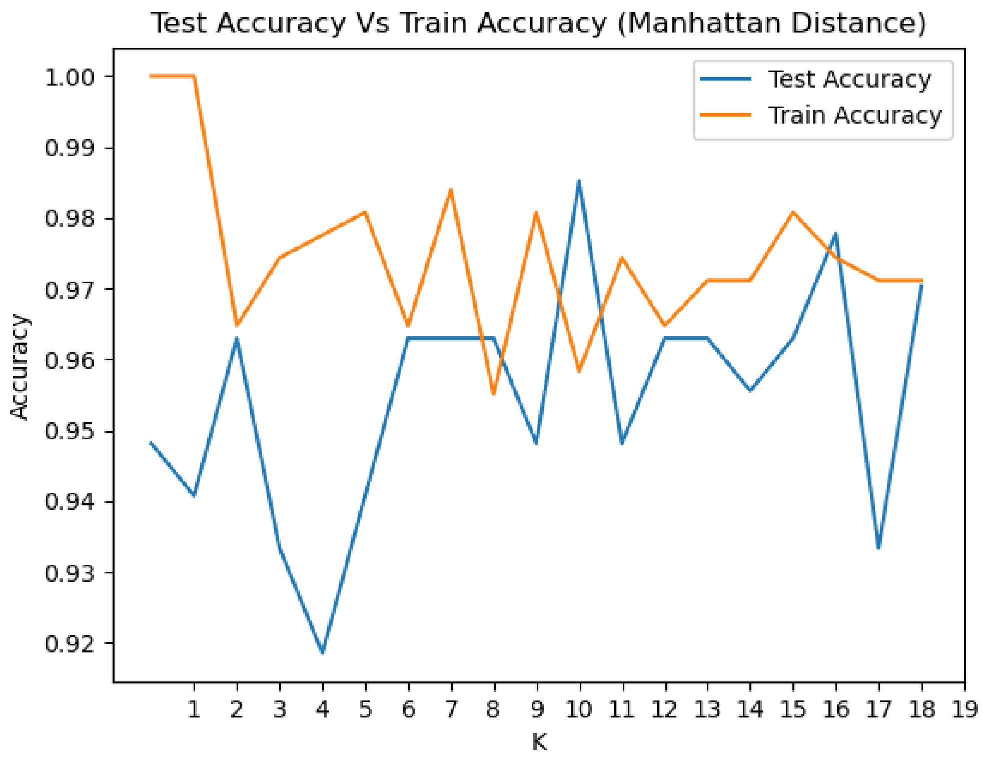

Figure 9: Test VS Train Accuracy Manhattan Distance

Conclusion

In conclusion, the Euclidean distance measurement is a better fit to facilitate the learning of the testing data. This is supported by the fact that the peak of the optimal k value of 10 is higher at over 99% when using the Euclidean distance in comparison to the peak of just under 99% when using the Manhattan distance. The lowest accuracy achieved with the Euclidean distance is at a k value of 17 where the accuracy drops to around 93.4% as opposed to the lowest point with the Manhattan distance, at a k value of 4, which drops to around 92%. The accuracy of the Euclidean distance model is on average higher than the Manhattan distance, meaning that the accuracy when using the Manhattan distance is more underfit than when using the Euclidean distance. The accuracy of the training data is also lower when using the Manhattan distance compared to the Euclidean distance, meaning that the Manhattan distance is also more prone to overfitting. Therefore, since the accuracy with the Manhattan distance is visually more prone to overfitting and underfitting than the accuracy with the Euclidean distance, the Euclidean distance is clearly the better model for achieving higher accuracy and effectively facilitates a very high accuracy of learning for the testing data at an optimal value of k = 10.

Bibliography

Wikipedia. (2021, April 13). *Euclidean Distance*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Euclidean_distance#:~:text=In%20mathematics%2C%20the%20Euclidean%20distance,being%20called%20the%20Pythagorean%20distance.

Wikipedia. (2021, February 17). *Taxicab Geometry*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Taxicab_geometry