CORK INSTITUTE OF TECHNOLOGY

MSC IN ARTIFICIAL INTELLIGENCE

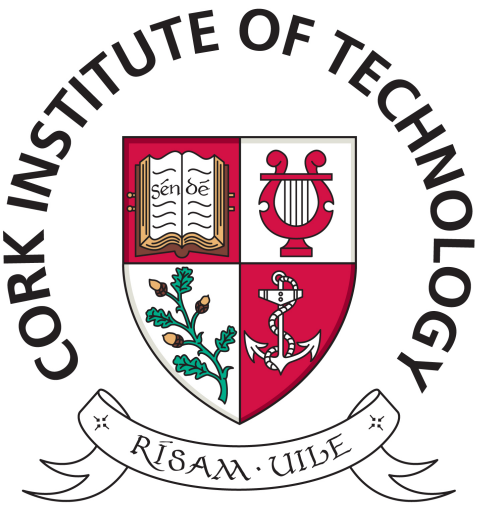
PRACTICAL MACHINE LEARNING

Assignment One

Author:
Jason Shawn D' SOUZA

Lecturer:
Dr Ted SCULLY

October 30, 2019



K-NN Algorithms

This project involved the building of our own *K- Nearest Neighbour*(**KNN**) Algorithm. It is a form of *superviesed* machine learning technique, wherein it learns from input feature data (training set) and gives a certain output on newer data (test data) based on its inferences from the training set.

Part-1

Part-1 involved us using Euclidean Distance Formula for a **test instance** with the training data. The method should be called *calculateDistances* as shown in the following snippet of code (Python Docstrings maybe added into the code later on and not be included in the report):

```
def calculateDistances(self, training_data, test_instance):
    # sqrt((x2-x1)**2+(y2-y1)**2)
    subtract = np.subtract(training_data, test_instance)
    square = np.square(subtract)
    euclid_sum = np.sum(square, axis=1)
    euclidean_distances = np.sqrt(euclid_sum)
    return euclidean_distances, np.argsort(euclidean_distances)
```

I have done it in each step, however due to the lightweight and flexible nature of python it can be written in one line if needed. We were also asked to calculate the accuracy in this part (obtained by correct predictions divided by the length of training instance, which would be multiplied by 100 to obtain the percentage)

```
def prediction(training_data, euclidean_indices, knn_k_value):
    # Get the data for the required k no. of indices and fetch the Class value/whether class 0,1,2
    k_instance_class = training_data[euclidean_indices[:knn_k_value]][:, -1]
    class_0_count = len(k_instance_class[k_instance_class == 0])
    class_1_count = len(k_instance_class[k_instance_class == 1])
    class_2_count = len(k_instance_class[k_instance_class == 2])
    # adding the equal in case of edge cases prefer the upper class
    if class_2_count >= class_1_count and class_2_count >= class_0_count:
        return 2
    if class_1_count >= class_0_count and class_1_count >= class_2_count:
        return 1
    if class_0_count >= class_2_count and class_0_count >= class_1_count:
        return 0
```

The calculations are carried out in the main method

```
D:\ProgramData\Anaconda3\python.exe D:/Programming/CIT/Labs/ML/Assignment1/Part_1.py
K : 1
Euclidean Correct : 895 Euclidean Incorrect : 105
Euclidean Correct: 89.5
Option chosen : 1

Process finished with exit code 0
```

Figure 1: Running with k=1 with the normal part-1 variant

Figure 1 shows the accuracy with k=1 in part-1 (option=1 and knn_value=k=1 in Part1.py).

Please note, the only duplication in code comes in the main methods (as I didnt define separate methods to call in my main for the various additions to the code, like the new distance metrics, varying k values etc. The core code is free of duplication

Part-2

In this part, we were asked to take a *distance-weighted* variant of the algorithm created in part-1. (We were asked to check for k=10 in part 1 and part2)

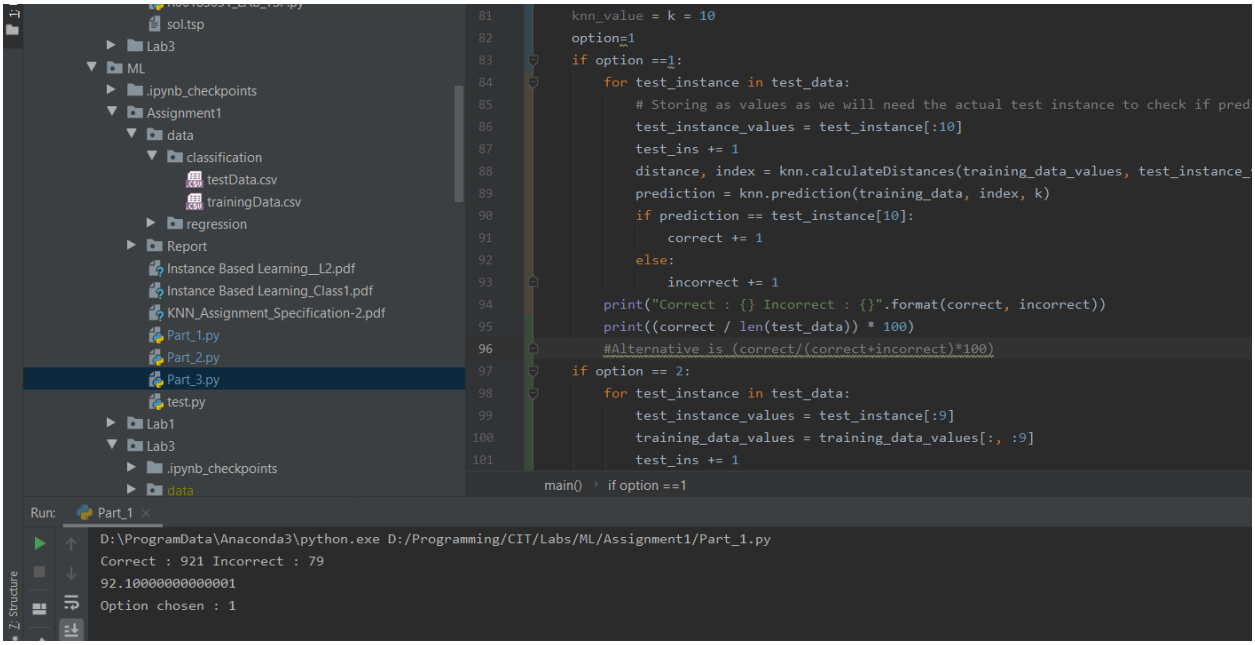


Figure 2: Running with k=10 with the normal part-1 variant

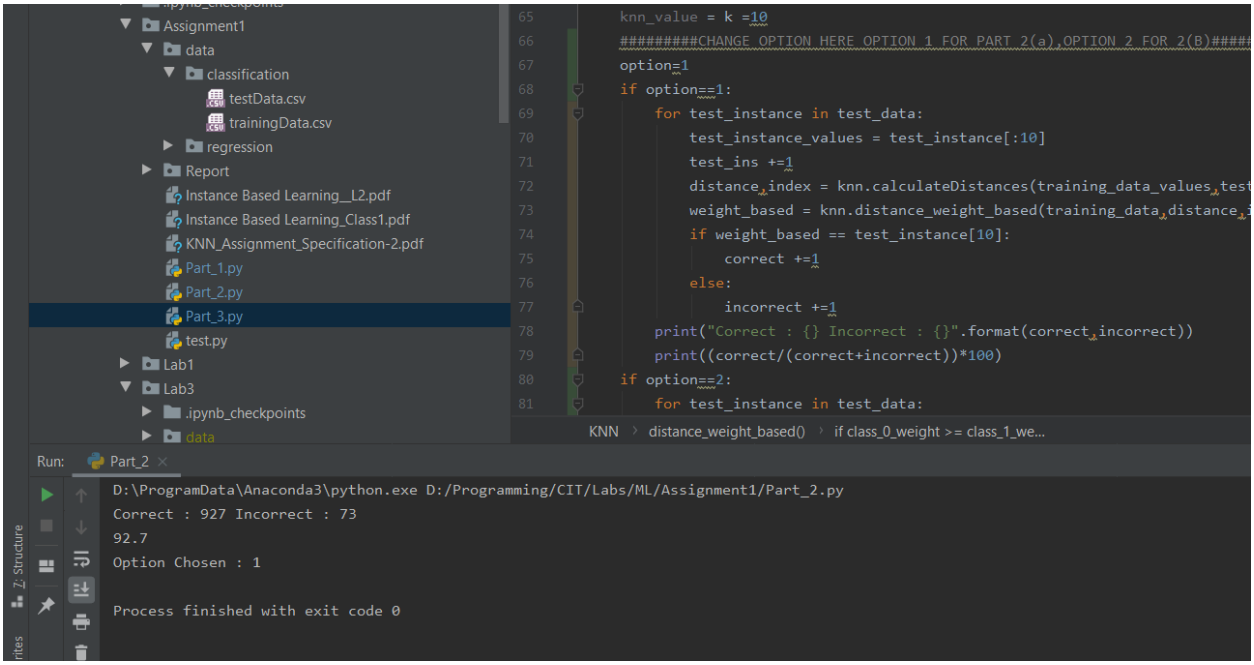


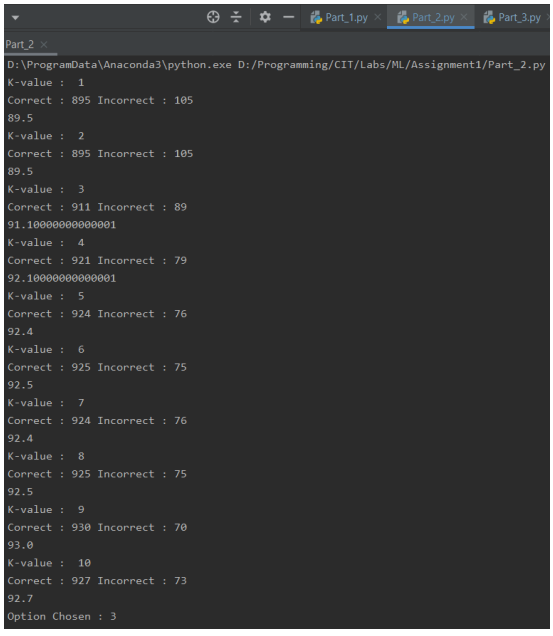
Figure 3: Running with k=10 with the distance-weighted variant

As can be compared from the two figures, the distance weighted variant reports a slightly larger accuracy **for the value of the hyper-parameter k =10**.

Testing varying K-values for Part-1 and Part-2

```
K Value : 1
Correct : 895 Incorrect : 105
89.5
K Value : 2
Correct : 889 Incorrect : 111
88.9
K Value : 3
Correct : 912 Incorrect : 88
91.2
K Value : 4
Correct : 912 Incorrect : 88
91.2
K Value : 5
Correct : 923 Incorrect : 77
92.30000000000001
K Value : 6
Correct : 921 Incorrect : 79
92.10000000000001
K Value : 7
Correct : 923 Incorrect : 77
92.30000000000001
K Value : 8
Correct : 921 Incorrect : 79
92.10000000000001
K Value : 9
Correct : 929 Incorrect : 71
92.9
K Value : 10
Correct : 921 Incorrect : 79
92.10000000000001
Option chosen : 3
```

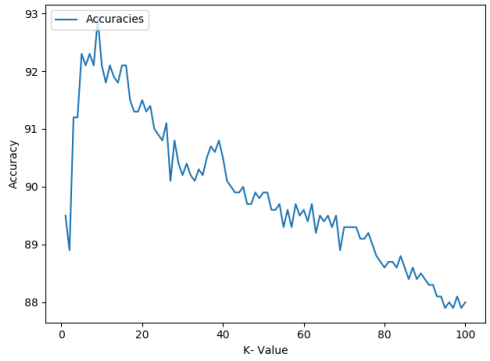
(a) Running with varying k values for part-1



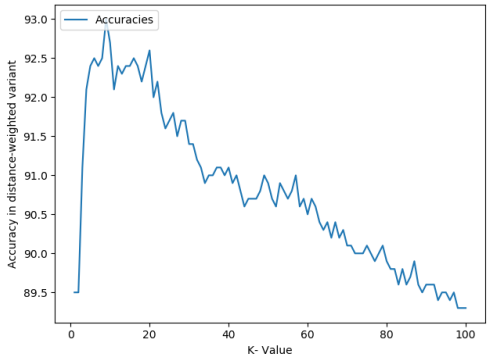
(b) Running with varying k values for part-2

The above subfigures are the outputs on running part-1 and part-2 (Figure 4a refers to the output of part-1 while Figure 4b refers to the output of part-2). It can be observed that the accuracies obtained in the case of the *distance-weighted* variant is higher than its counterpart in part-1 (normal euclidean distance). An **outlier** in this is the k value for 3, where the accuracy in Part-1 is 91.2 while its accuracy in part 2 is 91.1 (approx.). **To conclude,** in general the weighted-distance variant improves the accuracy of the K-NN algorithm.

On testing k-values from 1 to 100 and plotting the results on a graph (matplotlib was used along with option 3 in the main function for both part1 and part2.py), the following graphs are obtained



(a) Running with varying k values for part-1



(b) Running with varying k values for part-2

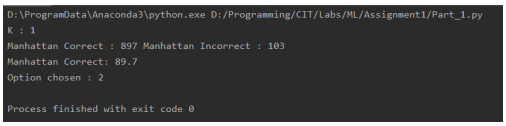
From Figures 5a and 5b it can be inferenced that the peak value of accuracy obtained is when the k value is in the range 0-20 (must be around 15).

Improving accuracies

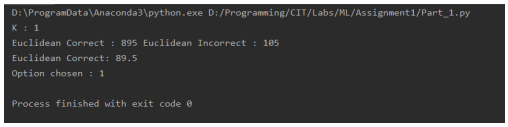
Possible techniques to improve accuracies involve a different distance metric (like Minkowski or Manhattan distance). **Hamming Distance** cannot be used as our dataset consist of *continuous valued features*,i.e, no categorical feature(Hamming distance deals with problems that have categorical features). Below is my implementation of the minkowski distance (as Manhattan formula is similar to euclidean (*difference being absolute value instead of the squares*)

```
 def calculateDistancesMinkowski(training_data, test_instance,minkowski_factor):
#     Factor = 1 or 2 1 for manhattan 2 for eulcidean
#   (|x-y|**a + ....)**1/a
minkowski_distance = np.power(np.sum(np.power(np.absolute(np.subtract(training_data, test_instance)),mi
    np.divide(1,minkowski_factor))
return minkowski_distance,np.argsort(minkowski_distance)
```

Using Manhattan Distance in Part-1



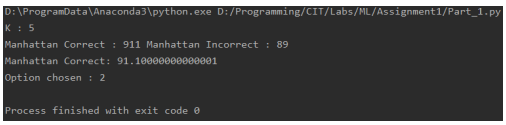
(a) Using Manhattan distance when k=1



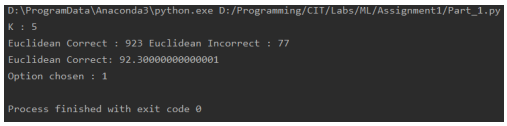
(b) Using Euclidean distance when k=1

As seen in the above figure, Manhattan distance for k=1 gives a higher value than its euclidean counterpart (Configs can be run by changing the option tag from 1 for euclidean to 2 for manhattan and manually changing k values. An optimal way to write the code would be to have a method for these configs and just call it, *due to time constraints, these haven't been implemented*.
For higher k values (For example:- k=5,k=10), euclidean performs better as shown in the figures (For k =10,

Euclidean accuracy was reported as **92.10000000000001** while manhattan reported **91.9** (**Screenshots will be provided in the zip but wont be included here to conserve space**)



(a) Using Manhattan distance when k=5



(b) Using Euclidean distance when k=5

Comparing distance metrics in Part-2

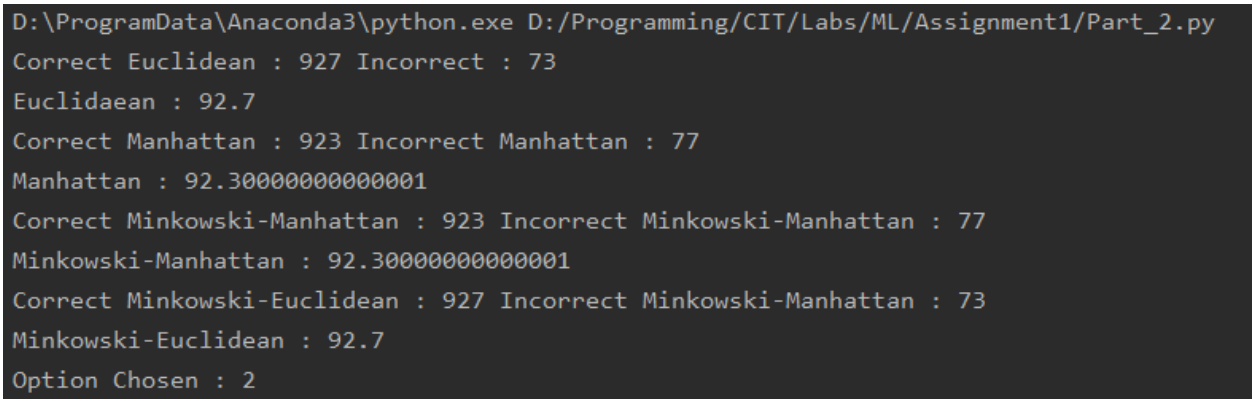


Figure 8: Running with k=10 with distance-weighted euclidean,minkowski(both euclidean and manhattan variants) and manhattan distance metrics

As seen in Figure 8, the Euclidean accuracy for k=10 is the ***same*** as the Minkowski-Euclidean variant. The same can be seen in the cases of the Manhattan and Minkowski-Manhattan variant

Conclusion on distance metrics

On comparing the results of the different distance metrics with the varying values of the hyper parameter 'K'. It can be seen that the overall accuracy does not see much of an increase (Small increase in accuracy when using Manhattan over Euclidean distance in Part-1 for K=1 as seen in Figure 6b

Part-3

For this section, we were asked to modify the distance weighted variant from part-2 and implement it for a regression problem in this task, along with an R^2 metric. , The code for this is included in the methods *distance_weight_based* and *r_square* and are heavily commented to explain how the python formula is obtained from the formula provided in both the slides and the specification.
A key difference between regression and classification, is the output(In the case of classification a class while in the case of regression it is numerical/continuous value)

On running the regression using euclidean distance and distance-weighted approach (with R^2 as an accuracy metric), the following accuracy is recorded (as mentioned in the slides the accuracy is in the range from 0 to 1 and greater the accuracy, greater is the model).

```
D:\ProgramData\Anaconda3\python.exe D:/Programming/CIT/Labs/ML/Assignment1/Part_3.py
K Value : 10
Accuracy 0.8506560730414083
Option chosen : 1

Process finished with exit code 0
```

Figure 9: Regression with k=10

Drawbacks of K-NN

As mentioned in the specification, each contribution of a feature is weighed equally.

Feature-Scaling

Assume a dataset has two features, Feature A (with data ranging from 1-10) and Feature B(with data ranging from 10-1000). This creates an imbalance as the distance (euclidean or any other metric) would reflect the gap between these two features. A solution to this problem as suggested in the lectures is data normalization(which I have implemented for task 3(b)). What data normalization allows us to have sort of an *even scale*. (For a given feature, data each value is normalized by the following formula :- $newValue = originalValue - minValue/maxValue - minValu$
where minValue is the minimum value in the feature range and maxValue the maximum value in the feature range. The accuracy will be mentioned in the later section when talking about the various techniques to tackle this drawback.

Techniques

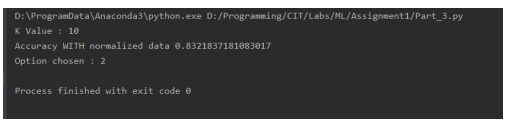
1. Data Normalization
2. KD-Trees (Helps in improving performance)
3. Principal Component Analysis (Type of dimensionality Reduction)

Data Normalization

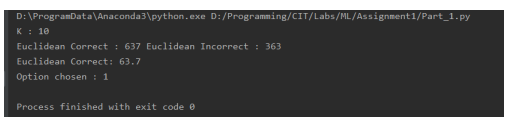
As mentioned earlier, I have implemented this technique

```
def normalizeData(self,data):
    values = data[:,:12] #exclude class column
    #Normalization from slide - newValue =original - minValue/maxVal-minVal
    numerator =np.subtract(values,np.min(values))
    denominator = np.subtract(np.max(values),np.min(values))
    newValue =np.divide(numerator,denominator)
    return newValue
```

On running this code with the regression dataset AND the classification dataset(part-1) with k=10 following is the output



(a) Normalized Data with regression (k=10)



(b) Euclidean distance with k=10

The results obtained can be seen in Figure 10a and Figure 10b. The difference in regression is smaller when compared to the difference in classification (Part-1 k=10 gives 92.10000000000001 percent while with normalization it gives 63.7 percent)

KD Trees

KD Trees are sort of binary trees that help in easing up computation of the KNN algorithm. This is achieved by picking **one** feature and splitting all data using the median value of that feature (as in slides). The subsequent feature is selected according to the median of the **first** feature and so on. This results in a Tree like structure (on the left all values less than the median and on the right all values greater than the median).
KD trees are only efficient when there are more instances than features in the dataset.(Just like in this projects dataset). The recommended value for n features in a dataset should be 2^n at the minimum (In this dataset we have 10 features and 1000 rows minimum (not counting the 4000 row training set). This value is greater than 2 to the power 10(1024), which means KD trees can be used.

Principal Component Analysis

This is a form of dimensionality reduction. As mentioned in the lectures, if a dataset with large number of features ($\geq$ 20) is used, a dimensionality reduction can be performed to reduce the number of features and use knn. Since our dataset has 10 features for classification and 12 for regression, there is no need to apply such techniques.
PCA is carried out in the following steps:-

1. Standardizing/Normalizing the dataset
2. Making a *covariance* matrix
3. Devovling the matrix into eigenvectors and its eigenvalues
4. Sorting the values in descending order and selecting the *k* largest values (K will be the new dimension of the feature space (for example, k=3 would result in a 3-dimensional feature space. k must be chose such that k $\leq$ d)
5. Making a new matrix with the chosen eigenvectors.
6. Matrix transformation of the original dataset using this new matrix (obtained in previous step) to generate the new feature sub-space

Distance - Weighted KNN

This technique also helps in increasing the accuracy of KNNS as shown in Part-2 of this assignment. This is because the class is assigned taking into consideration the distances of the points near it. One of the issues (as observed in the graphs above (Figures 5a and 5b) is the value of k (too small then more sensitive to outliers and if too largethen the neighborhood may include too many points from other classes.)

Conclusion

To conclude, KNNs are easy to undersand and implement (to a certain extent) and can handle multiclass/classification problems **as well as** regression problems when coupled with a moderate number of features and training instances