Comp219 Assignment 1 Document

## University ID: 201458436

# Check list of marking criteria:

F1 implemented successfully

F2 implemented successfully

F3 implemented successfully

F4 implemented successfully

F5 implemented successfully

# Summary

I am going to implement the K nearest neighbour algorithm on the digits dataset (Optical recognition of handwritten digits dataset). As I am using knn, I do not have a saved model. On my machine, it takes me 131.83121299743652 seconds to run my own knn algorithm, with a 98.5% level of accuracy in both training and testing. It also takes me 0.6284959316253662 seconds to run the scikit knn algorithm (first file of the submission files), with a 98.74739039665971% level of accuracy of the training data and a 99.16666666666667% level of accuracy of the testing data.

I am running all files on an Intel i7-7500U CPU@ 2.90GHZ, with 8GB of RAM on Windows10.

# Submission files

1. Comp219 Assignment 1 file 1 using scikit library functions.py
2. Comp219 Assignment 1 file 2 using own knn.py
3. Comp219 Assignment 1 Document.docx

The first file will be my attempt at implementing the F1, F2, F4 and F5 requirements. The second file is my attempt at implementing the F3, F4 and F5 requirements.

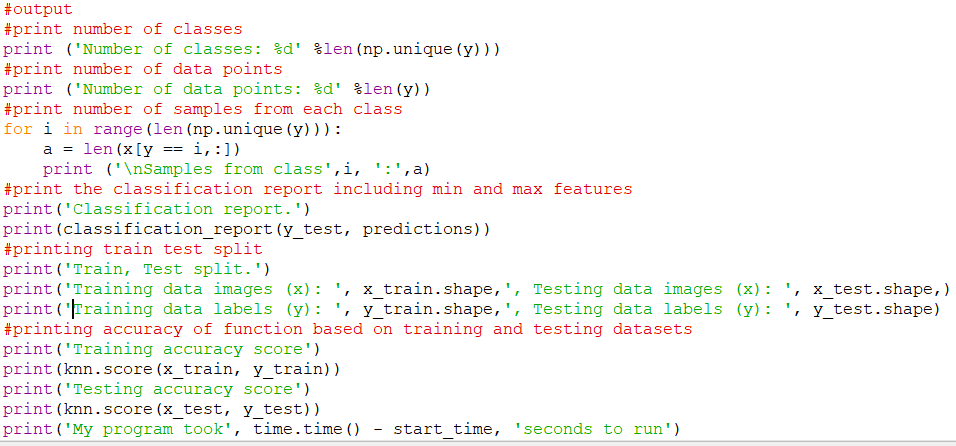
# Software dependencies

In order to run both programs, you will need to have the most up to date version of Python3.8 installed. Similarly, you will need to install scikit-learn, NumPy and Matplotlib.

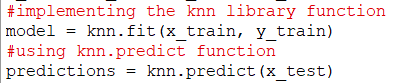
# How to run Comp219 Assignment 1 file 1

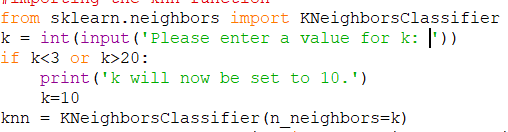
In order to run the first file, simply press f5 on your keyboard after opening the program in idle. This will load the digits dataset straight into the program and train our knn using the scikit KNeighborsClassifier function. When you run the program, it will display the number of classes of the digits dataset, the number of data points, the number of samples from each class, and the classification report based on the testing data labels and the predictions of the knn algorithm. Similarly, the program will display the train/test split of the data, the accuracy of the knn algorithm on our training data, the accuracy of our knn algorithm on the testing data and how long the program took to run.

# Implementation of the functional and additional requirements of Comp219 Assignment 1 file 1

F1 is met in this document via the output. This can be seen from this screenshot.

­­ The first two print statements are self-explanatory, however the number of samples for each class is calculated by counting the number of samples in a for loop, until we have no more classes left. The classification report displays the minimum and maximum values of each feature, as well as some other additional but still useful information such as the precision and recall. The train dataset and test dataset split, is calculated much earlier, when training the knn algorithm using the train\_test\_split function of the scikit library. Notice, we do not manually shuffle the x and y values the same way that we do in file 2. Here we add a random state via the train\_test\_split function.



F2 is met when we import the KNeighborsClassifier from the sklearn library. We then use it to train and test our algorithm. I successfully met F4 by outputting the train and test accuracy of the knn algorithm. I also implemented the F5 requirement by allowing the user to query the number of neighbours which we use. If the user selects a number of neighbours, which is outside of the available scope, k is set to 10 and this is displayed to the user.

# How to run Comp219 Assignment 1 file 2

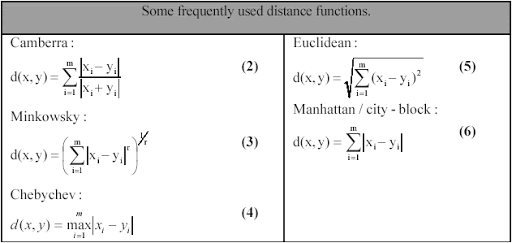
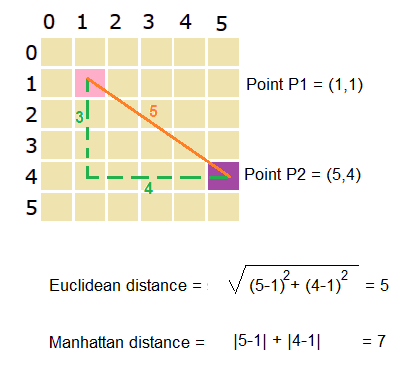
In order to run the second file, simply press f5 on your keyboard after opening the program in idle. This will load the digits dataset straight into the program and train our own knn algorithm using the predictClassification function, the getNeighbors function and the Euclidean function. When you run the program, it will display the accuracy of the knn algorithm on the training set, the accuracy of the knn algorithm on the test set and how long it takes for the program to run in seconds. There is clear commenting throughout the program, explaining each function. In summary, in order to correctly classify each digit, we need to calculate the nearest neighbours, which we can only do if we calculate the Euclidean distance between the test data point and all the training data.

# The implementation of knn

First, we need to compute the Euclidean distance between the test data point and all the training data. Then, we need to sort the calculated distances. Then, we need to take the k nearest neighbours. This can be done by taking the top k rows from the sorted array and finding the majority class of these rows, in order to predict where to place the test data. Then we return the predicted class and repeat this process for all of our training data. We then test the accuracy of our knn algorithm on the training data and the testing data. We should also implement a timer in order to see how long our program takes to run.

# Implementation of the functional and additional requirements of Comp219 Assignment 1 file 2

First, we calculate the Euclidean distance by finding the difference of the two squared values and then square rooting the distance. This is similar to the Manhattan distance, which takes the difference of two points in modulus. The Manhattan distance is less accurate for our scenario. The Euclidean distance is also similar to the minkowski distance, however the Minkowski distance is less accurate for our sample of data. I also take the absolute value of row1 and row2, even though the Euclidean distance already calculates the absolute value however, when testing, I found that taking the absolute value made the program run faster.



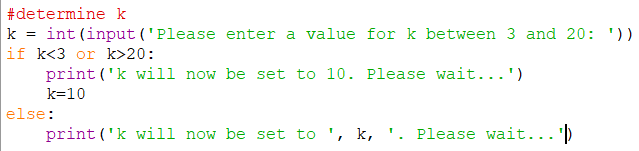
In order to find the k nearest neighbours, we need to first sort our data. We can sort our data in ascending order by using the argsort function, which belongs to the numpy library. Then we can arrange the data and split it acording to the number of neighbours.

The test data point will belong to whichever class has the majority votes, which is acomplished via the max function. The max function has 4 paramaters: iretable, \*iretables, key and default. The last 3 paramaters are optional however, we do use a key in our case. Our iretable is “Classes” and our key is the number of neighbours in each class. The max function passes the classes the number of neighbours in each class and compares them to the true value. The max function will then return a class which has the majority votes, making a prediction. The real MNIST dataset contains 70,000 handwritten digits labeled 0-9. These images are a collection of 28x28 gray scale pixels. The digits dataset which we use is much smaller, containing only 1797 samples of 8x8 images, making our job much easier and quicker.

We then need to process our data. This is similar to how we do it in file 1 however, this time, we set our target data to be an int8 variable, making sure it’s an integer. We find our independent and dependent variables; find the shape of our data. Throughout the program I try to use as many 1 liners as possible, in order to make the program as efficient as possible. We find that our x data is a collection of 1797 images which are 64 pixels. We find our y data to be a collection of 1797 labels.

We then manually shuffle our data. This is done in order to avoid overfitting and as mentioned previously, this is different to the way we do it in file 1, as we are not using the train\_test\_split function here. We split our dataset into 1000 training images and labels, as well as 50 testing variables and labels. By splitting our dataset in this way, we reduce our number of imports, as well as reducing our dataset directly. This allows for our program to be much more efficient. When testing, the difference between having a training set of 1000 images and 2000 images was only 0.5% accuracy, while taking almost four times as long. I then ran some tests in order to see whether we were propperly importing the dataset and were able to visualise the data, by using some matplot functions to convert our 64 columns into an 8x8 image. The output was digit 0, however this will constantly change each time we run our program, as we have shuffled our data.

I then tested the predict function and checked the last column against the true value. When found to be accurate, I knew that everything was correct up to this point if we ran through our program once.

Finally, we sort our true values and our prediction values, as well as our test values. I used 10 as the k number of neighbours, as when researching this, I found that 3 and 5 were the standard of nodes however, nobody really knows the ideal number of neighbours. Good syntax is to keep the number of neighbours to be the same as the number of classes (of a dataset). I allow the user to query the number of nodes, giving them the option to set the number of nodes between 3 and 20 however, if they set k to be outside this spectrum, I display a message saying that the number of nodes has been set to 10 and manually set the number of nodes to 10. This meets the requirement F5.

When first running the problem, I had realised that I had made an error somewhere because my accuracy was only around 10% and the program took very little time to run. This was because I had made an indentation error in my Euclidean function, meaning it wasn’t returning a Euclidean distance.

The last thing we need to do is display the accuracy of our knn algorithm on the training data and the testing data, as well as how long it took for the program to run.