Machine Learning Coursework, pt. I

Grzegorz Rybak

1. Implementation details.

1.1. Problem statement.

Given a dataset of 150 samples of 3 different types of iris plants, each with 4 features, the objective is to create a classifier that predicts the type of the given plant based on comparing its 4 features with the *n*-closest plants in terms of their features. To improve the accuracy of the prediction and establish the most efficient value of *n* – a nested cross-validation on the original data needs to be performed during the process of feeding the classifier with the test data.

1.2. IRIS dataset overview.

The Iris dataset includes 150 datapoints representing iris plants. Each datapoint (plant) has 4 features (namely: *sepal length*, *sepal width*, *petal length* and *petal width*) in form of float values ranging from 0.1 to 7.9 in total. Each plant can be of one of three possible types (*Iris-Setosa*, *Iris-Versicolour* or *Iris-Virginica*) which will be represented as "0", "1" or "2". There are 50 datapoints for each of the iris plants. The above will be stored in a variable **X** of shape "(150,4)" – 150 samples of 4-tuples – and variable **y** of shape "(150)", i.e. 150 values ranging from "0" to "2".

1.3. Codebase structure overview.

The codebase of this assignment consists of 2 main entities and 3 helper methods:

- **mykNN** class, performing the kNN classification. It contains 7 sub-functions:
    - **euclideanDistance**: returning the Euclidean distance of any two given points
    - **manhattanDistance**: returning the Manhattan distance of any two given points
    - **getNeighbours**: returning a list of indices of the nearest *n* neighbouring points from our training samples dataset to the given test datapoint.
    - **assignLabel**: assigning a predicted label ("0", "1", or "2") to the given test datapoint based on what were the labels of the *n* nearest neighbours (obtained from the *getNeighbours* function)
    - **__init__**: serving as the constructor function once the class is instantiated and performing the above functions in the following order: *euclideanDistance* (or *manhattanDistance*) -> *getNeighbours* -> *assignLabel* for each of the datapoints in the given test dataset.
    - **__repr__** and **__call__**: python's "dunder" functions returning the array with all the predicted labels of each datapoint in the given test dataset every time the *mykNN* class is called as: "*name_of_the_instance_of_mykNN()*" or as *"print(name_of_the_instance_of_mykNN)"*
- **myNestedCrossVal** function splitting the data into 5 folds, choosing the best parameters (number of neighbouring points to consider & type of distance measure) based on testing on validation data and performing the kNN classification with the best parameters on the test data.
- **myplotGrid**, **myAccuracy, myConfMat** helper methods that, successively, plot the visualisation of the dataset, calculate the overall accuracy of the implemented classifier and print confusion matrix of the results.

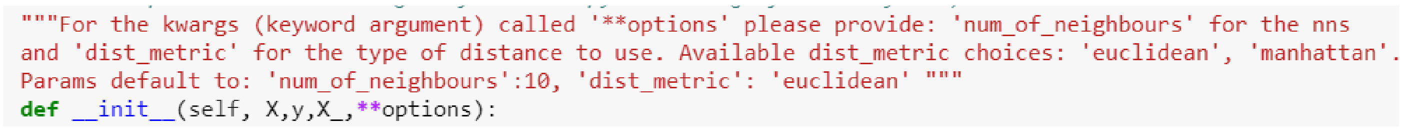
1.4. *mykNN* class.

The implementation of the *mykNN* class requires 3 positional arguments to work:

"X": array of *x* number of 4-sized arrays. "*y"*: the corresponding x number of the labels associated with the X.

X_: x number of 4-sized arrays to predict the label (also called "test datapoints").

Additionally, the class accepts 2 optional keyword arguments called "*num_of_neighbours*" and "*dist_metric*" which are implemented by adding the ***options* parameter (see figure [1] below).

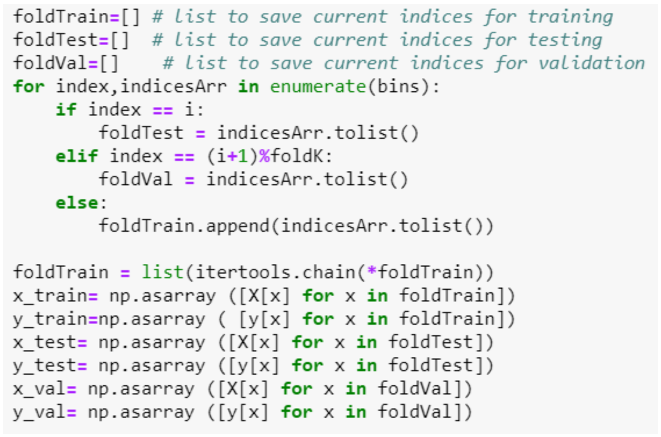


```
"""For the kwargs (keyword argument) called '**options' please provide: 'num_of_neighbours' for the nns
and 'dist_metric' for the type of distance to use. Available dist_metric choices: 'euclidean', 'manhattan'.
Params default to: 'num_of_neighbours':10, 'dist_metric': 'euclidean' """
def __init__(self, X,y,X_,**options):
```

Figure 1: mykNN class initialiser (constructor) definition

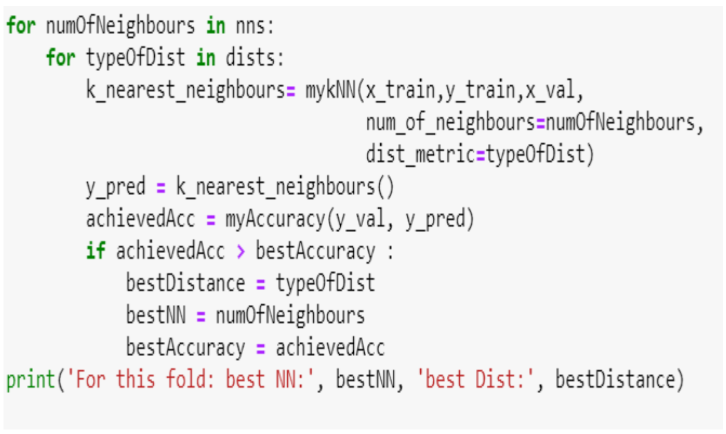
1.5. *myNestedCrossVal*.

This function starts by arranging the original data into *n* number of "folds"; in each fold the data is organised into 3 lists for training, validation and test datapoints. The data is always arranged differently in the lists for each fold. Then, the lists are used to create 6 lists containing the x and y arrays for training, validation and test using Python's built-in "itertools" library and Numpy's "toList()" (figure [2]) to avoid a redundant dimensionality (otherwise, the data would be stored in a shape: "(4,30,4)"). Next, for each fold the best performing "num_of_neighbours" and "dist_metrics" parameters are chosen using the validation data. This is done by running the *mykNN* classifier on each of the possible "nns" (number of neighbours) and "dists" (types of distance measure) and choosing the set of the parameters giving the best accuracy on the **validation data** (figure [3]) .



```
foldTrain=[] # list to save current indices for training
foldTest=[]  # list to save current indices for testing
foldVal=[]    # list to save current indices for validation
for index,indicesArr in enumerate(bins):
    if index == i:
        foldTest = indicesArr.tolist()
    elif index == (i+1)%foldK:
        foldVal = indicesArr.tolist()
    else:
        foldTrain.append(indicesArr.tolist())

foldTrain = list(itertools.chain(*foldTrain))
x_train= np.asarray ([X[x] for x in foldTrain])
y_train=np.asarray ( [y[x] for x in foldTrain])
x_test= np.asarray ([X[x] for x in foldTest])
y_test= np.asarray ([y[x] for x in foldTest])
x_val= np.asarray ([X[x] for x in foldVal])
y_val= np.asarray ([y[x] for x in foldVal])
```



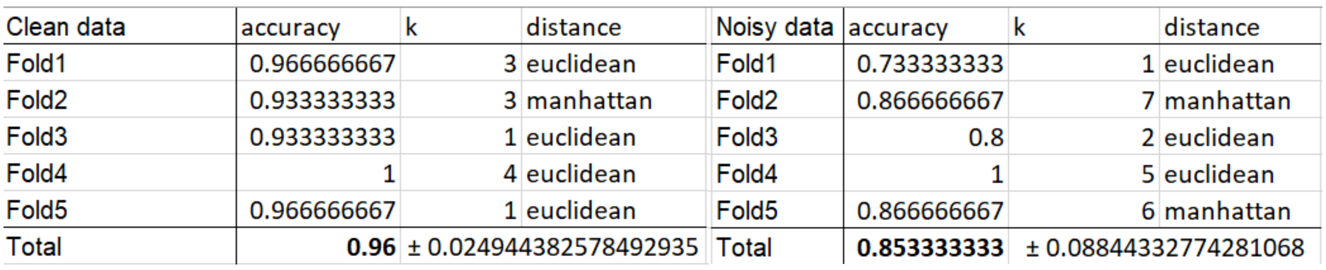
```
for numOfNeighbours in nns:
    for typeOfDist in dists:
        k_nearest_neighbours= mykNN(x_train,y_train,x_val,
                                    num_of_neighbours=numOfNeighbours,
                                    dist_metric=typeOfDist)
        y_pred = k_nearest_neighbours()
        achievedAcc = myAccuracy(y_val, y_pred)
        if achievedAcc > bestAccuracy :
            bestDistance = typeOfDist
            bestNN = numOfNeighbours
            bestAccuracy = achievedAcc
print('For this fold: best NN:', bestNN, 'best Dist:', bestDistance)
```

Figures 2 & 3: Creating the x & y train, validation and test sets and using the train and validation ones to choose the best set of parameters by running the kNN classifier on all the possible numbers of neighbours and types of distances.

Finally, *mykNN* classifier is instantiated once more to classify the actual **test data** using the best-performing parameters obtained through the previous step. This is repeated for all of the folds producing a list of accuracy results achieved in each that the function returns along the list containing all the generated predictions and another list holding the true labels (the last 2 lists are used to then construct the confusion matrix for a totality of the results).

2. Results analysis.

Figures [4] and [5] show the results achieved by the classifier:



Clean data	accuracy	k	distance	Noisy data	accuracy	k	distance
Fold1	0.966666667	3	euclidean	Fold1	0.733333333	1	euclidean
Fold2	0.933333333	3	manhattan	Fold2	0.866666667	7	manhattan
Fold3	0.933333333	1	euclidean	Fold3	0.8	2	euclidean
Fold4	1	4	euclidean	Fold4	1	5	euclidean
Fold5	0.966666667	1	euclidean	Fold5	0.866666667	6	manhattan
Total	**0.96**	± 0.024944382578492935		Total	**0.853333333**	± 0.08844332774281068	

Figures 4 & 5: Result tables of running the kNN classification (with nested cross-validation) on, consecutively, the clean and the noisy data.

2 main conclusions can be drawn from the above performance:

a) The *k* and *distance* parameters did change on each fold proving the nested cross-validation step was necessary to efficiently choose the parameters **before** performing kNN on the test data. Moreover, the Euclidean distance measure showed to be more efficient than the Manhattan one by achieving better (or equal) results with a smaller amount of the nearest neighbours needed to compare. In a production-rate task, this characteristic could allow for a performance improvement because the O(NlogN)[1] *getNeighbours* function would be called less times to achieve the same or better results compared to the Manhattan measure.

b) Generally, more neighbours were needed to perform an accurate prediction on the noisy data than it was needed for the clean data. This was expected as the noisier the data is, the more likely that a test datapoint lies in a vicinity of the outlier datapoints. Therefore, a bigger sample of the datapoints to compare (neighbours) is necessary in order to correctly determine the class of the test datapoint.

[1] The computational cost of O(NlogN) comes from the fact the *getNeighbours* needs to sort the array of distances with all the neighbours in order to obtain the indices of the first *n* neighbours.

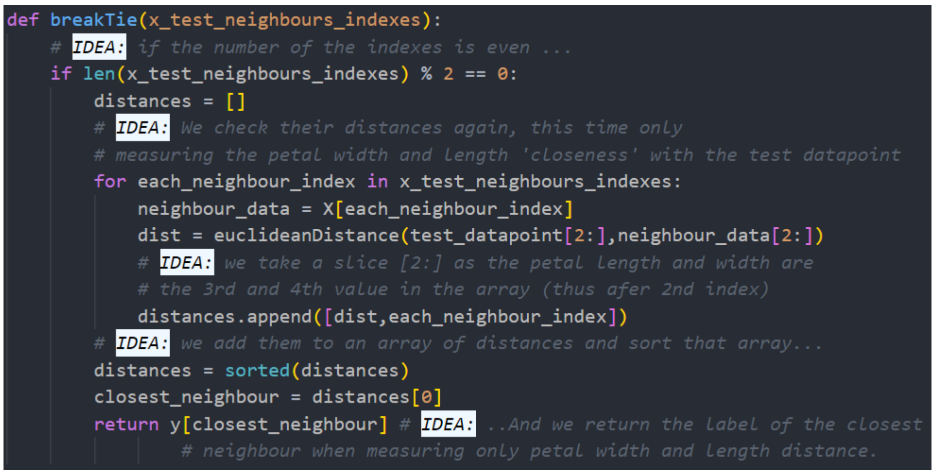
3. Discussion.

3.1. Exploratory Data Analysis.

By observing the plots of the clean data it is highly visible the data is structured and it is easy to distinguish the class instances due to the fact the feature's value ranges are class-correlated. For example, examining plants' "petal length x petal width" plot shows "class 0" plants have small petal sizes, "class 1" plants have medium petal sizes and almost all (45 of 50) of the class 2 have large petal values. Through introducing the Gaussian blur to the dataset, the values in the plots start to overlap much more frequently. However, it is still possible to distinguish the general trends in the plots of the most class-correlated values (like the petal sizes plot described above), therefore the classifier was able to achieve the relatively high score of 85% accuracy despite the Gaussian blur applied to the data.

3.2 Tie breaking.

In the case of a tie in assigning the predicted label based on an even number of neighbours with different classes, one of the solutions is calculating the Euclidean distance (as it generally gave better performance than Manhattan) only of the most class-correlated features (i.e. petal length and petal width, according to the dataset description) instead of all 4 features. A python-based pseudocode for this would look like the following:

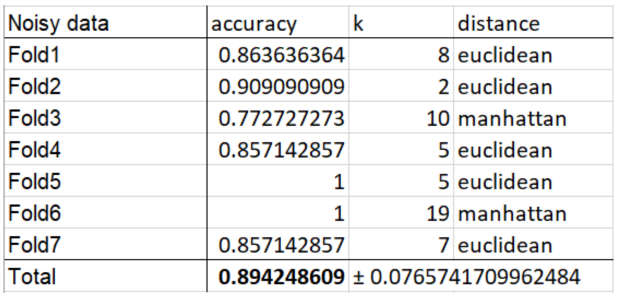


```
def breakTie(x_test_neighbours_indexes):
    # IDEA: if the number of the indexes is even ...
    if len(x_test_neighbours_indexes) % 2 == 0:
        distances = []
        # IDEA: We check their distances again, this time only
        # measuring the petal width and length 'closeness' with the test datapoint
        for each_neighbour_index in x_test_neighbours_indexes:
            neighbour_data = X[each_neighbour_index]
            dist = euclideanDistance(test_datapoint[2:],neighbour_data[2:])
            # IDEA: we take a slice [2:] as the petal length and width are
            # the 3rd and 4th value in the array (thus afer 2nd index)
            distances.append([dist,each_neighbour_index])
        # IDEA: we add them to an array of distances and sort that array...
        distances = sorted(distances)
        closest_neighbour = distances[0]
        return y[closest_neighbour] # IDEA: ..And we return the label of the closest
                # neighbour when measuring only petal width and length distance.
```

Figure 6: Pseudocode for the tie breaking method

3.3 Improving performance on noisy data.

Apart from implementing the tiebreak method, the quickest way of improving the results is to increase the number of folds and the allowed number of neighbours to compare, e.g. to 7 folds and up to 20 neighbours. This approach yields the following, better results:



Noisy data	accuracy	k	distance
Fold1	0.863636364	8	euclidean
Fold2	0.909090909	2	euclidean
Fold3	0.772727273	10	manhattan
Fold4	0.857142857	5	euclidean
Fold5	1	5	euclidean
Fold6	1	19	manhattan
Fold7	0.857142857	7	euclidean
Total	**0.894248609**	± 0.0765741709962484	

Figure 7: The results of running the kNN + nested cross-validation on the noisy data with an increased number of folds and possible neighbours to compare.