Machine Learning - Coursework 1

Elena Martina

March 2019

1 Implementation

My implementation of the k-Nearest Neighbours algorithm was inspired by the Scikit-Learn [3] and MATLAB [2] implementations of the k-NN classifier.

The function "$mykNN()$" is defined as following:

- 3 required parameters:
  - X: the training data
  - y: the training labels
  - $X\_$: the testing data
- 5 optional parameters:
  - $n\_neighbours$: the number of neighbours ($default = 5$)
  - $distance\_metric$: the distance function to be used ($default =$ "$minkowski''$)
  - p: power parameter of the Minkowski distance. ($default = 2$)
  - $breaktie\_metric$: the metric to use when a tie occurs ($default =$ "$smallest''$)
  - $weights\_method$: the method to use to weight the neighbours ($default =$ "$uniform''$)

The function finds the k nearest neighbours for the testing data: it firstly calculates the distances between each testing point and the training data, storing them along with the index of their respective data point, for then sorting the array of distances and returning only its first k elements. The distances of the k nearest neighbours are returned along with the neighbours in order to then calculate the weights.
After obtaining the neighbours, the function finds the mode/modes between the different labels among the neighbours, in order to assign to the test data the most frequent class. As explained in details in Section 3.3, the mode can be calculated also by weighting the neighbours according to their distances.
If two or more different classes appear to be the most frequent, the algorithm calls the "$break\_tie()$" function (explained in details in Section 3.2), which will deal with the tie in the method defined in the parameter $breaktie\_metric$, and return the predicted label.

1.1 Distance Metrics

The default distance metric of my implementation is the Minkowski metric, with $p = 2$, which corresponds to the Euclidean distance. The distance metrics I used were the following:

- **Minkowski:** The Minkowski distance is considered as a generalization of both the Euclidean distance and the Manhattan distance. It can be calculated with the following formula: $\sqrt[p]{\sum_{i=1}^{n}(q_i - p_i)^p}$
- **Euclidean:** The Euclidean distance is a special case of the Minkowski distance, where $p = 2$. It can be calculated with the following formula: $D(p, q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$
- **Manhattan**: The Manhattan distance is a special case of the Minkowski distance, where $p = 1$. It can be calculated with the following formula: $D(p, q) = \sum_{i=1}^{n}(|q_i - p_i|)$
- **Chebyshev**: The Chebyshev distance is a special case of the Minkowski distance, where $p = \infty$. It can be calculated with the following formula: $D(p, q) = \sum_{i=1}^{n} max(|q_i - p_i|)$
- **Hassanat:** This distance metric was defined by Hassanat in 2014 [1]; the values returned by this function are bounded by the interval $[0, 1)$, which makes this distance measure more invariant to different scale, noise and outliers. The more two values are similar, the nearest to zero the distance will be, and the more they are dissimilar, the nearest to one the distance will be. It can be calculated with the following formula: $\sum_{i=1}^{n} D(p_i, q_i)$, given:

$$D(p_i, q_i) = \begin{cases} 1 - \dfrac{1 + min(q_i - p_i)}{1 + max(q_i - p_i)} & \text{if } min(q_i - p_i) \geq 0\,, \\ 1 - \dfrac{1 + min(q_i - p_i) + |min(q_i - p_i)|}{1 + max(q_i - p_i) + |min(q_i - p_i)|} & \text{if } min(q_i - p_i) < 0\,. \end{cases}$$

1.2 Cross Validation and Evaluation

My implementation performs a $5k$ nested cross-validation, that divides the data in 5 different bins/folds; each fold has a Training, Validation, and Test set: $\frac{3}{5}$ of the data is assigned to Training, while Validation and Testing correspond to $\frac{1}{5}$ each. For each fold, the function loops through all the k-NN parameters (distance metric, k, breaktie metric and weights method) in order to run "$mykNN()$" function on the current training and validation data with all the possible combinations of parameters, in order to establish the best parameter set on the fold. At the end of each fold, the best set of parameters is run on the testing data and the accuracy is stored in the array "accuracy_fold".

Custom functions are used to evaluate the results:

- $myConfMat(y\_test, y\_pred)$ returns the confusion matrix displaying the given test data and predictions
- $myAccuracy(y\_test, y\_pred)$ returns the accuracy of the predictions given test data
- $myPrecision(y\_test, y\_pred)$ returns the precision corresponding to each individual class
- $myRecall(y\_test, y\_pred)$ returns the recall corresponding to each individual class

2 Results

2.1 Performance for Clean Data

Clean data	accuracy	k	distance	tie break	weights
Fold1	97	3	euclidean	smallest	distance
Fold2	94	4	euclidean	smallest	uniform
Fold3	94	1	euclidean	smallest	distance
Fold4	100	4	euclidean	smallest	uniform
Fold5	97	1	euclidean	smallest	distance
total	96 $\pm$ 0.25				

Table 1: Table showing the overall performance of the algorithm on the data without noise. The table shows test accuracy, standard deviation and parameter choices over the 5-folds.

2.2 Performance for Data with Noise

Noisy data	accuracy	k	distance	tie break	weights
Fold1	84	2	chebyshev	smallest	uniform
Fold2	77	6	hassanat	random	distance
Fold3	84	3	chebyshev	random	distance
Fold4	97	4	chebyshev	nearest	uniform
Fold5	84	3	chebyshev	random	distance
total	85 $\pm$ 0.65				

Table 2: Table showing the overall performance of the algorithm on the data after applying Gaussian noise. The table shows test accuracy, standard deviation and parameter choices over the 5-folds.

2.3 Confusion Matrices

Clean Data

		Predicted		
		Class 0	Class 1	Class 2
Actual	Class 0	50	0	0
	Class 1	0	47	3
	Class 2	0	3	47

Noisy Data

		Predicted		
		Class 0	Class 1	Class 2
Actual	Class 0	50	0	0
	Class 1	0	39	11
	Class 2	0	12	38

Table 3: Confusion Matrices over the 5-fold cross-validation for the Clean Data (on the left) and Noisy Data (on the right).

2.4 Results Analysis

The first thing that comes to mind while observing the data is the evident difference between the performance on the Clean and on the Noisy Data. Unsurprisingly, the algorithm performs better on the former, with an accuracy of 96%, more than 10% higher than the accuracy of the latter. The differences between noisy and clean data do not appear just in the accuracy score, but also in the best parameter choices:

In regards to the value of k, it is easy to observe how the range of best k is different between clean and noisy data. Without the noise, the best values of k are much lower, ranging from 1 to 4 (see Table 1), in respect to the best values of k with noise, which are greater, ranging from 2 to 6 (see Table 2). This is because noisy data is more sparse, and the algorithm performs better when considering a wider number of neighbours.

Moreover, in the clean data, as we can see from Table 1, the preferred distance appear to be the Euclidean distance for every fold, as well as the default tie breaking method, "smallest". After applying the noise however, as shown in Table 2, the Chebyshev distance results as the most frequent choice, while the preferred tie breaking method is "random".

Overall, the parameters per fold are very similar, while being different between clean and noisy data. However, it would be easy to find good-performing parameters for both of the two data used: Euclidean distance indeed, performs almost as well as Chebyshev distance on the noisy data, and would therefore be a valid parameter choice in both cases. As for the number of k nearest neighbours, $k = 4$ would probably an effective value for both, being the best-performing k value for both clean and noisy data.

3 Questions

3.1 Exploratory Data Analysis

The scatterplots in Figure 1 make it possible to visualize how the data is distributed among the 3 different classes.

At a first glance, it can be noticed how, for every pair of features, one particular class (the one in a light blue color) is always easy to discern by eye from the other two. This particular class (class 0), appeared to be easily identifiable also by the k-NN classifier, as reported in the results in Table 3, where it can be seen how class 0 was never miss-classified by the algorithm. The other two classes instead, class 1 and 2, seem to have more similar features, as their data points are adjacent in most of the scatterplots, and even slightly overlap in certain cases (i.e. when comparing the sepal width with the sepal length and vice versa).

After adding Gaussian noise to the data, we can see how the data points for each label moved further away from their center. This "expansion" causes the data from different categories to overlap and being less easily classifiable: although class 0 appears to be still separated from the other two classes, the data points from class 1 and 2 heavily overlap in all the different scatterplots, which indeed results in a less accurate classification of the two.

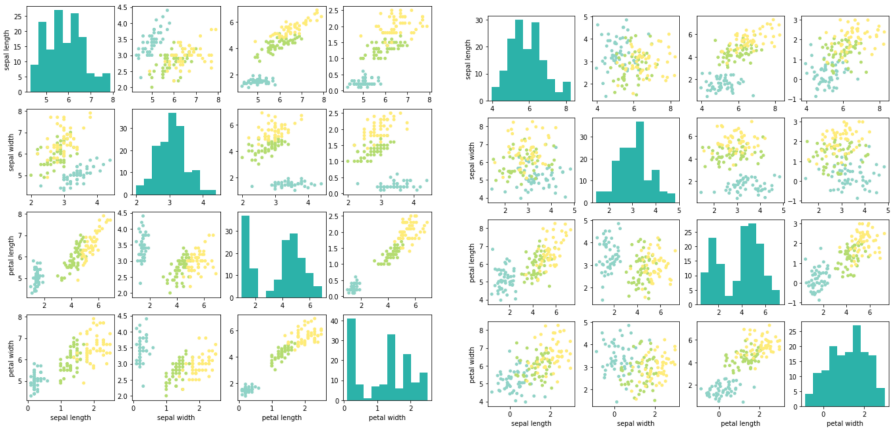

Figure 1: Plots for every pair of features. On the left, plots for data without noise, on the right, plots for data with noise.

3.2 Tie Breaking

There are multiple ways of breaking a tie in the k-NN algorithm.

One method could consist in decreasing the k (number of neighbours) by 1 until one class will prevail over the others. This method would always be valid, regardless of the weighting scheme or the value of k, since it would be impossible to have a tie when $k = 1$.

In my implementation, I used three different methods to deal with ties, without needing to reduce k. After calculating the most frequent class/classes amongst the neighbours' labels, if a tie is found, the *assign_label*() function would call a function, *break_tie*(), in order to apply a particular metric to break the tie and return the decided label.

```
def break_tie(breaktie_metric, labels, indices):

    if breaktie_metric == 'random':
        label = random.choice(labels)
        return label

    elif breaktie_metric == 'nearest':
        for i in indices:
            if i in labels:
                return i
    else:
        return labels[0] # default: smallest

```

Figure 2: My "*break_tie*()" function implementation.

The *break_tie*() function, shown in Figure 2, will deal with the tied groups in one of the following ways:

- If the *breaktie_metric* is "*smallest*" (default value), it will simply pick the first class it finds (the smallest index) among tied groups;
- If the *breaktie_metric* is "*nearest*", it will pick the class with the nearest neighbour among tied groups;
- If the *breaktie_metric* is "*random*", it will randomly pick a class among tied group.

3.3 Improving performance on noisy data

After adding the Gaussian noise, the data for each label looked more sparse and far away from the label's center, as shown in Figure 1; this is why, as observed in the results in Section 2, the best values of k resulted higher in respect to the clean data. As the data becomes more sparse, it is necessary to consider a wider range of neighbours in order to be able to decide which class to assign to a sample.

In my implementation, I was able to achieve a better accuracy on the noisy data by applying distances like the Chebyshev and the Hassanat distances, which resulted more resistant to noise than the Euclidean distance, and by using the method "random" to break ties.

I also found useful weighting the neighbours. I implemented the weighting by creating an array of weights of the same shape of the neighbours array; the function is shown in Figure 3. If the weighting metric corresponds "$uniform$" or is None (default value), the array of weights consists in an array of ones, as all points in each neighborhood are weighted equally. Otherwise, if the weighting metric is set to "*distance*", each neighbour is weighted by the inverse of its distance. In this case, closer neighbours of a query point will have a greater influence than neighbors which are further away. This improved the accuracy on the noisy data.

```
def get_weighted_modes(labels, weights_method, neighbours_distances):


    if weights_method == 'distance':
        weights = [(1/d) for d in neighbours_distances]
    else:
        weights = np.ones(len(neighbours_distances))

    classes, indices = np.unique(labels, return_inverse=True)
    counts = np.zeros(classes.shape)

    for c, w in zip(indices, weights):
        counts[c] += w

    return np.argwhere(counts == np.amax(counts)).flatten().tolist()

```

Figure 3: My "*get_weighted_mode*()" function implementation.

References

[1] Ahmad Basheer Hassanat, Mohammad Ali Abbadi, Ghada Awad Altarawneh, and Ahmad Ali Alhasanat. Solving the Problem of the K Parameter in the KNN Classifier Using an Ensemble Learning Approach. Technical Report 8, 2014.

[2] The MathWorks Inc. MATLAB Documentation. Classification knn, 2019. https://uk.mathworks.com/help/stats/classificationknn.html, Last accessed on 2019-03-7.

[3] Scikit-Learn Documentation. sklearn.neighbors.kneighborsclassifier, 2018. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html, Last accessed on 2019-03-7.