Assignment 1: Implementation of KNN

Dalia Ibrahim[1] and Carlos Dasaed Salcedo[2]

Studnet ID: [1]201893217, [2]201892008

January 25, 2019

1 Introduction

For this assignment, we have implemented KNN from scratch using Python and the following libraries: pandas, numpy, math, random, time, and sys. The library sklearn was also used, but only for comparison and validation purposes. The output of the program will be displayed in the terminal, but it will also be stored in a csv file containing the results of the TestData. In our KNN, we used the euclidean distance function, and did a majority vote to determine the class. The techniques and strategies used to improve the algorithm will be discussed in the improvements section. We have also included an appendix containing a table with the results of our validation tests, which is the basis of our conclusion and decision behind the current version of our KNN.py program. To execute the program, you must run the following command from the command line:
$python3 KNN.py [TrainingData] [TestData] [K]

2 pseudo-code of KNN.py

This section only includes the functions that are relevant and currently being used by the algorithm. The source code itself contains other functions that were used at some point for normalization, weight calculations and cross-validation, among other things.

2.1 KNNfunction()

Algorithm 1 KNN Function

Require: TrainingData , TestData, K
1: $FinalOuput = NewDataFrame('Class','Probability'$
2: **for** testRow **in** Test Data **do**
3: $AllDistancePerTest \leftarrow NewDataFrame()$
4: **for** trainRow **in** Training Data **do**
5: $DistPerTest[trainRow] \leftarrow EuclideanDistance(trainRow, testRow))$
6: **end for**
7: $TotalDistances \leftarrow sort(DistPerTest, Ascending)$
8: $BotKPerTest \leftarrow$Top K rows of TotalDistances
9: $TempOutput \leftarrow Calculate\_Nearest\_Neighbors(BotKPerTest, K)$
10: $FinalOutput.append(TempOutput)$
11: $print(FinalOutput)$
12: **end for**
13: $FinalOutput.csv \leftarrow FinalOutput$

2.2 EuclideanDistance()

Algorithm 2 Euclidean Distance Function

Require: trainRow , testRow
1: $Summation \leftarrow 0$
2: **for** i **in** range(NumFeatures) **do**
3: $d \leftarrow trainRow[i] - testRow[i]$
4: $Summation \leftarrow Summation + d^2$
5: **end for**
6: $Distance \leftarrow [SQRT(Summation), trainRow[LastValue]]$
7: return Distance

2.3 Calculate_Nearest_neighbour()

Algorithm 3 Calculate Nearest Neighbor Function

Require: topMatched , K

if $topMatched[0].distance == 0$ **then**
  $FinalOutput \leftarrow [topMatched.class, 1.00]$
  return FinalOutput
end if
$TieBreaker \leftarrow NewDataframe()$
$TieBreaker \leftarrow UniqueRandomNumbers()$
$topMatched.CreateColumn('Count') \leftarrow Count('Class')$
$concatenate(topMatched, TieBreaker)$
$topMatched = topMatched.Filter('Count' = Count.Max())$
$topMatched.sortBy('tieBreaker')$
$TopMatch \leftarrow topMatched.row(0)$
$Probability \leftarrow \frac{topMatch[Count]}{K}$
$FinalOutput \leftarrow [TopMatch[Class], Probability)]$
$returnFinalOutput$

3 KNN improvements

The chosen algorithm uses the normal Euclidean Distance formula to calculate the distance to all the neighbors, and then applies majority vote with only the K closest neighbors.

To prevent possible ties, random numbers without repetition are generated and assigned to each of the selected closest neighbors. In the case of a tie in the number of votes, the class of the neighbor with the smallest random number would be selected.
The following modifications were implemented and tested in an effort to improve the algorithm.

1. Normalization of the training and testing data
2. Addition of weights to the neighbors to improve voting in the selection of the closest neighbor
3. Normalization and Addition of weights together.

4 Testing the Algorithm

To validate the accuracy of our algorithm, we used crossed validation with the help of the sklearn library. These are the steps we followed to implement Cross Validation and test our code:

1. Shuffle the data using *trainingdata.sample*() function from sklearn Library
2. Split the data using *trainTestSplit* function in the sklearn Library and make the testing part equal to 40% of the total training data.
3. Calculate average accuracy by comparing actual data with the predicated data, and then applying the formula in (1)

$$Accuracy = \frac{CorrectPredicted}{AllOutputs} \quad (1)$$

4. Generate a report with Precision and Recall using the $classificationReport$ function from the sklearn Library
5. Repeat all pervious steps for Kfold = 5

5 Conclusion

Since we were using the sklearn to validate our results, we were able to generate a table to compare the accuracy of the results of the algorithm as we made and applied the different modifications. However, to our surprise, Classical KNN with a tie breaker function added yielded better results. Normalizing the data and adding weights actually reduced our accuracy to 79%.

6 Appendix

6.1 Output

Table 1 shows the results of running KNN with serveral modifications, and K=3.

Method	**Accuracy**
Classical KNN	85.00%
Classical KNN with TieBreaker	**100%**
KNN Weighted	84%
Normalized KNN	91.00%
Normalize KNN Weighted	79.00%

Table 1: Accuracy calculated using sklearn

	Class						
K=3	**1**	**2**	**3**	**5**	**6**	**7**	**Accurcy**
Classical KNN	Pre = 79% Recall = 85%	Pre = 88% Recall = 90%	Pre = 67% Recall = 40%	Pre = 100% Recall = 40%	Pre = 80% Recall = 80%	Pre = 100% Recall = 91%	85.00%
KNN Weighted	Pre =75% Recall = 81%	Pre = 93%Recall = 83%	Pre = 57% Recall = 67%	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	Pre = 91% Recall = 91%	84.00%
Normalized KNN	Pre = 92% Recall = 89%	Pre = 91%Recall = 94%	Pre = 60% Recall = 60%	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	**91.00%**
Normalize KNN Weighted	Pre = 71% Recall = 87%	Pre = 89%Recall = 74%	Pre = 60% Recall = 43%	Pre = 75% Recall = 100%	Pre = 50% Recall = 67%	Pre =100 %Recall = 100%	79.00%