Assignment 2: Choosing the Best Parameters to Use for a Binary KNN classifier using on 5-fold cross-validation

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

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

February 16, 2019

1 Introduction

For this assignment, we have implemented a cross-fold validation algorithm from scratch using Python and the following libraries: pandas, numpy, math, random, and sys. To improve the efficiency of the algorithm, unsupervised filtering was also used based on the correlation matrix and variances of the data using sklearn libraries. The other sklearn libraries included in the final program were used to calculate KNN, and related precision metrics. To run the program, the following line must be executed from the command line in Linux:
$python3 A2_t2.py [DataFile.tsv]

2 Preliminary Steps - Feature Selection

feature selection
cross correlation
low variance

3 Main Pseudo-code

This section only includes the functions that are relevant and currently being used by the algorithm. Functions, such as the ones from sklearn, will only be mentioned in the pseudo-code, but will not be individually described.

3.1 FoldSplitter()

Algorithm 1 FoldSplitter Function

Require: kfolds {kfolds = 5 was used}
$data \leftarrow dataframe(DataFile.tsv)$
$class0 \leftarrow$ data.where(class=0).ShuffleRows()
$class0partition \leftarrow RowsInclass0/kfolds$
$class1 \leftarrow$ data.where(class=1).ShuffleRows()
$class1partition \leftarrow RowsInclass1/kfolds$
$leftOvers0 \leftarrow$class0 rows after(class0partition * cvfolds)]
$leftOvers1 \leftarrow$class1 rows after(class1partition * cvfolds)]
$leftOvers \leftarrow concatenation(leftOvers0, leftOvers1)$
$theFolds \leftarrow newPythonDictionary$
for i=0 **to** kfolds-1 **do**
  $createfold$**i** {i = corresponding iteration in the for cycle}
  $class0Range \leftarrow$class0[**from** i * class0partition **to** (i * class0partition)+class0Partition
  $class1Range \leftarrow$ class1[**from** i * class1partition **to** (i * class1partition)+class1Partition
  $TempOutput \leftarrow Calculate\_Nearest\_Neighbors(BotKPerTest, K)$
  $FinalOutput.append(TempOutput)$
  $print(FinalOutput)$
end for
$FinalOutput.csv \leftarrow FinalOutput$

3.2 EuclideanDistance()

Algorithm 2 Euclidean Distance Function

Require: trainRow , testRow

$Summation \leftarrow 0$

for i **in** range(NumFeatures) **do**

  $d \leftarrow trainRow[i] - testRow[i]$

  $Summation \leftarrow Summation + d^2$

end for

$Distance \leftarrow [SQRT(Summation), trainRow[LastValue]]$

return Distance

3.3 Calculate_Nearest_neighbour()

Algorithm 3 Calculate Nearest Neighbor Function

Require: topMatched , K

$TieBreaker \leftarrow NewDataframe()$

$TieBreaker \leftarrow UniqueRandomNumbers()$

$topMatched.CreateColumn('Count') \leftarrow Count('Class')$

$concatenate(topMatched, TieBreaker)$

$topMatched = topMatched.Filter('Count' = Count.Max())$

$topMatched.sortBy('tieBreaker')$

$TopMatch \leftarrow topMatched.row(0)$

$Probability \leftarrow \frac{topMatch[Count]}{K}$

$FinalOutput \leftarrow [TopMatch[Class], Probability)]$

$returnFinalOutput$

4 Deciding on Performance

Since there are several different performance metrics that can be used to determine how good an algorithm will perform, we decided to use the AUC of the ROC curve

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.

5 Results

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)

$$Accuracy = \frac{CorrectPredicted}{AllOutputs} \tag{1}$$

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

6 Conclusion

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.

7 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 accuracy reaches up to 100%. Normalizing the data and adding weights actually reduced our accuracy to 79%.

8 Appendix

K=3	Class						
	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 = 79%	79.00%
Classical KNN with TieBreaker	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	Pre = 100% Recall = 100%	**100.00%**