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 17, 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 calculations of a 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 Feature Selection

To improve the performance of the algorithm, a process of feature selection was applied by eliminating least relevant features in the model. Eliminating features reduces the amount of calculations and overall complexity of the algorithm, but doing so cannot be done in a trivial and random manner. The process by which the features were eliminated will be described in the following subsections.

2.1 Selecting Highly Correlated Features

The correlation matrix function in the sklearn libraries was used to find highly correlated features. Since highly correlated features bring the same information, some of these can be removed without significantly affecting the overall performance. For this particular data set, various thresholds were tested, but 0.80 was determined to be the ideal candidate, as it reduced the features from 348 to 305, without affecting the performance.

2.2 Removing low-variance features

The next method of feature selection consisted in the removal of features with low variance across all the observations. For example, if a feature has zero variance, it means that all its values are the same, and therefore does not add any new information to the model. In our algorithm, a threshold of 0.9 was selected to allow for a total of up to 5 features.

3 Main Pseudo-code

Algorithm 3 from the lecture notes was used for the current implementation, but with only 5 folds, and skipping step one(i.e. using Nexp=1). A result grid of features versus KNN neighbors is generated to store the average AUC values. After the cross-correlation loop is completed, and using the result grid, the three models with the highest AUC are chosen as the best models, while the 2 models with the lowest AUCs are chosen as the worst. Other functions that were manually coded are a fold splitter function and a SplitData function. The fold splitter function is used to divide the data into 5 folds with similar proportions of 0s to 1s as in the original data set as a pre-step to cross validation. As the cross validation function iterates through the folds generated from the fold splitter, the SplitData function converts the fold in the current iteration into the testing data, and merges all the other folds into the training data. These functions were manually programmed as they are not part of the sklearn libraries used, and their respective pseudo-code can be found in the appendix section of this PDF if interested.

4 Loss Function

The Area Under the Curve (AUC) was chosen as the loss function. The best model was selected based on the ROC graph of the models with the highest AUCs. The model with the smallest AUC was also included for the purpose of having better comparisons.

5 Deciding on Performance

Since there are several different performance metrics that can be used to determine how good an algorithm will perform, it is important to make an effort to chose an adequate performance metric for the task at hand. Upon inspection of the training data, it became obvious that the amount of zeros drastically outnumbered the amount of ones by a ratio of about 10 to 1. This meant that blindly using accuracy as the factor to determine the best model would be insufficient, since an algorithm that always predicts the output class to be zero would actually be correct 90% of the time. Therefore, to determine which model was best, we decided to focus on identifying instances of class 1. In other words, we considered ones as positives, and zeros as negatives for the TPR, FPR, ROC curves, loss function and selection of our best model.

6 Results

To better display the results of our algorithm, we used the sklearn metrics library, which allowed us to generate Figures 1 and 2. Figure 1 shows a graphical representation of the best and worst models with the current algorithm. A sample run of KNN without any filtration of the features and 23 neighbors as seen in Figure 2 was also executed to have a base model to compare against. With only 4 features and 25 neighbors, our model managed to obtain an AUC of .88, slightly better than the .87 of the base model, with a lot less features, proving that KNN with effective parameter tuning and cross validation will ultimately generate a better model.

	K= 23, NumOfFeatures=4, AUC = 0.87		
	Precision	**Recall**	**F1-Score**
0	0.94	0.98	0.96
1	0.62	0.36	0.45

Table 1: Performance metrics of the best model

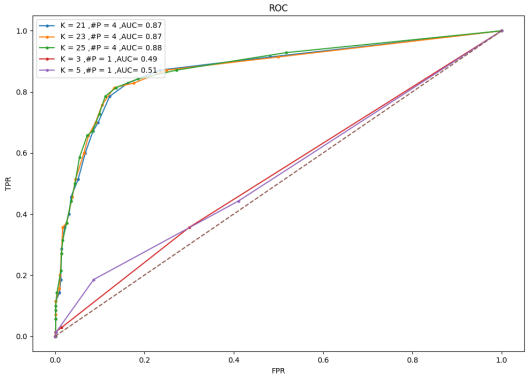

Fig. 1 ROC Curve for the 3 best models and the 2 worst models

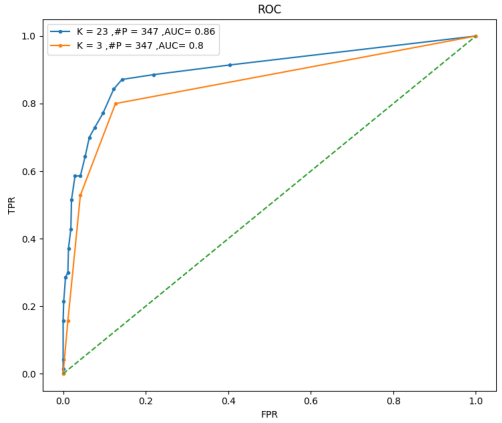

Fig. 2 ROC for KNN with 3 and 23 neighbors, and all the features in the dataset.

7 Conclusion

KNN is a powerful and useful machine learning classifier. However, just like any other classifier, without the proper parameters such as an adequate feature selection, number of neighbors, and correct training set, the model can easily become skewed or flawed. For this reason, it is important to select the proper performance metrics and to run cross validation tests with at least 5 folds. In our particular case, using 5 fold cross validation, and selecting the features based on a correlation matrix and variance, yielded consistent models with AUCs of up to 0.88.

8 Appendix

8.1 FoldSplitter()

Algorithm 1 Split Data in K folds

Require: kfolds {kfolds = 5 was used}

data ← $dataframe(DataFile.tsv)$
*class*0 ← data.where(class=0).ShuffleRows()
$class0partition \leftarrow RowsInclass0/kfolds$
*class*1 ← data.where(class=1).ShuffleRows()
$class1partition \leftarrow RowsInclass1/kfolds$
$leftOvers0$ ←class0 rows after(class0partition * cvfolds)]
$leftOvers1$ ←class1 rows after(class1partition * cvfolds)]
$leftOvers$ ← concatenation(leftOvers0, leftOvers1)
$theFolds \leftarrow newPythonDictionary$
for i=0 **to** kfolds-1 **do**
  $createfold$**i** {i = corresponding iteration in the for cycle}
  $class0Range$ ←class0[**from** i * class0partition **to** (i * class0partition)+class0partition
  $class1Range$ ← class1[**from** i * class1partition **to** (i * class1partition)+class1partition
  $fold$**i** ← $class0range + class1range$
  if $i = kfolds - 1$ **then**
    $fold$**i** ← $fold$**i** + $leftOvers$
  end if
  $theFolds[fold$**i**$] = TempData$
end for
return theFolds

8.2 SplitData()

Algorithm 2 Create Learning and Training Data

Require: DictionaryOfFolds , iterNum

$testFold \leftarrow fold\{iterNum\}$ {iterNum refers to the iteration number, such that testFold will get fold1,fold2,...,fold5}
$testDF \leftarrow DataFrame(testFold)$
$trainDF \leftarrow NewDataFrame$
for foldKey, foldValue **in** DictionaryOfFolds **do**
  if $foldkey = testFold$ **then**
    Skip this Iteration
  end if
  $trainDF.append(foldValue)$
end for
xtraining ← trainDF without the last column
ytraining ← testDF without the last column
xtesting ← trainDF with only the last column
ytesting ← testDF with only the last column
return xtraining, xtesting, ytraining, ytesting