lab2_report_adrien-guezennec

February 29, 2020

1 Report lab 2: Decision Tree.

1.1 Introduction

The main goal of this asignment was to create a decision tree from scratch for classification task. Before to rush in the code, I started to make some research of different kind of decision tree implementation. I found different implementation, like CART or ID3. I start to implement both on different code base but I stick with an alike CART implementation (whithout the regression task). I mainly looked on wikipedia website for information: Decision tree, ID3 Decsion tree. All the splitting part was, at the beginning, confusing for me but I also read this article which helped me a lot.

The code is available online on my Github account through this link.

1.2 Data Loader

At first, I create my own data loader. As it was indicate to test our decision tree with multiple dataset I simplify the process. Also because it's going to be helpful in other assignment.

Here is my DataLoader class:

[1]:
```
from sklearn import datasets

class Loader:
  def __init__(self):
    self.dataLoader = {
      'iris': datasets.load_iris,
      'wine': datasets.load_wine,
      'cancer': datasets.load_breast_cancer,
    }

  def getDataset(self, name):
    if name not in self.dataLoader:
      print('Error: the only dataset available are: ', [key for key in self.
 ↪dataLoader])
      return(84)
    return self.dataLoader[name]()
```

As you can see you are limited for the type of data possible but it can grow further as I need specific dataset.

1.3 Cross Validator

This wasn't required specificly in the assignment but I wanted my decision tree compliant with the cross validator that we had to made in assignment 1. My cross validator wasn't that generic as I thought so I had to change the extract function, used for extract the label of the instance in the dataset, to make it work. I used the same as I used in the decision tree.

[4]:
```
import random
import copy

class CrossValidator:
  def __init__(self, algo=None, dataset=None, nbFolds=10):
    random.seed(1)
    self.folds = list()
    self.algorithm = algo
    self.dataset = self.__transformDataIntoList(dataset)
    self.nbFolds = nbFolds
    self.rocData = list()

  def __transformDataIntoList(self, data):
    if data is not None:
      return [list(instance) for instance in data]
    return None

  # Method to check if the CrossValidator class as everything needed to start.
  def __checkNotEmptyAttributes(self):
    if (self.algorithm is None or self.dataset is None or self.nbFolds <= 1):
      print("Error: Algorithm and dataset shouldn't be empty and nbFolds␣
↪neither less nor equal to 0")
      return False
    return True

  # nbInstances as to be lower than nbFolds, I round the return to get a␣
↪integer and not a float.
  def __getFoldSize(self, nbInstances, nbFolds):
    return round(nbInstances / nbFolds)

  # Fill the folds Class variable with all folds of instances shuffled.
  def __splitDatasetIntoKFolds(self):
    copyDataset = self.dataset.copy()
    # Desorganize the dataset
    random.shuffle(copyDataset)
    # Get the number of instances in each fold.
    foldSize = self.__getFoldSize(len(self.dataset), self.nbFolds)

    # I move the pointer start and end to cur the dataset into the number of␣
↪instances calculated.
    for nb in range(self.nbFolds):
```

```
      start = foldSize * nb
      end = foldSize + start
      self.folds.append(copyDataset[start:end])

  # Return the taget label from the dataset, target as to be at the end.
  def __extractTargetFromDataset(self, data):
    newDataset = list()
    target = list()
    for instance in data:
      target.append(instance[len(instance) - 1])
      newDataset.append(list(instance[:-1][0]))
    return newDataset, target

  # Count the correct answer and return the accuracy of them, scaled on 0 to␣
↪100%.
  def __getAccuracy(self, original, predictions):
    nbCorrectPredictions = 0

    # Loop through all the predictions.
    for index in range(len(predictions)):
      # Get the class predicted.
      predictClass = predictions[index][0]

      # If she correspond to the target label then add a correct answer.
      if (original[index] == predictClass):
        nbCorrectPredictions += 1

    return nbCorrectPredictions / len(original) * 100

  # Return a simple list of instances as a deep copy and delete the testing␣
↪fold.
  def __getTrainData(self, dataToSquash, indexToRemove):
    data = copy.deepcopy(self.folds)
    data.pop(indexToRemove)
    return sum(data, [])

  # Some magic here. Add the target label to the prediction information for the␣
↪ROC.
  def __appendTargetToPrediction(self, targets, predictions):
    for index in range(len(predictions)):
      tmp = list(predictions[index])
      tmp.append(targets[index])
      predictions[index] = tmp
    return predictions

  # Calculate the score of the accuracy:
  #   - all the accuracy as a list, len(list accuracy) = nbFolds.
```

```
#   - the mean accuracy based on all the accuracy.
#   - a list with -> Prediction, Classes probabilities, Real target expected.
def score(self):
  if (self.__checkNotEmptyAttributes() is False):
    return False

  # Split the data into K folds.
  self.__splitDatasetIntoKFolds()
  accuracyScores = list()
  for index, fold in enumerate(self.folds):
    trainData = self.__getTrainData(self.folds, index)
    # Extract the label from the dataset.
    X_train, targetTrain = self.__extractTargetFromDataset(trainData)
    testData = copy.deepcopy(fold)
    X_test, targetTest = self.__extractTargetFromDataset(testData)

    # Train the Naive Bayes algorithm.
    self.algorithm.fit(X_train, targetTrain)
    # Predict with the fold.
    acc, predictionFold = self.algorithm.predict(X_test, targetTest)
    # Add to the data for the ROC the prediction information with the target␣
↪label.
    self.rocData.extend(self.__appendTargetToPrediction(targetTest,␣
↪predictionFold))
    # Add the accuracy calculate.
    accuracyScores.append(acc)
  return accuracyScores, sum(accuracyScores) / len(accuracyScores), self.
↪rocData
```

I'm not gonna re-explain it because it is the same as in the assignment 2, but if you need more understanding I added a lot of comment in the code to explain what I did an how it works. You could also check on the assignment 1 report to get more information about it.

1.4 Decision Tree

Before to go any further and after my research about decision trees, I found important to have the same way of use for all algorithm. In assignment 1, I took inspiration from sklearn with the way to use the different algorithm. They all have the same function to train and to predict. So, as it was important for my cross validator to have the same function, I used the same standard from my naive bayes.

The function to train is `fit()` and take 2 parameters: 1 - Dataset (mandatory) 2 - Label for each instance in param 1 (mandatory)

The function for prediction is `predict()` and take 2 parameters as well: 1 - Dataset (mandatory) 2 - Label for each instance in param 1 (optional) If no label provided then return a list with all prediction made for each instance. It's up to the user to check and get the accuracy after prediction in this case. Otherwise, the accuracy is also returned with the list.

For the development I also add a function to display the tree created, I thought this could be usefull so I let the function in it. Even though, the display might be confusing to understand at the beginning.

At the creation of the decision tree you can precise if you want a max depth for the tree. I add this parameter which I found usefull if you have a huge number of attributes in your dataset.

[5]:
```
from math import pow

class DecisionTree:
  def __init__(self, maxDepth=None):
    self.__tree = None
    self.__maxDepth = maxDepth

  def __transformDataIntoList(self, data):
    return [list(instance) for instance in data]

  # Return the taget label from the dataset, target as to be at the end.
  def __extractTargetFromDataset(self, data):
    newDataset = list()
    target = list()
    for instance in data:
      target.append(instance[len(instance) - 1])
      newDataset.append(instance[:-1])
    return newDataset, target

  # Add target label to the dataset (add 1 column).
  def __concateTargetWithDataset(self, dataset, targetDataset):
    data = list()
    for index, instance in enumerate(dataset):
      tmp = list()
      if type(instance) is not list:
        print('NOT', instance)
        tmp.append(instance)
      else:
        tmp = list(instance)
      tmp.append(targetDataset[index])
      data.append(tmp)
    return data

  def __countUniqueValue(self, data):
    return list(set(data))

  # Return a dict with all the classes as key and the nb of each class as value.
  def __countAllElemInList(self, listElem):
    nbAllElem = dict()

    for elem in listElem:
```

```
      if elem not in nbAllElem:
        nbAllElem[elem] = 1
      else:
        nbAllElem[elem] += 1
    return nbAllElem

  def __calculateGiniScore(self, leafs):
    # Get size of all instance in each leaf.
    nbInstances = sum([len(leaf) for leaf in leafs])
    giniScore = list()

    for leaf in leafs:
      data, target = self.__extractTargetFromDataset(leaf)
      # Get size of the leaf and if 0 then return 0 since there is no data.
      sizeLeaf = len(data)
      if sizeLeaf == 0: continue
      # Count all instance depending on each classes.
      nbClassElem = self.__countAllElemInList(target)
      # Add the score for each class together.
      classScore = sum([pow((val / sizeLeaf), 2) for val in nbClassElem.
↪values()])
      giniScore.append((1.0 - classScore) * (sizeLeaf / nbInstances))
    return sum(giniScore)

  # Return the 2 leaf containing the splitted data on the breakpoint.
  def __split(self, data, indexAttr, splitValue):
    leftLeaf = list()
    rightLeaf = list()

    # According to the subject, lower value to the left and rest at the right.
    for instance in data:
      # print(instance[indexAttr], splitValue)
      if instance[indexAttr] < splitValue:
        leftLeaf.append(instance)
      else:
        rightLeaf.append(instance)
    return leftLeaf, rightLeaf

  # Return a dict for the best split node found.
  def __foundBestSplit(self, dataset):
    # Detached the target label from the dataset.
    data, _target = self.__extractTargetFromDataset(dataset)
    tree = dict()
    indexAttr = 0

    # Loop through each attribute, zip return all the column at once.
    for attribute in zip(*data):
```

```
      for value in attribute:
        # Get the two leaf for split (left and right leafs).
        leftLeaf, rightLeaf = self.__split(dataset, indexAttr, value)
        # Calculate gini scrore for value as breakpoint.
        giniScore = self.__calculateGiniScore((leftLeaf, rightLeaf, ))
        if not tree or tree['gini'] > giniScore:
          tree = {'breakpoint': value, 'indexAttr': indexAttr, 'leftLeaf':␣
↪leftLeaf, 'rightLeaf': rightLeaf, 'gini': giniScore}
      indexAttr += 1
    return tree

  def __getResult(self, leafs):
    # Extract the target from the leafs to get the result.
    # If multiple target then take the highest one.
    _data, target = self.__extractTargetFromDataset(leafs)
    return max(self.__countUniqueValue(target))

  def __recursiveCreation(self, tree, depth, maxDepth):
    # Check empty data in split.
    if not tree['leftLeaf'] or not tree['rightLeaf']:
      joinLeaf = tree['leftLeaf'] + tree['rightLeaf']
      tree['leftLeaf'] = self.__getResult(joinLeaf)
      tree['rightLeaf'] = self.__getResult(joinLeaf)
      return
    elif maxDepth is not None and maxDepth >= depth:
      # If maxDepth set then check if value is reach.
      tree['leftLeaf'] = self.__getResult(tree['leftLeaf'])
      tree['rightLeaf'] = self.__getResult(tree['rightLeaf'])
      return

    # Split left
    tree['leftLeaf'] = self.__foundBestSplit(tree['leftLeaf'])
    self.__recursiveCreation(tree['leftLeaf'], depth + 1, maxDepth)

    # Split right
    tree['rightLeaf'] = self.__foundBestSplit(tree['rightLeaf'])
    self.__recursiveCreation(tree['rightLeaf'], depth + 1, maxDepth)

  def __createTree(self, dataset):
    # Create root node of the tree.
    self.__tree = self.__foundBestSplit(dataset)
    # Create rest of the tree.
    self.__recursiveCreation(self.__tree, 0, self.__maxDepth)

  # Function to train and create a decision tree.
  def fit(self, dataset, target):
    # Get the root of the tree at first.
```

```
    dataset = self.__transformDataIntoList(dataset)
    # Start creating the leaf of the tree with the split.
    # print(dataset, target)
    dataset = self.__concateTargetWithDataset(dataset, target)
    self.__createTree(dataset)

  def __getAccurary(self, predictions):
    nbPredictions = len(predictions)
    counter = 0

    for predict in predictions:
      if predict[0] == predict[1]:
        counter += 1
    return round(counter / nbPredictions, 2)

  # Make recursive prediction through the all tree.
  def __makePrediction(self, instance, tree):
    # If attribute of the instance is lower than the breakpoint found in the␣
↪training
    # then go to the right of the tree.
    if instance[tree['indexAttr']] > tree['breakpoint']:
      # Check if the right is a leaf or a endpoint.
      if isinstance(tree['rightLeaf'], dict):
        return self.__makePrediction(instance, tree['rightLeaf'])
      # If not a real leaf then return the result branch
      return tree['rightLeaf']
    else:
      # Doing exactly the same but for the left branch of the tree.
      if isinstance(tree['leftLeaf'], dict):
        return self.__makePrediction(instance, tree['leftLeaf'])
      # If not a real leaf then return the result branch
      return tree['leftLeaf']

  # Make prediction on an instance or a list.
  # Return a list of Tuple as (TARGET, PREDICTION).
  # If no target provide then return list of predictions.
  def predict(self, dataset, target=None):
    if self.__tree is None:
      print('Error: You need to fit the decision tree first with fit(dataset,␣
↪target).')
      return 84

    predictions = list()
    # Check for one row only prediction.
    if len(dataset) == 1:
      self.__makePrediction(dataset, self.__tree)
    else:
```

```
    # Otherwise iterate through the all dataset and make a prediction for␣
↪each instance.
    for index, instance in enumerate(dataset):
      result = self.__makePrediction(instance, self.__tree)
      if target is None:
        predictions.append(result)
      else:
        predictions.append((target[index], result, ))
    accuracy = self.__getAccurary(predictions)
    return accuracy, predictions

 # Simple recursion function to display the tree trained.
 def __displayTree(self, tree, depth, label='root'):
   sentence = ''
   if isinstance(tree, dict):
     for _space in range(0, depth):
       sentence += ' '
     sentence += '%s -> X%d, value < %.3f, gini: %.3f' % (label,␣
↪tree['indexAttr'] + 1, tree['breakpoint'], tree['gini'])
     print(sentence)
     self.__displayTree(tree['leftLeaf'], depth + 1, 'left')
     self.__displayTree(tree['rightLeaf'], depth + 1, 'right')
   else:
     for _space in range(0, depth):
       sentence += ' '
     print(sentence, tree)

 # Function to call to display the training tree result.
 def show(self):
   if self.__tree is not None:
     self.__displayTree(self.__tree, 0)
   else:
     print('Error: No present tree. You need to fit the DecistionTree first.')
```

I force myself to make the function name and even the variable name the more clear possible to understand at the first lecture of the code. Concerne that it can be confusing, I also add lot of comments of each step in the code.

For the training, I used the gini function instead of the entropy in the dataset. Even if they provide the same kind of information which is the impurity of the given dataset. I found the gini easier to calculate and implement, that's why I started with this one. An implementation of the entropy with the information gain is ongoing but unfortunetly I missing time to delivered the working code for this part. Stay tuned on my Github repository to check the update.

1.5 Main

Here you will find the 4 main function called when you called the `main.py` file. Why 4 different main? Because I tested my desicion tree with 3 different dataset: Iris, Wine and Breast cancer.

The iris is tested whithout cross validation but the wine and breast cancer are tested with cross validation. If you gonna run the script, the breast cancer dataset is quite long to compute so just be patient.

I actually got 6 point ish difference with sklearn decision tree.

[7]:
```
from sklearn import tree
from sklearn.model_selection import train_test_split

from modules.Loader import Loader
from modules.CrossValidator import CrossValidator
from modules.DecisionTree import DecisionTree

loader = Loader()

# Add target label to the dataset (add 1 column).
def concateTargetWithDataset(dataset, targetDataset):
  data = list()
  for index, instance in enumerate(dataset):
    tmp = list()
    if type(instance) is not list:
      tmp.append(instance)
    else:
      tmp = list(instance)
    tmp.append(targetDataset[index])
    data.append(tmp)
  return data

def mainBreastCancer():
  cancerData = loader.getDataset('cancer')
  cancerData = concateTargetWithDataset(cancerData['data'],␣
 ↪cancerData['target'])
  decisionTree = DecisionTree()
  crossValidator = CrossValidator(algo=decisionTree, dataset=cancerData,␣
 ↪nbFolds=10)
  _scoresByFold, meanAccuracy, _rocData = crossValidator.score()
  print('Dataset: Breast cancer\nAccuracy: %.2f%%\n' % meanAccuracy)

def mainSklearn():
  irisData = loader.getDataset('iris')
  X_train, X_test, y_train, y_test = train_test_split(irisData['data'],␣
 ↪irisData['target'], test_size=0.80, random_state=42)
  clf = tree.DecisionTreeClassifier()
  clf = clf.fit(X_train, y_train)
  print('SKLEARN\nDataset: Iris\nAccuracy: %.2f%%\n' % clf.score(X_test,␣
 ↪y_test))

def crossValTest():
```

```
  irisData = loader.getDataset('iris')
  irisData = concateTargetWithDataset(irisData['data'], irisData['target'])
  decisionTree = DecisionTree()
  crossValidator = CrossValidator(algo=decisionTree, dataset=irisData,␣
 ↪nbFolds=10)
  _scoresByFold, meanAccuracy, _rocData = crossValidator.score()
  print('Dataset: Iris\nAccuracy: %.2f%%\n' % meanAccuracy)

def main():
  wineData = loader.getDataset('wine')
  X_train, X_test, y_train, y_test = train_test_split(wineData['data'],␣
 ↪wineData['target'], test_size=0.80, random_state=42)
  decisionTree = DecisionTree()
  decisionTree.fit(X_train, y_train)
  acc, _predictions = decisionTree.predict(X_test, y_test)
  print('Dataset: Wine\nAccurary: %.2f%%\n' % acc)

if __name__ == "__main__":
    main()
    crossValTest()
    mainBreastCancer()
    mainSklearn()
```

```
Dataset: Wine
Accurary: 0.87%

Dataset: Iris
Accuracy: 0.89%

Dataset: Breast cancer
Accuracy: 0.93%

SKLEARN
Dataset: Iris
Accuracy: 0.93%
```