```
"""Info
  @author Bardia Mojra
  @date April 13, 2021
  @brief Project 02 on Decision Trees for CSE6363 Machine Learning w Dr. Huber.
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import operator as opt
import inspect as i
import pdb
import pprint as pp

"""Globals
"""
eps = np.finfo(float).eps

class decisionTree:
  def __init__(self, XYtrain, maxDepth=None, prt=True):
    self.depth = 0
    self.maxDepth = maxDepth
    self.prt = prt
    self.X = XYtrain.drop(XYtrain.columns[-1], axis=1)
    self.Y = XYtrain[XYtrain.columns[-1]]
    self.features = np.asarray(self.X.columns)
    self.nFeatures = XYtrain.shape[1]
    self.mSamples = XYtrain.shape[0]
    self.df = self.X.copy()
    self.df['Y'] = self.Y.copy()
    # build decision tree
    if self.prt is True:
      print('-----------------------------------------------------')
      print("\n\n --> Initiate tree...")
    self.tree = self.buildTree(self.df)
    self.printTree()
    return

  def buildTree(self, df, tree=None):
    # determine which input feature results in highest infoGain
    feature = self.getBestSplit(df)
    if self.prt is True:
      print('feature: ', feature)
    # init tree
    if tree == None:
      tree = dict()
      tree[feature] = dict()

    if df[feature].dtypes != object: # can add numerical dTree later
      print('\n')
      print('df[feature]: ', df[feature])
      print('\n>>>Err: non-object feature at ln:', i.getframeinfo(i.currentframe
()).lineno)
      return
    else: # only works with labels
      for val in np.unique(df[feature]): # for each possible value in 'feature'
col
        df_ch = self.splitSamples(df, val, feature, opt.eq) # get child subset
        Y_objs, cnts = np.unique(df_ch['Y'], return_counts=True) # return Y clas
s cnts
```

```
        #pdb.set_trace()
        if(len(cnts)==1): # single-class, pure group -- Leaf Node
          tree[feature][val] = Y_objs[0]
        else: # impure
          self.depth += 1
          #pp.pprint(tree)
          #pdb.set_trace()
          if self.maxDepth is not None and self.depth >= self.maxDepth:
            tree[feature][val] = Y_objs[np.argmax(cnts)]
          else:
            tree[feature][val] = self.buildTree(df_ch)
    return tree

  def printTree(self):
    if self.prt is True:
      print('\n')
      print('-------------------- Decision Tree --------------------')
      print('Tree depth: ', self.maxDepth)
      pp.pprint(self.tree)
    return

  def splitSamples(self, df, val, col, _opt):
    df_new = df[_opt(df[col], val)] # in df, all in 'col' w 'val' that satisfy '
_opt' condition
    df_new = df_new.reset_index(drop=True) # drop old index, reset to num index
    return df_new

  def getTotalEntropy(self, data):
    """Calculates total entropy of the give dataset.
    """
    totalEntropy = 0
    for y in np.unique(data['Y']):
      frac = data['Y'].value_counts()[y] / len(data['Y'])
      totalEntropy += -frac * np.log2(frac)
    return totalEntropy

  def getFeatureEntropy(self, data, a):
    """Calculates entropy per feature for a given dataset, H_D(Y|A).
    """
    entropy = 0
    #threshold = None # for numeric features
    if data[a].dtypes == object: # make sure datatype is what we expect
      for val in np.unique(data[a]):  # sum of H_D(Y|A=a)
        featureEntropy = 0
        for y in np.unique(data['Y']):  # add all per datum feature entropies
          num = len(data[a][data[a] == val][data['Y'] == y])
          den = len(data[a][data[a] == val])
          infoGain = num / (den + eps)  # information gain
          if infoGain > 0:
            featureEntropy += -infoGain * np.log2(infoGain)
        featureWeight = len(data[a][data[a] == val]) / len(data)
        #print('feature: {1:>5s}'.format(4, a),'  weight:{:.5f}'.format(featureW
eight))
        entropy += featureWeight * featureEntropy
    else: # else could be numeric data
      print('>>>Err: none object data at ln:', i.getframeinfo(i.currentframe()).
lineno)
    return entropy

  def getBestSplit(self, df):
```

```
    """
      For a given dataset, it return the feature with highest information gain.
      InfoGain = Entropy(data) - Sum of Entropy(data_subsets)
      -> IG_D(Y|A) = H_D(Y) - H_D(Y|A) ; where D is given data and A is some inp
ut
      feature.
      Entropy =
      Sum of all Entropy(data_subsets) =
    """
    infoGain = list()
    igSum = 0.0
    parentEntropy = self.getTotalEntropy(df) # H_D(Y)
    if self.prt is True:
      print('\nparentEntropy:{:.5f}'.format(parentEntropy))
    for a in list(df.columns[:-1]):
      featureEntropy = self.getFeatureEntropy(df, a) # H_D(Y\A=a)
      infoGain_a = parentEntropy - featureEntropy
      igSum += infoGain_a
      infoGain.append(infoGain_a)
      #print('feature:{1:>5s}'.format(4, a), '  infoGain:{:.5f}'.format(infoGain
_a))
    if self.prt is True:
      print('Sum of infoGains: {:.5f}'.format(igSum))
    return df.columns[:-1][np.argmax(infoGain)]

  def y_est(self, xDatum, features, tree):
    #pdb.set_trace()
    for node in tree.keys():
      val = xDatum[node]
      if type(val) == str:
        #pdb.set_trace()
        if val not in tree[node]:
          tree = tree[0][val]
        else:
          tree = tree[node][val]
      else:
        pdb.set_trace()
        print('new val', list(tree[node].key())[0])
        tree = tree[node][val]
        pdb.set_trace()

      if type(tree) is dict:
        pred = self.y_est(xDatum, features, tree)
      else:
        pred = tree
        return pred
    return pred

  def getEst(self, X):
    predictions  = list()
    features = {label: i for i, label in enumerate(list(X.columns))}
    for idx in range(len(X)):
      predictions.append(self.y_est(X.iloc[idx], features, self.tree))
    return predictions
  # end of decisionTree class


class randForrest:
  def __init__(self, df, nTrees, nSamp, maxDep=None):
    self.df = df.copy(deep=True)
```

```
    self.X = df.drop(df.columns[-1], axis=1)
    self.Y = df[df.columns[-1]]
    self.nTrees = nTrees
    self.nFeat = int(np.log2(self.X.shape[1]))
    #self.nFeat = nFeat
    self.size = nSamp
    self.mxDep = maxDep

    #print(self.nFeat, "sha: ", self.X.shape[1])
    #pdb.set_trace()

    self.trainResHist = list()
    self.trees = list()

    for t in range(self.nTrees):
      df_temp = self.df.sample(self.size, replace=True) # new set, rand with rep
lacement
      df_new = df_temp.copy(deep=True)
      df_new.reset_index(drop=True, inplace=True)
      #print("df_new")
      #pp.pprint(df_new)
      Xn, Yn = getData(df_new)
      tree = decisionTree(df_new, maxDepth=maxDep, prt=False)
      Yn_est = tree.getEst(Xn)
      acc = getAcc(Yn, Yn_est)
      self.trainResHist.append(acc)
      self.trees.append(tree)
      #pdb.set_trace()
    return

  # end of bagging class

def getAcc(gndTruth, Est):
  correct = 0
  for i in range(len(gndTruth)):
    if gndTruth[i] == Est[i]:
      correct += 1
  return correct / float(len(gndTruth)) * 100.0

def getData(XY):
  X = XY.drop(XY.columns[-1], axis=1)
  Y = XY[XY.columns[-1]]
  return X, Y

"""Main
"""
if __name__ == "__main__":

  # import data
  XYtrain = pd.read_csv("./tic-tac-toe_train.csv")
  XYtrain = XYtrain.rename({'x':'p0','x.1':'p1','x.2':'p2','o':'p3','b':'p4', \
    'b.1':'p5','x.3':'p6','o.1':'p7','o.2':'p8'}, axis='columns')
  XYtest = pd.read_csv("./tic-tac-toe_test.csv")
  XYtest = XYtest.rename({'x':'p0','x.1':'p1','x.2':'p2','o':'p3','b':'p4', \
    'b.1':'p5','x.3':'p6','o.1':'p7','o.2':'p8'}, axis='columns')
  Xt, Yt = getData(XYtrain)
  Xtest, Ytest = getData(XYtest)

  # test
  print('Original Data Set:')
```

```
print(XYtrain)
#pdb.set_trace()


dTree = decisionTree(XYtrain, maxDepth=1)
Y_est = dTree.getEst(Xt)
acc = getAcc(Yt, Y_est)
print('Training set accuracy: {:.5f}'.format(acc))
Y_est = dTree.getEst(Xtest)
acc = getAcc(Ytest, Y_est)
print('Test accuracy: {:.5f}'.format(acc))

dTree = decisionTree(XYtrain, maxDepth=2)
Y_est = dTree.getEst(Xt)
acc = getAcc(Yt, Y_est)
print('Training set accuracy: {:.5f}'.format(acc))
Y_est = dTree.getEst(Xtest)
acc = getAcc(Ytest, Y_est)
print('Test accuracy: {:.5f}'.format(acc))

dTree = decisionTree(XYtrain, maxDepth=3)
Y_est = dTree.getEst(Xt)
acc = getAcc(Yt, Y_est)
print('Training set accuracy: {:.5f}'.format(acc))
Y_est = dTree.getEst(Xtest)
acc = getAcc(Ytest, Y_est)
print('Test accuracy: {:.5f}'.format(acc))

dTree = decisionTree(XYtrain, maxDepth=4)
Y_est = dTree.getEst(Xt)
acc = getAcc(Yt, Y_est)
print('Training set accuracy: {:.5f}'.format(acc))
Y_est = dTree.getEst(Xtest)
acc = getAcc(Ytest, Y_est)
print('Test accuracy: {:.5f}'.format(acc))

dTree = decisionTree(XYtrain, maxDepth=5)
Y_est = dTree.getEst(Xt)
acc = getAcc(Yt, Y_est)
print('Training set accuracy: {:.5f}'.format(acc))
Y_est = dTree.getEst(Xtest)
acc = getAcc(Ytest, Y_est)
print('Test accuracy: {:.5f}'.format(acc))

dTree = decisionTree(XYtrain, maxDepth=6)
Y_est = dTree.getEst(Xt)
acc = getAcc(Yt, Y_est)
print('Training set accuracy: {:.5f}'.format(acc))
Y_est = dTree.getEst(Xtest)
acc = getAcc(Ytest, Y_est)
print('Test accuracy: {:.5f}'.format(acc))

dTree = decisionTree(XYtrain, maxDepth=7)
Y_est = dTree.getEst(Xt)
acc = getAcc(Yt, Y_est)
print('Training set accuracy: {:.5f}'.format(acc))
Y_est = dTree.getEst(Xtest)
acc = getAcc(Ytest, Y_est)
print('Test accuracy: {:.5f}'.format(acc))
```

```
  dTree = decisionTree(XYtrain, maxDepth=8)
  Y_est = dTree.getEst(Xt)
  acc = getAcc(Yt, Y_est)
  print('Training set accuracy: {:.5f}'.format(acc))
  Y_est = dTree.getEst(Xtest)
  acc = getAcc(Ytest, Y_est)
  print('Test accuracy: {:.5f}'.format(acc))

  dTree = decisionTree(XYtrain, maxDepth=9)
  Y_est = dTree.getEst(Xt)
  acc = getAcc(Yt, Y_est)
  print('Training set accuracy: {:.5f}'.format(acc))
  Y_est = dTree.getEst(Xtest)
  acc = getAcc(Ytest, Y_est)
  print('Test accuracy: {:.5f}'.format(acc))


  '''
  plt.style.use('ggplot')
  plt.plot(range(len(pOR.accHist)), pOR.accHist, label='OR Perceptron Taining Ac
c')
  #plt.show()
  #figOR = plt.figure()
  #ax = figOR.add_subplot(111, projection='3d')



  print("\n\n")
  print('-->> Training and testing with XOR')
  xorX, xorY = get_data(dataXOR)
  pXOR = perceptron( 0.1, 1000, True, True)
  pXOR.train(xorX,xorY)
  #plt.style.use('fivethirtyeight')

  plt.plot(range(len(pXOR.accHist)), pXOR.accHist, label='XOR Perceptron Taining
 Acc')

  plt.xlabel('Training iterations')
  plt.ylabel('Training accuracy [%]')
  plt.title('OR vs. XOR training accuracy')
  plt.legend()
  #plt.grid(True)
  #plt.tight_layout()
  plt.show()
  '''
```