CS6375.004 - MACHINE LEARNING - PROGRAMMING ASSIGNMENT 2

SATWIK ARVAPALLI(**SXA220012**) & ABHINAV YALAMADDI(**YXA210040**)

```
import numpy as np
import math
from matplotlib import pyplot as plt
import pandas as pd


def partition(x):
    """
    Partition the column vector x into subsets indexed by its unique values (v1, ... v

    Returns a dictionary of the form
    { v1: indices of x == v1,
      v2: indices of x == v2,
      ...
      vk: indices of x == vk }, where [v1, ... vk] are all the unique values in the ve

   """
    partitionVector = {}
    for i in x:
        partitionVector[i] = []

    for i in range(len(x)):
        k=partitionVector[x[i]]
        k.append(i)

    return partitionVector

    raise Exception('Function not yet implemented!')


def entropy(y, weight):
    """
    Compute the entropy of a vector y by considering the counts of the unique values (

    Returns the entropy of z: H(z) = p(z=v1) log2(p(z=v1)) + ... + p(z=vk) log2(p(z=vk
    """
    entropy = 0.0
    p_of_y = 0.0

    class_0 = np.sum(weight[y == 0])
    class_1 = np.sum(weight[y == 1])
    class_sum = class_0 + class_1
    partition_of_y = partition(y)

    probability_of_class_0 = class_0 / class_sum
    probability_of_class_1 = class_1 / class_sum
```

```
    # if probability_of_class_0:
    #   entropy += probability_of_class_0 * math.log2(probability_of_class_0) * -1
    # if probability_of_class_1:
    #   entropy += probability_of_class_1 * math.log2(probability_of_class_1) * -1

    for key in partition_of_y:
        ind = partition_of_y[key]
        n = len(partition_of_y[key])
        sum_of_weights = 0
        for k in partition_of_y[key]:
          sum_of_weights = sum_of_weights + weight[k]
        p_of_y = sum_of_weights/(np.sum(weight))
        entropy +=  (-1*p_of_y*(math.log(p_of_y,2)))
    return entropy

    raise Exception('Function not yet implemented!')



def mutual_information(x, y, weight):
    """
    Compute the mutual information between a data column (x) and the labels (y). The c
    over all the examples (n x 1). Mutual information is the difference between the er
    the weighted-average entropy of EACH possible split.

    Returns the mutual information: I(x, y) = H(y) - H(y | x)
    """
    h_of_y = entropy(y, weight)
    partition_of_x = partition(x)
    h_of_yx=0.00

    for key in partition_of_x:
      wt = []
      temp = []
      for i in partition_of_x[key]:
          temp.append(y[i])
      for i in partition_of_x[key]:
          wt.append(weight[i])
      h_of_key = entropy(temp, wt)
      #p_of_key = len(partition_of_x[key])/len(x)
      #(np.sum(weight[indices])/np.sum(weight))
      # sum_of_weights = 0.00
      # for k in partition_of_x[key]:
      #   sum_of_weights = sum_of_weights + weight[k]
      p_of_key = np.sum(wt)/(np.sum(weight))


      h_of_yx = h_of_yx + (p_of_key * h_of_key)

    i_of_xy = h_of_y - h_of_yx

    return i_of_xy
```

```
    raise Exception('Function not yet implemented!')



def id3(x, y, weight, used_attribute_pairs=None, depth=0, max_depth=3):
    """
    Implements the classical ID3 algorithm given training data (x), training labels (y
    attribute-value pairs to consider. This is a recursive algorithm that depends on t
        1. If the entire set of labels (y) is pure (all y = only 0 or only 1), then re
        2. If the set of attribute-value pairs is empty (there is nothing to split on)
           value of y (majority label)
        3. If the max_depth is reached (pre-pruning bias), then return the most common
    Otherwise the algorithm selects the next best attribute-value pair using INFORMATI
    and partitions the data set based on the values of that attribute before the next

    The tree we learn is a BINARY tree, which means that every node has only two branc
    to be chosen from among all possible attribute-value pairs. That is, for a problem
    (taking values a, b, c) and x2 (taking values d, e), the initial attribute value p
    attributes with their corresponding values:
    [(x1, a),
     (x1, b),
     (x1, c),
     (x2, d),
     (x2, e)]
     If we select (x2, d) as the best attribute-value pair, then the new decision node
     the attribute-value pair (x2, d) is removed from the list of attribute_value_pair

    The tree is stored as a nested dictionary, where each entry is of the form
                    (attribute_index, attribute_value, True/False): subtree
    * The (attribute_index, attribute_value) determines the splitting criterion of the
    indicates that we test if (x4 == 2) at the current node.
    * The subtree itself can be nested dictionary, or a single label (leaf node).
    * Leaf nodes are (majority) class labels

    Returns a decision tree represented as a nested dictionary, for example
    {(4, 1, False):
        {(0, 1, False):
            {(1, 1, False): 1,
             (1, 1, True): 0},
         (0, 1, True):
            {(1, 1, False): 0,
             (1, 1, True): 1}},
     (4, 1, True): 1}
    """
    tree = {}

    if len(x)==0 or len(y)==0:
        return

    if(len(partition(y)) < 2):
        return y[0]
```

```
        return y[0]

    if used_attribute_pairs != None and len(used_attribute_pairs) == 0:
      true_count=0
      false_count=0
      for i in range(len(y)):
          if y[i] == 1:
            true_count=true_count+1
          else:
            false_count=false_count+1
      if true_count > false_count:
          return 1
      else:
        return 0

    if depth==max_depth:
        true_count=0
        false_count=0
        for i in range(len(y)):
            if y[i] == 1:
                true_count=true_count+1
            else:
                false_count=false_count+1
        if true_count > false_count:
            return 1
        else:
          return 0


    mutual_info = {}
    shape1=np.shape(x)
    for j in range(0,shape1[1]):
        xi=[]
        for k in range(0,shape1[0]):
            xi.append(x[k][j])
        temp_part = partition(xi)
        for key in temp_part:
            partition_on_key = [0 for it in range(0, len(xi))]
            for i in temp_part[key]:
              partition_on_key[i] = 1

            temp = mutual_information(partition_on_key, y, weight)
            mutual_info[(j,key)] = temp

    if used_attribute_pairs == None:
      used_attribute_pairs = list(mutual_info.keys())
    else:
      for key in list(mutual_info.keys()):
        if key not in used_attribute_pairs:
          mutual_info.pop(key)

    if len(mutual_info) == 0:
```

```
        return

    xi_to_partition = max(mutual_info, key = mutual_info.get)

    mutual_info_of_xy = max(mutual_info, default=0.0)

    y_true=[]
    y_false=[]
    x_true=[]
    x_false=[]
    weight_of_true = []
    weight_of_false = []
    for i in range(len(y)):
        if (x[i][xi_to_partition[0]] == xi_to_partition[1]):
            y_true.append(y[i])
            x_true.append(x[i])
            weight_of_true.append(weight[i])
        else:
            y_false.append(y[i])
            x_false.append(x[i])
            weight_of_false.append(weight[i])



    updated_pairs=used_attribute_pairs[:]
    updated_pairs.remove(xi_to_partition)

    #if xi_to_partition in used_attribute_pairs: del used_attribute_pairs[xi_to_partit

    tree[(xi_to_partition[0],xi_to_partition[1],True)]=id3(x_true,y_true,weight_of_tru

    tree[(xi_to_partition[0],xi_to_partition[1], False)]=id3(x_false,y_false,weight_of


    return tree

   # raise Exception('Function not yet implemented!')


def compute_error(y_true, y_pred):
    """
    Computes the average error between the true labels (y_true) and the predicted labe

    Returns the error = (1/n) * sum(y_true != y_pred)
    """
    count=0
    n = len(y_true)
    for i in range(n):
        if y_true[i]!=y_pred[i]:
            count = count + 1
    return count/n
```

```
        raise Exception('Function not yet implemented!')

    def bagging(x,y,max_depth,num_trees):
        import random
        random.seed(0)
        lenX = len(x)
        sequence = list(range(len(x)))
        weight = np.ones(lenX)
        hypothesis = {}
        alpha = 1

        for tn in range(num_trees):
            indices = random.choices(sequence,k=lenX)

            decision_tree = id3(x[indices], y[indices],weight, max_depth=max_depth)
            hypothesis[tn] = (alpha,decision_tree)

        return hypothesis


    def boosting(x,y,max_depth,num_stumps):
        lenX = len(x)
        hypothesis = {}
        weight = np.ones(lenX)/lenX

        for ns in range(num_stumps):
            decision_tree = id3(x, y, weight, max_depth=max_depth)

            y_pred = [predict_example_base_learner(xe, decision_tree) for xe in x]

            #y_pred1 = [predict_example(xe, decision_tree, "boosting") for xe in x]

            epsilon = (np.dot(np.absolute(y - y_pred), weight))/ np.sum(weight)
            alpha = 0.5 * (np.log(((1 - epsilon) / epsilon)))
            # print(alpha)
            # print(epsilon)
            for i in range(len(y_pred)):
                if y_pred[i] == y[i]:
                    weight[i] *= np.exp(-alpha)
                else:
                    weight[i] *= np.exp(alpha)

            #weight /= 2 * np.sqrt(epsilon * (1 - epsilon))
            hypothesis[ns] = (alpha, decision_tree)


        return hypothesis


    def predict_example(x,h_ens,ensemble_type):
        """
```

```
    h_ens is an ensemble of weighted hypotheses.
    The ensemble is represented as an array of pairs [(alpha_i, h_i)], where each hypo
    are represented by the pair: (alpha_i, h_i).
    """
    if ensemble_type == "bagging":
        predictions = []
        for k in h_ens:
            y_pred = predict_example_base_learner(x,h_ens[k][1])
            predictions.append(y_pred)

        predict_egz = max(predictions, key=predictions.count)
        return predict_egz
    else:
        predictions = []
        sum_alpha = 0

        for y in h_ens:
            alpha, tree = h_ens[y]
            tst_pred = predict_example_base_learner(x, tree)

            predictions.append(tst_pred*alpha)
            sum_alpha += alpha
        predict_egz = np.sum(predictions) / sum_alpha
        if predict_egz >= 0.5:
            return 1
        else:
            return 0

def predict_example_base_learner(x, tree):
    """
    Predicts the classification label for a single example x using tree by recursively
    a label/leaf node is reached.

    Returns the predicted label of x according to tree
    """
    if type(tree) is not dict:
      return tree

    if x[list(tree.keys())[0][0]]==list(tree.keys())[0][1]:
        temp = True
    else:
        temp = False

    if type(tree[(list(tree.keys())[0][0],list(tree.keys())[0][1],temp)]) is dict:
        return predict_example_base_learner(x,tree[(list(tree.keys())[0][0],list(tree.

    return tree[(list(tree.keys())[0][0],list(tree.keys())[0][1],temp)]

    raise Exception('Function not yet implemented!')
```

```
def visualize(tree, depth=0):
    """
    Pretty prints (kinda ugly, but hey, it's better than nothing) the decision tree to
    print the raw nested dictionary representation.
    DO NOT MODIFY THIS FUNCTION!
    """

    if depth == 0:
        print('TREE')

    for index, split_criterion in enumerate(tree):
        sub_trees = tree[split_criterion]

        # Print the current node: split criterion
        print('|\t' * depth, end='')
        print('+-- [SPLIT: x{0} = {1} {2}]'.format(split_criterion[0], split_criterion

        # Print the children
        if type(sub_trees) is dict:
            visualize(sub_trees, depth + 1)
        else:
            print('|\t' * (depth + 1), end='')
            print('+-- [LABEL = {0}]'.format(sub_trees))

def confusion_mat(y_true,y_pred):
    tp,tn,fp,fn = 0,0,0,0
    for i in range(len(y_pred)):
        if y_pred[i] == 1 and y_true[i] == 1:
            tp += 1
        elif y_pred[i] == 0 and y_true[i] == 0:
            tn += 1
        elif y_pred[i] == 1 and y_true[i] == 0:
            fp += 1
        elif y_pred[i] == 0 and y_true[i] == 1:
            fn += 1
    # mat = np.array([tn,fp,fn,tp])     #This is simlar to sklearn convention
    mat = np.array([tp,fn,fp,tn]).reshape(2,2)      #This is for current assignment
    print("\t\tClassifier Prediction")
    print("\t\t\tPositive\tNegative")
    print("Actual | Positive\t",mat[0][0],"\t\t",mat[0][1])
    print("Value  | Negative\t",mat[1][0],"\t\t",mat[1][1])


if __name__ == '__main__':
    # Load the training data
    M = np.genfromtxt('./mushroom.train', missing_values=0, skip_header=0, delimiter='
    ytrn = M[:, 0]
    Xtrn = M[:, 1:]

    # Load the test data
    M = np.genfromtxt('./mushroom.test', missing_values=0, skip_header=0, delimiter=',
    ytst = M[:, 0]
```

```
    Xtst = M[:, 1:]

    testing = []
    # Bagging
    print("----------------------BAGGING----------------------")
    for depth in [3,5]:
      for bag_size in [10,20]:
        # Learn a decision tree of depth dep
        print("Depth: ", depth, "Bag Size: ", bag_size)
        ensemble_bag = bagging(Xtrn,ytrn,depth,bag_size)

        # Compute the test error
        y_pred = [predict_example(x, ensemble_bag, "bagging") for x in Xtst]
        # print(y_pred)

        tst_err = compute_error(ytst, y_pred)
        testing.append(tst_err*100)

        # # print('depth=',depth, end=" ")
        print('Test Error = {0:4.2f}%'.format(tst_err * 100))
        confusion_mat(ytst,y_pred)


    # Boosting
    print("----------------------Boosting----------------------")
    for depth in [1,2]:
      for bag_size in [20,40]:
        # Learn a decision tree of depth dep
        print("Depth: ", depth, "bag_size: ", bag_size)
        ensemble_boost = boosting(Xtrn,ytrn,depth,bag_size)

        # # Compute the test error
        y_pred = [predict_example(x, ensemble_boost, "boosting") for x in Xtst]
        # # print(y_pred)

        tst_err = compute_error(ytst, y_pred)
        testing.append(tst_err*100)
        # print('depth=',depth, end=" ")
        print('Test Error = {0:4.2f}%'.format(tst_err * 100))
        confusion_mat(ytst,y_pred)
```

```
----------------------BAGGING----------------------
Depth:  3 Bag Size:  10
Test Error = 4.23%
                Classifier Prediction
                        Positive        Negative
Actual | Positive        815             29
Value  | Negative        57              1130
Depth:  3 Bag Size:  20
Test Error = 4.23%
                Classifier Prediction
```

```
                        Positive        Negative
Actual | Positive        815             29
Value  | Negative        57              1130
Depth:  5 Bag Size:  10
Test Error = 0.20%
                Classifier Prediction
                        Positive        Negative
Actual | Positive        844             0
Value  | Negative        4               1183
Depth:  5 Bag Size:  20
Test Error = 0.20%
                Classifier Prediction
                        Positive        Negative
Actual | Positive        844             0
Value  | Negative        4               1183
---------------------Boosting---------------------
Depth:  1 bag_size:  20
Test Error = 11.18%
                Classifier Prediction
                        Positive        Negative
Actual | Positive        793             51
Value  | Negative        176             1011
Depth:  1 bag_size:  40
Test Error = 11.18%
                Classifier Prediction
                        Positive        Negative
Actual | Positive        793             51
Value  | Negative        176             1011
Depth:  2 bag_size:  20
Test Error = 6.40%
                Classifier Prediction
                        Positive        Negative
Actual | Positive        823             21
Value  | Negative        109             1078
Depth:  2 bag_size:  40
Test Error = 6.40%
                Classifier Prediction
                        Positive        Negative
Actual | Positive        823             21
Value  | Negative        109             1078
```

```
sklearn_training_set = ["train"]
sklearn_testing_set = ["test"]
sklearn_names_set = ["mushroom data"]


for train,test,name in zip(sklearn_training_set,sklearn_testing_set,sklearn_names_set)
  # Load the training data
    M = np.genfromtxt('./mushroom.train', missing_values=0, skip_header=0, delimiter='
    ytrn = M[:, 0]
    Xtrn = M[:, 1:]

    # Load the test data
    M = np.genfromtxt('./mushroom.test', missing_values=0, skip_header=0, delimiter=',
```

```
    ytst = M[:, 0]
    Xtst = M[:, 1:]

from sklearn import tree
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import confusion_matrix,accuracy_score

max_depth = [3,5]
bag_size = [10,20]

for md in max_depth:
  for bs in bag_size:

    print("Bagging : max_depth =",md,"bag_size = ",bs)
    clf = BaggingClassifier(tree.DecisionTreeClassifier(random_state = 42, max_depth =
    clf = clf.fit(Xtrn, ytrn)
    y_pred = clf.predict(Xtst)
    accuracy = accuracy_score(ytst,y_pred)
    print("Test Error = ", (1-accuracy)*100)
    print("Confusion matrix: \n",confusion_matrix(ytst,y_pred))
    # print("\n\n")

max_depth = [1,2]
bag_size = [20,40]

for md in max_depth:
  for bs in bag_size:

    print("AdaBoost : max_depth =",md,"bag_size = ",bs)
    clf = AdaBoostClassifier(tree.DecisionTreeClassifier(random_state = 42, max_depth
    clf = clf.fit(Xtrn, ytrn)
    y_pred = clf.predict(Xtst)
    accuracy = accuracy_score(ytst,y_pred)
    print("Test Error = ", (1-accuracy)*100)
    print("Confusion matrix: \n",confusion_matrix(ytst,y_pred))
```

```
Bagging : max_depth = 3 bag_size =  10
Test Error =  4.382077794190053
Confusion matrix:
 [[1102   85]
 [   4  840]]
Bagging : max_depth = 3 bag_size =  20
Test Error =  4.382077794190053
Confusion matrix:
 [[1102   85]
 [   4  840]]
Bagging : max_depth = 5 bag_size =  10
Test Error =  1.1816838995568735
Confusion matrix:
 [[1187    0]
 [  24  820]]
Bagging : max_depth = 5 bag_size =  20
```

```
Test Error =  1.1816838995568735
Confusion matrix:
 [[1187    0]
 [  24  820]]
AdaBoost : max_depth = 1 bag_size =  20
Test Error =  0.1969473165928104
Confusion matrix:
 [[1185    2]
 [   2  842]]
AdaBoost : max_depth = 1 bag_size =  40
Test Error =  0.0
Confusion matrix:
 [[1187    0]
 [   0  844]]
AdaBoost : max_depth = 2 bag_size =  20
Test Error =  0.0
Confusion matrix:
 [[1187    0]
 [   0  844]]
AdaBoost : max_depth = 2 bag_size =  40
Test Error =  0.0
Confusion matrix:
 [[1187    0]
 [   0  844]]
```