sxr_190067_Code

February 27, 2021

```
decision_tree.py
———
Licensing Information: You are free to use or extend these projects for
personal and educational purposes provided that (1) you do not distribute
or publish solutions, (2) you retain this notice, and (3) you provide clear
attribution to UT Dallas, including a link to http://cs.utdallas.edu.

This file is part of Homework for CS6375: Machine Learning.
Gautam Kunapuli (gautam.kunapuli@utdallas.edu)
Sriraam Natarajan (sriraam.natarajan@utdallas.edu),
Anjum Chida (anjum.chida@utdallas.edu)


INSTRUCTIONS:
————
1. This file contains a skeleton for implementing the ID3 algorithm for
Decision Trees. Insert your code into the various functions that have the
comment “INSERT YOUR CODE HERE”.
```

[13]:
```python
import numpy as np
import os
import graphviz

import math
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import pandas as pd
from sklearn import tree
import pydotplus
from IPython.display import Image

from sklearn.model_selection import train_test_split
```

[14]:
```python
def partition(x):
    """
    Partition the column vector x into subsets indexed by its unique values (v1,␣
 ↪... vk)

    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 vector z.
    """

    # INSERT YOUR CODE HERE
    #raise Exception('Function not yet implemented!')

    dictionary={}
    xLen = len(x)
    for i in range(0,xLen):
        item=x[i]
        if item in dictionary:
            dictionary[x[i]].append(i);
        else:
            dictionary[x[i]]=[]
            dictionary[x[i]].append(i)

    return dictionary
```

[15]:
```python
def entropy(y):
    """
    Compute the entropy of a vector y by considering the counts of the unique␣
 ↪values (v1, ... vk), in z
```

```
    Returns the entropy of z: H(z) = p(z=v1) log2(p(z=v1)) + ... + p(z=vk)␣
↪log2(p(z=vk))
    """

    # INSERT YOUR CODE HERE
    # raise Exception('Function not yet implemented!')
    entropy=0
    count=0
    y = np.array(y)
    yLen = len(y)
    for i in set(y):
        P = (y == i).sum()/yLen
        entropy = entropy + P*math.log2(P)
    return -entropy
```

[16]:
```
def mutual_information(x, y):
    """
    Compute the mutual information between a data column (x) and the labels (y).␣
↪The data column is a single attribute
    over all the examples (n x 1). Mutual information is the difference between␣
↪the entropy BEFORE the split set, and
    the weighted-average entropy of EACH possible split.

    Returns the mutual information: I(x, y) = H(y) - H(y | x)
    """

    # INSERT YOUR CODE HERE
    # raise Exception('Function not yet implemented!')
    yEnt = entropy(y)
    yx_Ent=0
    X = partition(x)

    for v in X:
        y_temp=[]
        for i in X[v]:
            y_temp.append(y[i])
        P=x.count(v)/len(x)
        yx_Ent= yx_Ent + P*entropy(y_temp)

    H = yEnt-yx_Ent
    return H
```

[17]:
```
def id3(x, y, attribute_value_pairs=None, depth=0, max_depth=5):
    """
    Implements the classical ID3 algorithm given training data (x), training␣
↪labels (y) and an array of
```

```
    attribute-value pairs to consider. This is a recursive algorithm that␣
↪depends on three termination conditions
        1. If the entire set of labels (y) is pure (all y = only 0 or only 1),␣
↪then return that label
        2. If the set of attribute-value pairs is empty (there is nothing to␣
↪split on), then return the most common
           value of y (majority label)
        3. If the max_depth is reached (pre-pruning bias), then return the most␣
↪common value of y (majority label)
    Otherwise the algorithm selects the next best attribute-value pair using␣
↪INFORMATION GAIN as the splitting criterion
    and partitions the data set based on the values of that attribute before the␣
↪next recursive call to ID3.

    The tree we learn is a BINARY tree, which means that every node has only two␣
↪branches. The splitting criterion has
    to be chosen from among all possible attribute-value pairs. That is, for a␣
↪problem with two features/attributes x1
    (taking values a, b, c) and x2 (taking values d, e), the initial attribute␣
↪value pair list is a list of all pairs of
    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 becomes: [ (x2 == d)? ] and
     the attribute-value pair (x2, d) is removed from the list of␣
↪attribute_value_pairs.

    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 current node. For example, (4, 2)
    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}
    """

    # INSERT YOUR CODE HERE. NOTE: THIS IS A RECURSIVE FUNCTION.
    # raise Exception('Function not yet implemented!')

    if attribute_value_pairs == None:
        attribute_value_pairs=[]
        for i in range(0,x.shape[1]):
            for v in set(x[:,i]):
                attribute_value_pairs.append((i,v))


    if len(attribute_value_pairs)==0 or depth== max_depth:
        frequency = np.bincount(np.array(y))
        return np.argmax(frequency)
    elif all(z==y[0] for z in y):
        return y[0]
    else:
        maximum=0
        xLen = len(x)
        for attr in attribute_value_pairs:
            x_temp = []
            i = attr[0]

            for j in range(0,xLen):
                val = x[j][i]
                if val==attr[1]:
                    x_temp.append(1)
                else:
                    x_temp.append(0)
            InfoG = mutual_information(x_temp,y)
            if InfoG >= maximum:
                maximum = InfoG
                bestsplit = attr

        val = bestsplit[1]
        i = bestsplit[0]
        x_temp=[]

        for j in range(0,xLen):
            x_temp.append(x[j][i])
        X=partition(x_temp)
        bestlist=X[val]

        true_X=[]
```

```
        false_X=[]
        true_Y=[]
        false_Y=[]

        for i in range(0,len(x)):
            temp_array = np.asarray(x[i])
            if i in bestlist:
                true_X.append(temp_array)
                true_Y.append(y[i])
            else:
                false_X.append(temp_array)
                false_Y.append(y[i])

        true_AVP = attribute_value_pairs.copy()
        false_AVP = attribute_value_pairs.copy()

        true_AVP.remove(bestsplit)
        false_AVP.remove(bestsplit)

        tree = {(bestsplit[0],bestsplit[1],True):
↪id3(true_X,true_Y,true_AVP,depth+1,max_depth),(bestsplit[0],bestsplit[1],False):
↪id3(false_X,false_Y,false_AVP,depth+1,max_depth)}

        return tree
```

[18]:
```
def predict_example(x, tree):
    """
    Predicts the classification label for a single example x using tree by␣
↪recursively descending the tree until
    a label/leaf node is reached.

    Returns the predicted label of x according to tree
    """

    # INSERT YOUR CODE HERE. NOTE: THIS IS A RECURSIVE FUNCTION.
    # raise Exception('Function not yet implemented!')

    try:
        len(tree.keys())
    except Exception as e:
        return tree

    item = list(tree.keys())[0]

    if x[item[0]] == item[1]:
        return predict_example(x, tree[item[0],item[1],True])
    else:
```

```
        return predict_example(x, tree[item[0],item[1],False])
```

[19]:
```
def compute_error(y_true, y_pred):
    """
    Computes the average error between the true labels (y_true) and the␣
 ↪predicted labels (y_pred)

    Returns the error = (1/n) * sum(y_true != y_pred)
    """

    # INSERT YOUR CODE HERE
    # raise Exception('Function not yet implemented!')

    errorCount=0
    yLen = len(y_true)
    for i in range(0, yLen):
        if y_true[i] != y_pred[i]:
            errorCount+=1
    return errorCount/yLen
```

[20]:
```
def pretty_print(tree, depth=0):
    """
    Pretty prints the decision tree to the console. Use print(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[1], split_criterion[2]))

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

[21]:
```
def render_dot_file(dot_string, save_file, image_format='png'):
    """
    Uses GraphViz to render a dot file. The dot file can be generated using
        * sklearn.tree.export_graphviz()' for decision trees produced by␣
↪scikit-learn
        * to_graphviz() (function is in this file) for decision trees produced␣
↪by  your code.
    DO NOT MODIFY THIS FUNCTION!
    """
    if type(dot_string).__name__ != 'str':
        raise TypeError('visualize() requires a string representation of a␣
↪decision tree.\nUse tree.export_graphviz()'
                        'for decision trees produced by scikit-learn and␣
↪to_graphviz() for decision trees produced by'
                        'your code.\n')

    # Set path to your GraphViz executable here
    os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz2.38/bin/'
    graph = graphviz.Source(dot_string)
    graph.format = image_format
    graph.render(save_file, view=True)
```

[22]:
```
def to_graphviz(tree, dot_string='', uid=-1, depth=0):
    """
    Converts a tree to DOT format for use with visualize/GraphViz
    DO NOT MODIFY THIS FUNCTION!
    """

    uid += 1       # Running index of node ids across recursion
    node_id = uid  # Node id of this node

    if depth == 0:
        dot_string += 'digraph TREE {\n'

    for split_criterion in tree:
        sub_trees = tree[split_criterion]
        attribute_index = split_criterion[0]
        attribute_value = split_criterion[1]
        split_decision = split_criterion[2]

        if not split_decision:
            # Alphabetically, False comes first
            dot_string += '    node{0} [label="x{1} = {2}?"];\n'.format(node_id,␣
↪attribute_index, attribute_value)

        if type(sub_trees) is dict:
            if not split_decision:
```

```
                dot_string, right_child, uid = to_graphviz(sub_trees,␣
↪dot_string=dot_string, uid=uid, depth=depth + 1)
                dot_string += '    node{0} -> node{1} [label="False"];\n'.
↪format(node_id, right_child)
            else:
                dot_string, left_child, uid = to_graphviz(sub_trees,␣
↪dot_string=dot_string, uid=uid, depth=depth + 1)
                dot_string += '    node{0} -> node{1} [label="True"];\n'.
↪format(node_id, left_child)

        else:
            uid += 1
            dot_string += '    node{0} [label="y = {1}"];\n'.format(uid,␣
↪sub_trees)
            if not split_decision:
                dot_string += '    node{0} -> node{1} [label="False"];\n'.
↪format(node_id, uid)
            else:
                dot_string += '    node{0} -> node{1} [label="True"];\n'.
↪format(node_id, uid)

    if depth == 0:
        dot_string += '}\n'
        return dot_string
    else:
        return dot_string, node_id, uid


if __name__ == '__main__':

    #b.Learning Curves
    for i in range(1,4):
        testingdatapath = "./monks_data/monks-"+str(i)+".test"
        trainingdatapath = "./monks_data/monks-"+str(i)+".train"


        # Load the training data
        M = np.genfromtxt(trainingdatapath, missing_values=0, skip_header=0,␣
↪delimiter=',', dtype=int)
        ytrn = M[:, 0]
        Xtrn = M[:, 1:]

        # Load the test data
        M = np.genfromtxt(testingdatapath, missing_values=0, skip_header=0,␣
↪delimiter=',', dtype=int)
        ytst = M[:, 0]
```

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

        trnError = {}
        tstError = {}

        for d in range(1, 11):
            # Determine the decision tree of depth d
            decision_tree = id3(Xtrn, ytrn, max_depth=d)

            # Calculating the training error
            trainy_pred = [predict_example(x, decision_tree) for x in Xtrn]
            trn_err = compute_error(ytrn, trainy_pred)

            # Calculating the testing error
            testy_pred = [predict_example(x, decision_tree) for x in Xtst]
            tst_err = compute_error(ytst, testy_pred)

            trnError[d] = trn_err
            tstError[d] = tst_err

        # Below we plot the testing and training error for all the depths
        plt.figure()
        plt.plot(trnError.keys(), trnError.values(), marker='o', linewidth=3,␣
↪markersize=12)
        plt.plot(tstError.keys(), tstError.values(), marker='s', linewidth=3,␣
↪markersize=12)
        plt.xlabel('Depth', fontsize=16)
        plt.ylabel('Training/Test Error', fontsize=16)
        plt.xticks(list(trnError.keys()), fontsize=12)
        plt.legend(['Training Error', 'Test Error'], fontsize=16)
        plt.xscale('log')
        plt.yscale('log')
        plt.title("MONKS-"+str(i))

    #c.Weak Learners
    # Load the training data
    M = np.genfromtxt('./monks_data/monks-1.train', missing_values=0,␣
↪skip_header=0, delimiter=',', dtype=int)
    ytrn = M[:, 0]
    Xtrn = M[:, 1:]

    # loading the testing data
    M = np.genfromtxt('./monks_data/monks-1.test', missing_values=0,␣
↪skip_header=0, delimiter=',', dtype=int)
    ytst = M[:, 0]
    Xtst = M[:, 1:]
```

```python
    tst_err = {}

    for i in range(1, 6, 2):

        # Learn a decision tree of depth 3
        decision_tree = id3(Xtrn, ytrn, max_depth=i)

        # Pretty print it to console
        pretty_print(decision_tree)

        # Visualize the tree and save it as a PNG image
        dot_str = to_graphviz(decision_tree)
        render_dot_file(dot_str, './monks1learn-'+str(i))

        # Compute the test error
        y_pred = [predict_example(x, decision_tree) for x in Xtst]
        tst_err[i] = compute_error(ytst, y_pred)

        print('\nTest Error = {0:4.2f}%.'.format(tst_err[i] * 100))
        print("MONKS Dataset: Confusion matrix for depth ",i )
        print(pd.DataFrame(confusion_matrix(ytst, y_pred), columns=['Predicted␣
↪Positives', 'Predicted Negatives'],
                           index=['True Positives', 'True Negatives']))

    #d.scikit-learn
    for i in range(1,6,2):
        Data_names = ['X1','X2','X3','X4','X5','X6']
        decision_tree = tree.
↪DecisionTreeClassifier(criterion='entropy',max_depth=i)
        decision_tree.fit(Xtrn, ytrn)
        dot_data = tree.export_graphviz(decision_tree, out_file=None,␣
↪feature_names=Data_names,
                                filled=True, rounded=True,␣
↪special_characters=True)
        graph = pydotplus.graph_from_dot_data(dot_data)
        graph.write_png('monks1sklearn-'+str(i)+'.png')
        Image(filename='monks1sklearn-'+str(i)+'.png')
        y_pred = decision_tree.predict(Xtst)
        tst_err[i] = compute_error(ytst, y_pred)

        print('\nTest Error = {0:4.2f}%.'.format(tst_err[i] * 100))
        print("MONKS Dataset: Confusion matrix for depth ",i )
        print(pd.DataFrame(confusion_matrix(ytst, y_pred),columns=['Predicted␣
↪Positives', 'Predicted Negatives'],
                           index=['True Positives', 'True Negatives']))

    #e.Other Data Sets
```

```
    IUData = np.genfromtxt('./DishonestIUData.txt',skip_header=0,delimiter=' ',␣
↪dtype=int)
    X=IUData[:,0:4]
    y=IUData[:,4]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,␣
↪random_state=42)

    #e.1.new data set id3
    for i in range(1, 4, 2):
        decision_tree = id3(X_train, y_train, max_depth=i)
        pretty_print(decision_tree)
        dot_str = to_graphviz(decision_tree)
        render_dot_file(dot_str, './IUDataImg-'+str(i))
        y_pred = [predict_example(x, decision_tree) for x in X_test]

        print("Dishonest Internet Users Dataset: Confusion matrix for depth ",i )
        print(pd.DataFrame(confusion_matrix(y_test, y_pred),columns=['Predicted␣
↪Positives', 'Predicted Negatives'],
                            index=['True Positives', 'True Negatives']
                            ))

    #e.1.new data set scikit-learn
    for i in range(1, 4, 2):
        Data_names = ['X1', 'X2', 'X3', 'X4']
        decision_tree = tree.DecisionTreeClassifier(criterion='entropy',␣
↪max_depth=i)
        decision_tree.fit(X_train, y_train)
        dot_data = tree.export_graphviz(decision_tree, out_file=None,␣
↪feature_names=Data_names,
                                  filled=True, rounded=True,␣
↪special_characters=True)
        graph = pydotplus.graph_from_dot_data(dot_data)
        graph.write_png('IUDatasklearn-'+str(i)+'.png')
        Image(filename='IUDatasklearn-'+str(i)+'.png')
        y_pred = decision_tree.predict(X_test)

        print("Dishonest Internet Users Dataset: Confusion matrix  for depth ",i␣
↪)
        print(pd.DataFrame(confusion_matrix(y_test, y_pred),columns=['Predicted␣
↪Positives', 'Predicted Negatives'],
                            index=['True Positives', 'True Negatives']))
```

```
TREE
+-- [SPLIT: x4 = 1 True]
|       +-- [LABEL = 1]
+-- [SPLIT: x4 = 1 False]
|       +-- [LABEL = 0]
```

```
Test Error = 25.00%.
MONKS Dataset: Confusion matrix for depth  1
                Predicted Positives  Predicted Negatives
True Positives                 216                    0
True Negatives                 108                  108
TREE
+-- [SPLIT: x4 = 1 True]
|       +-- [LABEL = 1]
+-- [SPLIT: x4 = 1 False]
|       +-- [SPLIT: x0 = 1 True]
|       |       +-- [SPLIT: x1 = 1 True]
|       |       |       +-- [LABEL = 1]
|       |       +-- [SPLIT: x1 = 1 False]
|       |       |       +-- [LABEL = 0]
|       +-- [SPLIT: x0 = 1 False]
|       |       +-- [SPLIT: x1 = 1 True]
|       |       |       +-- [LABEL = 0]
|       |       +-- [SPLIT: x1 = 1 False]
|       |       |       +-- [LABEL = 1]

Test Error = 16.67%.
MONKS Dataset: Confusion matrix for depth  3
                Predicted Positives  Predicted Negatives
True Positives                 144                   72
True Negatives                   0                  216
TREE
+-- [SPLIT: x4 = 1 True]
|       +-- [LABEL = 1]
+-- [SPLIT: x4 = 1 False]
|       +-- [SPLIT: x0 = 1 True]
|       |       +-- [SPLIT: x1 = 1 True]
|       |       |       +-- [LABEL = 1]
|       |       +-- [SPLIT: x1 = 1 False]
|       |       |       +-- [LABEL = 0]
|       +-- [SPLIT: x0 = 1 False]
|       |       +-- [SPLIT: x1 = 1 True]
|       |       |       +-- [LABEL = 0]
|       |       +-- [SPLIT: x1 = 1 False]
|       |       |       +-- [SPLIT: x4 = 3 True]
|       |       |       |       +-- [SPLIT: x1 = 3 True]
|       |       |       |       |       +-- [LABEL = 0]
|       |       |       |       +-- [SPLIT: x1 = 3 False]
|       |       |       |       |       +-- [LABEL = 1]
|       |       |       +-- [SPLIT: x4 = 3 False]
|       |       |       |       +-- [SPLIT: x3 = 1 True]
|       |       |       |       |       +-- [LABEL = 1]
|       |       |       |       +-- [SPLIT: x3 = 1 False]
```

```
|       |       |       |       |       +-- [LABEL = 1]

Test Error = 16.67%.
MONKS Dataset: Confusion matrix for depth  5
                Predicted Positives  Predicted Negatives
True Positives                  156                   60
True Negatives                   12                  204

Test Error = 25.00%.
MONKS Dataset: Confusion matrix for depth  1
                Predicted Positives  Predicted Negatives
True Positives                  216                    0
True Negatives                  108                  108

Test Error = 16.67%.
MONKS Dataset: Confusion matrix for depth  3
                Predicted Positives  Predicted Negatives
True Positives                  144                   72
True Negatives                    0                  216

Test Error = 16.67%.
MONKS Dataset: Confusion matrix for depth  5
                Predicted Positives  Predicted Negatives
True Positives                  168                   48
True Negatives                   24                  192
TREE
+-- [SPLIT: x3 = 3 True]
|       +-- [LABEL = 0]
+-- [SPLIT: x3 = 3 False]
|       +-- [LABEL = 1]
Dishonest Internet Users Dataset: Confusion matrix for depth  1
                Predicted Positives  Predicted Negatives
True Positives                   11                   21
True Negatives                    0                   75
TREE
+-- [SPLIT: x3 = 3 True]
|       +-- [LABEL = 0]
+-- [SPLIT: x3 = 3 False]
|       +-- [SPLIT: x2 = 2 True]
|       |       +-- [SPLIT: x3 = 4 True]
|       |       |       +-- [LABEL = 0]
|       |       +-- [SPLIT: x3 = 4 False]
|       |       |       +-- [LABEL = 1]
|       +-- [SPLIT: x2 = 2 False]
|       |       +-- [SPLIT: x0 = 4 True]
|       |       |       +-- [LABEL = 1]
|       |       +-- [SPLIT: x0 = 4 False]
|       |       |       +-- [LABEL = 1]
```

```
Dishonest Internet Users Dataset: Confusion matrix for depth  3
                Predicted Positives  Predicted Negatives
True Positives                   23                    9
True Negatives                    0                   75
Dishonest Internet Users Dataset: Confusion matrix  for depth  1
                Predicted Positives  Predicted Negatives
True Positives                    0                   32
True Negatives                    0                   75
Dishonest Internet Users Dataset: Confusion matrix  for depth  3
                Predicted Positives  Predicted Negatives
True Positives                   23                    9
True Negatives                    0                   75
```

output_10_1.png

output_10_2.png

output_10_3.png

[ ]: