



 [http://cs.utdallas.edu](http://cs.utdallas.edu/)



















**import** numpy **as** np

**import** os **import** graphviz **import** math

**import** matplotlib.pyplot **as** plt



**def** partition(x): """

Partition the column vector x into subsets indexed by its unique values ( 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 """

indices **=** {}

**for** value **in** range(len(x)):

indices[x[value]] **=** indices**.**get(x[value], []) **+** [value]

**return** indices

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





partition([1,5,3,1,7,5,3,5,3,2])





**def** entropy(y): """

Compute the entropy of a vector y by considering the counts of the unique Returns the entropy of z: H(z) = p(z=v1) log2(p(z=v1)) + ... + p(z=vk) lo """

values, count **=** np**.**unique(y, return\_counts**=**"True") h **=** 0

**for** c **in** count:

h **+=** ((c**/**len(y))**\*** math**.**log(c**/**len(y), 2))

**return -**h

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



entropy([1,0,1,1,0,1,0,1,1])



**def** mutual\_information(x, y): """

Compute the mutual information between a data column (x) and the labels ( over all the examples (n x 1). Mutual information is the difference betwe the weighted-average entropy of EACH possible split.

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

e\_y **=** entropy(y) d **=** {}

values, count **=** np**.**unique(x, return\_counts**=**"True")

**for** v **in** values: new\_y1 **=** [] new\_y2 **=** []

**for** i **in** range(len(y)):

**if** (x[i] **==** v): new\_y1**.**append(y[i])

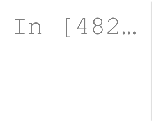
**else**:

new\_y2**.**append(y[i])

d[v] **=** e\_y **-** (((len(new\_y1)**/**len(y))**\***entropy(new\_y1)) **+** ((len(new\_y2)**/ return** d

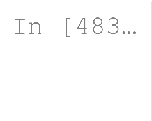
**raise** Exception('Function not yet implemented!')





**def** info\_gain\_specified\_value(x, v, y): d **=** mutual\_information(x,y)

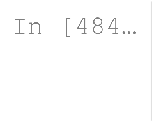
**return** d**.**get(v)



m1 **=** ['a','a','b','c','a','b','c','a','b','c'] m2 **=** [1,0,1,0,0,0,1,1,1,0]

mutual\_information(m1, m2)



m1 **=** ['a','a','b','c','a','b','c','a','b','c'] m2 **=** [1,0,1,0,0,0,1,1,1,0]

info\_gain\_specified\_value(m1, 'b', m2)



**def** id3(x, y, attribute\_value\_pairs**=None**, depth**=**0, max\_depth**=**5): """

Implements the classical ID3 algorithm given training data (x), training attribute-value pairs to consider. This is a recursive algorithm that dep

1. If the entire set of labels (y) is pure (all y = only 0 or only 1)
2. If the set of attribute-value pairs is empty (there is nothing to value of y (majority label)
3. If the max\_depth is reached (pre-pruning bias), then return the mo Otherwise the algorithm selects the next best attribute-value pair using and partitions the data set based on the values of that attribute before The tree we learn is a BINARY tree, which means that every node has only to be chosen from among all possible attribute-value pairs. That is, for (taking values a, b, c) and x2 (taking values d, e), the initial attribut 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 deci the attribute-value pair (x2, d) is removed from the list of attribute\_v The tree is stored as a nested dictionary, where each entry is of the for

(attribute\_index, attribute\_value, True/False): subtree

* The (attribute\_index, attribute\_value) determines the splitting criteri indicates that we test if (x4 == 2) at the current node.
* The subtree itself can be nested dictionary, or a single label (leaf no
* 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} """

x **=** np**.**asarray(x) y **=** np**.**asarray(y)

*#populate attribute\_value\_pairs*

**if** depth **==** 0: attribute\_value\_pairs **=** [] **for** attr **in** range(len(x[0])):

**for** val **in** np**.**unique(np**.**transpose(x[:,attr])): pair **=** (attr, val) attribute\_value\_pairs**.**append(pair)

*#checks if entire set of labels(y) is pure*

**if** len(np**.**unique(y)) **==** 1:

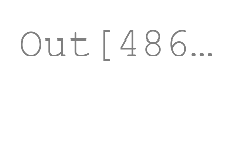
**return** y[0]

*#checks if attr val pairs are empty, or if at max depth. In both cases, r*

**if** (**not** attribute\_value\_pairs) **or** depth **==** max\_depth **or** len(x) **==** 0: values, counts **=** np**.**unique(y, return\_counts **=** "True")

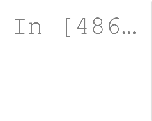
**if** counts**.**size **==** 0:

## return 0









x **=** [[0,0,0],[1,2,2],[2,0,1],[0,1,1],[2,2,2],[1,0,1],[1,1,0],[3,2,1],[1,3,3],

y **=** [0,0,1,1,1,0,0,1,1,1]

id3(x, y)



**def** predict\_example(x, tree): """

Predicts the classification label for a single example x using tree by re a label/leaf node is reached.

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

root **=** list(tree**.**keys()) attr **=** root[0][0]

val **=** root[0][1]

**if** x[attr] **==** val:

subTree **=** tree**.**get((attr, val, **True**)) **else**:

subTree **=** tree**.**get((attr, val, **False**))

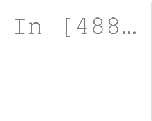
**if** subTree **==** 0 **or** subTree **==** 1:

**return** subTree

**else**:

**return** predict\_example(x, subTree)

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



x **=** [[0,0,0],[1,2,2],[2,0,1],[0,1,1],[2,2,2],[1,0,1],[1,1,0],[3,2,1],[1,3,3],

y **=** [0,0,1,1,1,0,0,1,1,1]

predict\_example([1,2,1],id3(x, y))



**def** compute\_error(y\_true, y\_pred): """

Computes the average error between the true labels (y\_true) and the predi Returns the error = (1/n) \* sum(y\_true != y\_pred)

"""

n **=** len(y\_true) sum **=** 0

**for** i **in** range(n):

**if** (y\_true[i] **!=** y\_pred[i]): sum **+=** 1

**return** (1**/**n) **\*** sum

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



**def** pretty\_print(tree, depth**=**0): """

Pretty prints the decision tree to the console. Use print(tree) to print """

**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\_

*# 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))



**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 scik
* to\_graphviz() (function is in this file) for decision trees produce

"""

**if** type(dot\_string)**.** name **!=** 'str':

**raise** TypeError('visualize() requires a string representation of a de 'for decision trees produced by scikit-learn and to\_g 'your code.\n')

*# Set path to your GraphViz executable here*

os**.**environ["PATH"] **+=** os**.**pathsep **+** 'C:/Program Files (x86)/Graphviz2.38/b graph **=** graphviz**.**Source(dot\_string)

graph**.**format **=** image\_format graph**.**render(save\_file, view**=True**)

**def** to\_graphviz(tree, dot\_string**=**'', uid**=-**1, depth**=**0): """



Converts a tree to DOT format for use with visualize/GraphViz """

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\_

**if** type(sub\_trees) **is** dict:

**if not** split\_decision:

dot\_string, right\_child, uid **=** to\_graphviz(sub\_trees, dot\_str dot\_string **+=** ' node{0} -> node{1} [label="False"];\n'**.**for

## else:

dot\_string, left\_child, uid **=** to\_graphviz(sub\_trees, dot\_stri dot\_string **+=** ' node{0} -> node{1} [label="True"];\n'**.**form

## else:

uid **+=** 1

dot\_string **+=** ' node{0} [label="y = {1}"];\n'**.**format(uid, sub\_

**if not** split\_decision:

dot\_string **+=** ' node{0} -> node{1} [label="False"];\n'**.**for

## else:

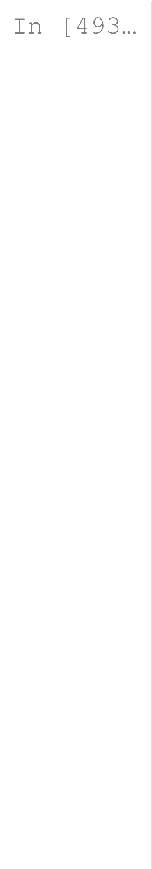
dot\_string **+=** ' node{0} -> node{1} [label="True"];\n'**.**form

**if** depth **==** 0: dot\_string **+=** '}\n' **return** dot\_string

## else:

**return** dot\_string, node\_id, uid

**if** name **==** ' main ':



*# Load the training data*

M **=** np**.**genfromtxt('./monks-1.train', missing\_values**=**0, skip\_header**=**0, del ytrn **=** M[:, 0]

Xtrn **=** M[:, 1:]

*# Load the test data*

M **=** np**.**genfromtxt('./monks-1.test', missing\_values**=**0, skip\_header**=**0, deli ytst **=** M[:, 0]

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

*# Learn a decision tree of depth 3*

decision\_tree **=** id3(Xtrn, ytrn, max\_depth**=**3)

*# 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, './my\_learned\_tree')

*# Compute the test error*

y\_pred **=** [predict\_example(x, decision\_tree) **for** x **in** Xtst] tst\_err **=** compute\_error(ytst, y\_pred)

print('Test Error = {0:4.2f}%.'**.**format(tst\_err **\*** 100))





















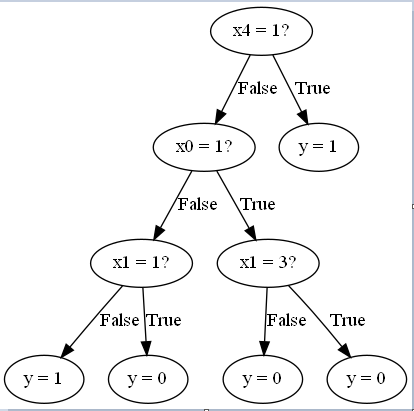
















*# Load the test data*

M **=** np**.**genfromtxt('./monks-1.test', missing\_values**=**0, skip\_header**=**0, delimite ytst **=** M[:, 0]

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

testErr **=** {} trainErr **=** {}

**for** i **in** range(1,11):

decision\_tree **=** id3(Xtrn, ytrn, max\_depth**=**i)

*#compute test error*

y\_pred **=** [predict\_example(x, decision\_tree) **for** x **in** Xtst] testErr[i] **=** compute\_error(ytst, y\_pred)

*#compute training error*

y\_pred **=** [predict\_example(x, decision\_tree) **for** x **in** Xtrn] trainErr[i] **=** compute\_error(ytrn, y\_pred)

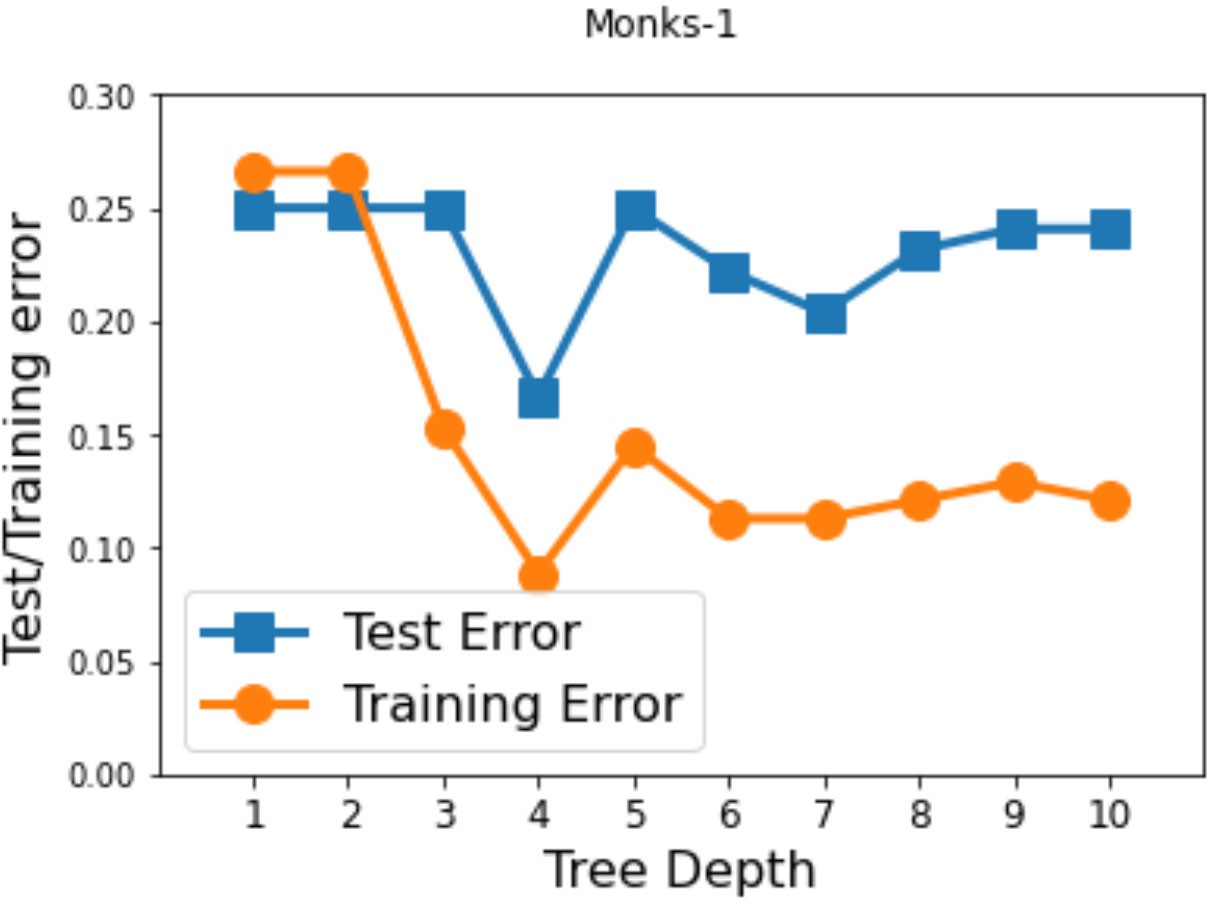
*#Plot tree depth on the x-axis and test/training errors on the y-axis.*

fig **=** plt**.**figure() fig**.**suptitle("Monks-1")

plt**.**plot(testErr**.**keys(), testErr**.**values(), marker**=**'s', linewidth**=**3, markersiz plt**.**plot(trainErr**.**keys(), trainErr**.**values(), marker**=**'o', linewidth**=**3, markers plt**.**xlabel('Tree Depth', fontsize**=**16)

plt**.**ylabel('Test/Training error', fontsize**=**16) plt**.**xticks(list(range(1,11)), fontsize**=**12) plt**.**legend(['Test Error', 'Training Error'], fontsize**=**16) plt**.**axis([0, 11, 0, 0.3])

plt**.**show()





*# Load the test data*

M **=** np**.**genfromtxt('./monks-2.test', missing\_values**=**0, skip\_header**=**0, delimite ytst **=** M[:, 0]

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

testErr **=** {} trainErr **=** {}

**for** i **in** range(1,11):

decision\_tree **=** id3(Xtrn, ytrn, max\_depth**=**i)

*#compute test error*

y\_pred **=** [predict\_example(x, decision\_tree) **for** x **in** Xtst] testErr[i] **=** compute\_error(ytst, y\_pred)

*#compute training error*

y\_pred **=** [predict\_example(x, decision\_tree) **for** x **in** Xtrn] trainErr[i] **=** compute\_error(ytrn, y\_pred)

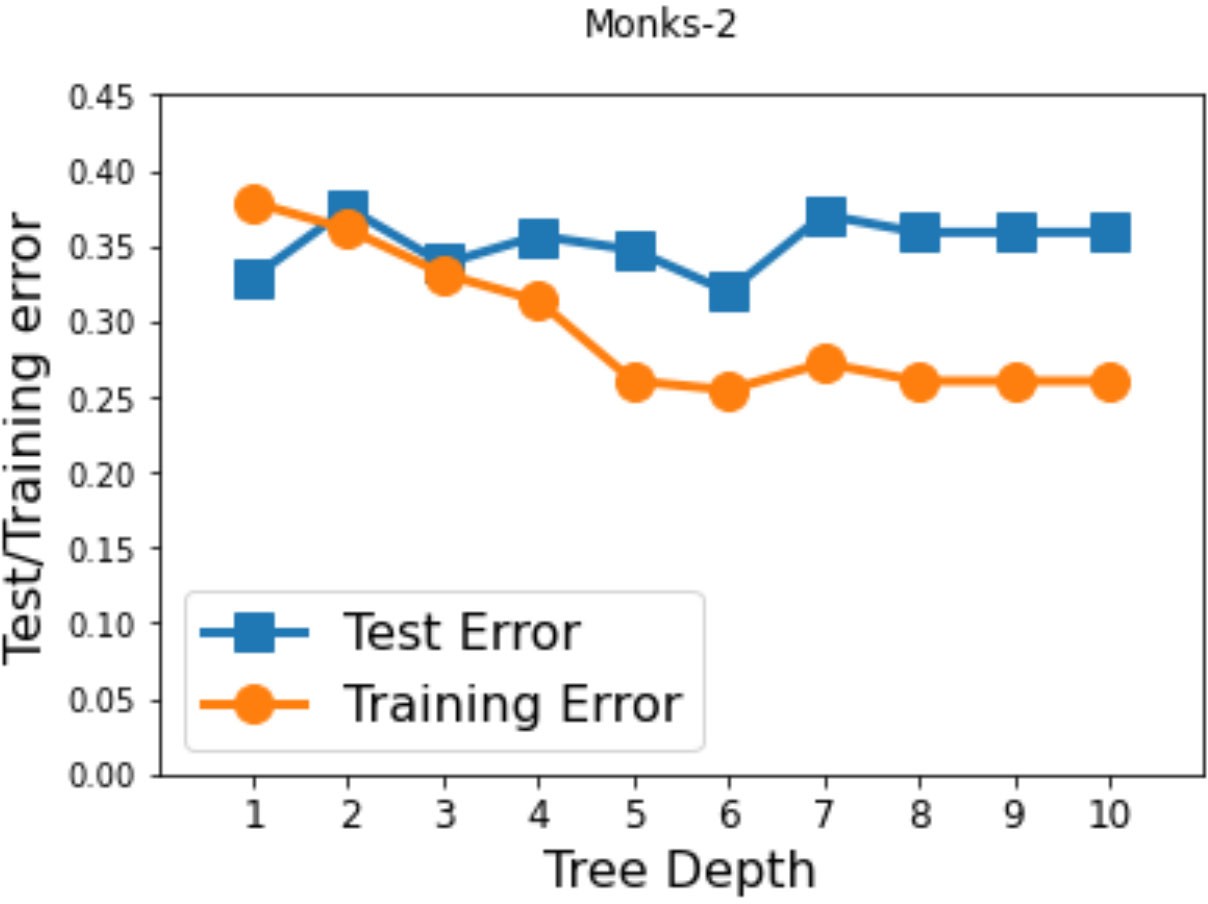
*#Plot tree depth on the x-axis and test/training errors on the y-axis.*

fig **=** plt**.**figure() fig**.**suptitle("Monks-2")

plt**.**plot(testErr**.**keys(), testErr**.**values(), marker**=**'s', linewidth**=**3, markersiz plt**.**plot(trainErr**.**keys(), trainErr**.**values(), marker**=**'o', linewidth**=**3, markers plt**.**xlabel('Tree Depth', fontsize**=**16)

plt**.**ylabel('Test/Training error', fontsize**=**16) plt**.**xticks(list(range(1,11)), fontsize**=**12) plt**.**legend(['Test Error', 'Training Error'], fontsize**=**16) plt**.**axis([0, 11, 0, 0.45])

plt**.**show()



M **=** np**.**genfromtxt('./monks-3.train', missing\_values**=**0, skip\_header**=**0, delimit ytrn **=** M[:, 0]



Xtrn **=** M[:, 1:]

*# Load the test data*

M **=** np**.**genfromtxt('./monks-3.test', missing\_values**=**0, skip\_header**=**0, delimite ytst **=** M[:, 0]

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

testErr **=** {} trainErr **=** {}

**for** i **in** range(1,11):

decision\_tree **=** id3(Xtrn, ytrn, max\_depth**=**i)

*#compute test error*

y\_pred **=** [predict\_example(x, decision\_tree) **for** x **in** Xtst] testErr[i] **=** compute\_error(ytst, y\_pred)

*#compute training error*

y\_pred **=** [predict\_example(x, decision\_tree) **for** x **in** Xtrn] trainErr[i] **=** compute\_error(ytrn, y\_pred)

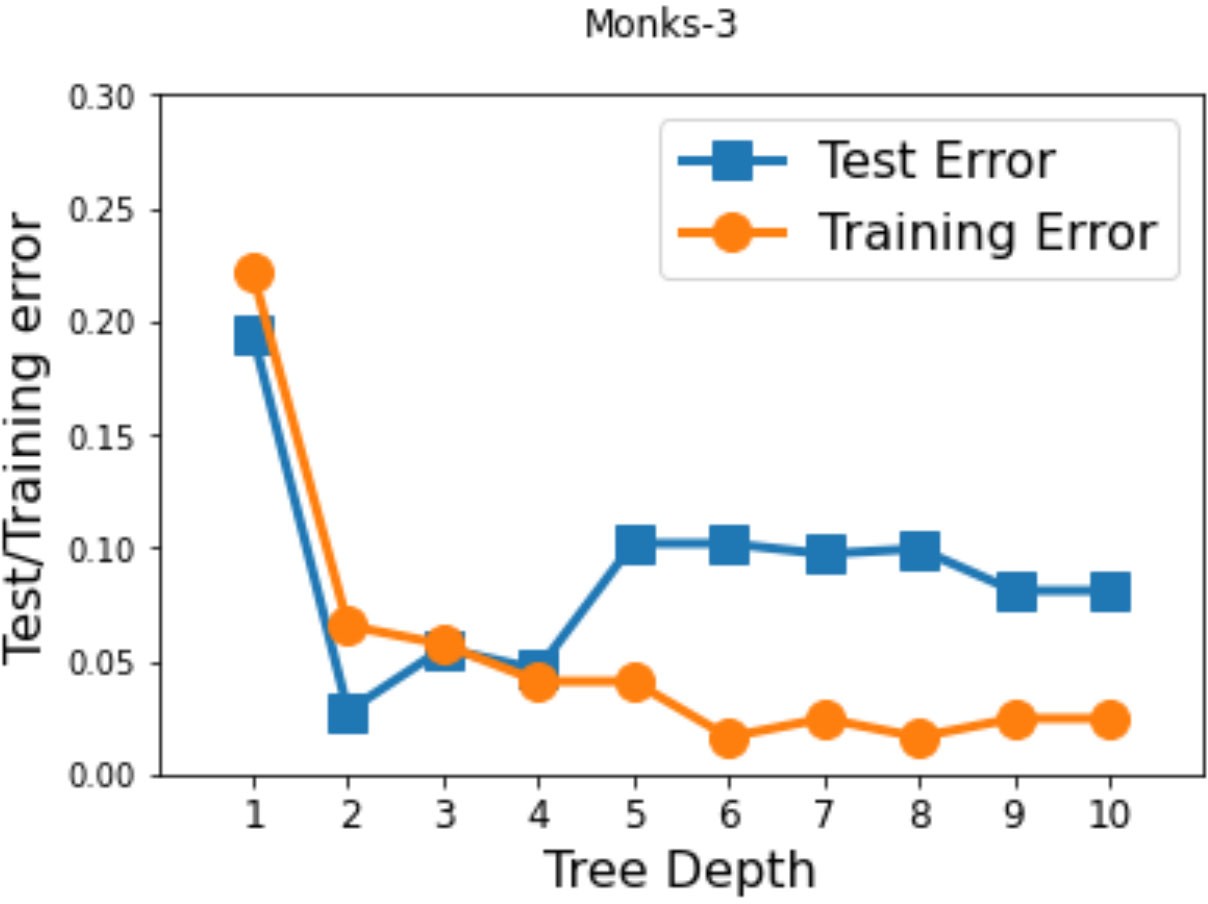
*#Plot tree depth on the x-axis and test/training errors on the y-axis.*

fig **=** plt**.**figure() fig**.**suptitle("Monks-3")

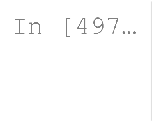
plt**.**plot(testErr**.**keys(), testErr**.**values(), marker**=**'s', linewidth**=**3, markersiz plt**.**plot(trainErr**.**keys(), trainErr**.**values(), marker**=**'o', linewidth**=**3, markers plt**.**xlabel('Tree Depth', fontsize**=**16)

plt**.**ylabel('Test/Training error', fontsize**=**16) plt**.**xticks(list(range(1,11)), fontsize**=**12) plt**.**legend(['Test Error', 'Training Error'], fontsize**=**16) plt**.**axis([0, 11, 0, 0.3])

plt**.**show()





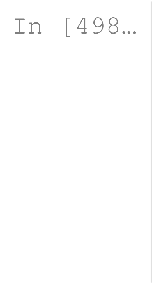


**from** sklearn.metrics **import** confusion\_matrix

**from** sklearn.metrics **import** plot\_confusion\_matrix

**from** sklearn.svm **import** SVC





M **=** np**.**genfromtxt('./monks-1.train', missing\_values**=**0, skip\_header**=**0, delimit ytrn **=** M[:, 0]

Xtrn **=** M[:, 1:]

*# Load the test data*

M **=** np**.**genfromtxt('./monks-1.test', missing\_values**=**0, skip\_header**=**0, delimite ytst **=** M[:, 0]

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



decision\_tree\_1c **=** id3(Xtrn, ytrn, max\_depth**=**1)

*# Pretty print it to console*

pretty\_print(decision\_tree\_1c)

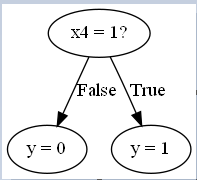


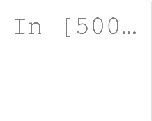












y\_pred\_1c **=** [predict\_example(x, decision\_tree\_1c) **for** x **in** Xtst] print("Confusion Matrix (Depth = 1):") print(confusion\_matrix(ytst, y\_pred\_1c))









*# Learn a decision tree of depth 3*

decision\_tree\_3c **=** id3(Xtrn, ytrn, max\_depth**=**3)

*# Pretty print it to console*

pretty\_print(decision\_tree\_3c)



















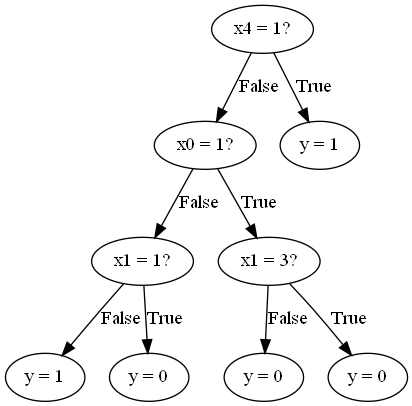


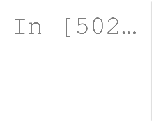












y\_pred\_3c **=** [predict\_example(x, decision\_tree\_3c) **for** x **in** Xtst] print("Confusion Matrix (Depth = 3):") print(confusion\_matrix(ytst, y\_pred\_3c))









*# Learn a decision tree of depth 5*

decision\_tree\_5c **=** id3(Xtrn, ytrn, max\_depth**=**5)

*# Pretty print it to console*

pretty\_print(decision\_tree\_5c)























































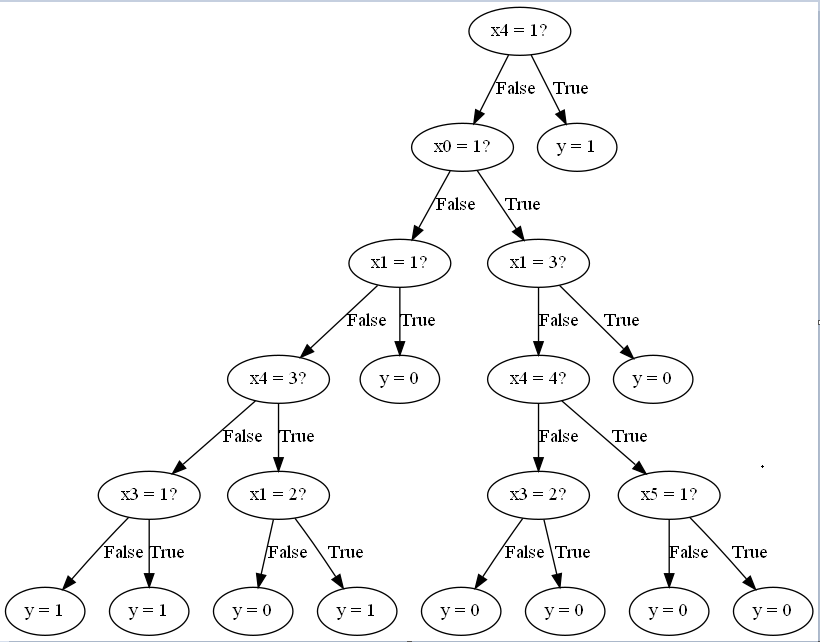


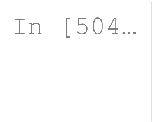












y\_pred\_5c **=** [predict\_example(x, decision\_tree\_5c) **for** x **in** Xtst] print("Confusion Matrix (Depth = 5):") print(confusion\_matrix(ytst, y\_pred\_5c))

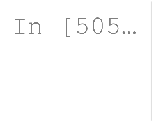












**from** sklearn.tree **import** DecisionTreeClassifier

**from** sklearn **import** tree

**from** sklearn.datasets **import** load\_iris



M **=** np**.**genfromtxt('./monks-1.train', missing\_values**=**0, skip\_header**=**0, delimit ytrn **=** M[:, 0]

Xtrn **=** M[:, 1:]

M **=** np**.**genfromtxt('./monks-1.test', missing\_values**=**0, skip\_header**=**0, delimite ytst **=** M[:, 0]

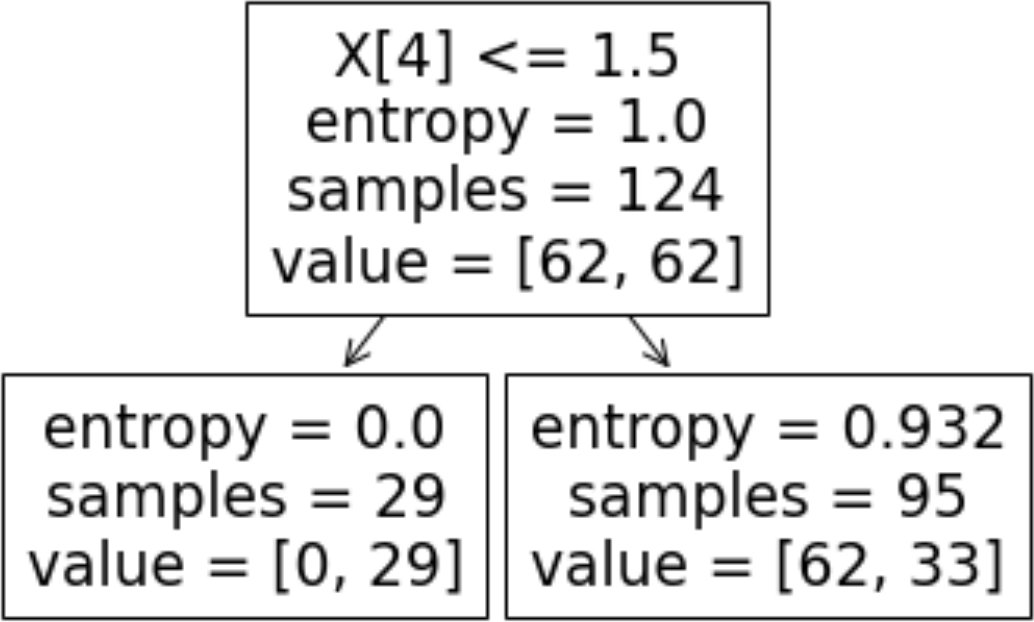
Xtst **=** M[:, 1:]

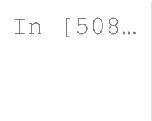


decision\_tree\_1d **=** DecisionTreeClassifier(criterion**=**'entropy', max\_depth **=** 1) tree**.**plot\_tree(decision\_tree\_1d)

# 







y\_pred\_1d **=** [decision\_tree\_1d**.**predict([x]) **for** x **in** Xtst] print("Confusion Matrix (Depth = 1):") print(confusion\_matrix(ytst, y\_pred\_1d))





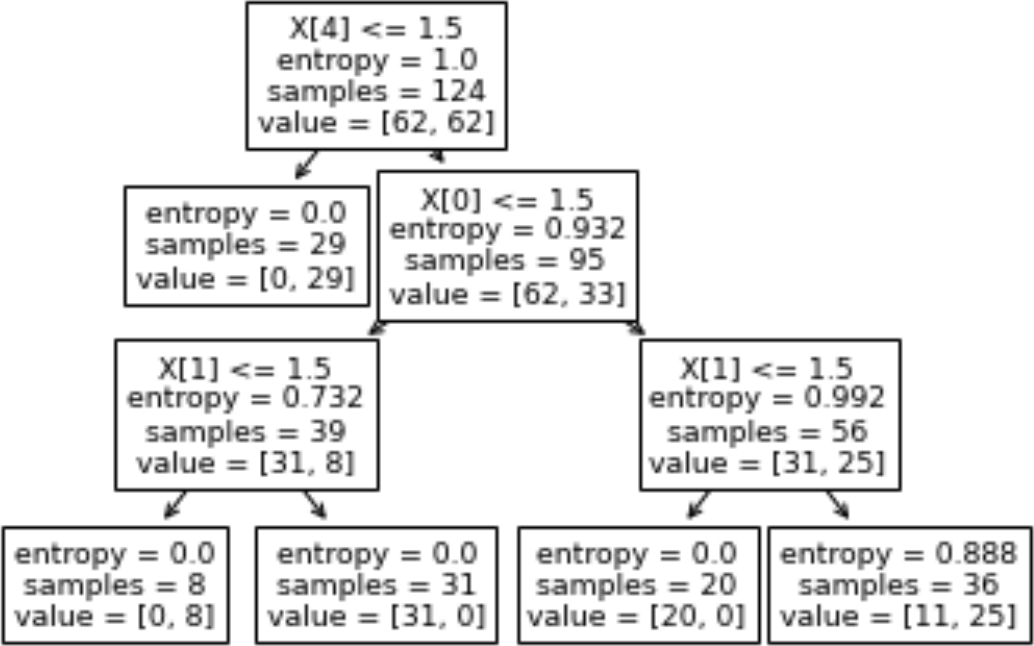


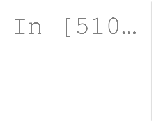


decision\_tree\_3d **=** DecisionTreeClassifier(criterion**=**'entropy', max\_depth **=** 3) tree**.**plot\_tree(decision\_tree\_3d)







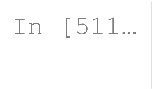


y\_pred\_3d **=** [decision\_tree\_3d**.**predict([x]) **for** x **in** Xtst] print("Confusion Matrix (Depth = 3):") print(confusion\_matrix(ytst, y\_pred\_3d))





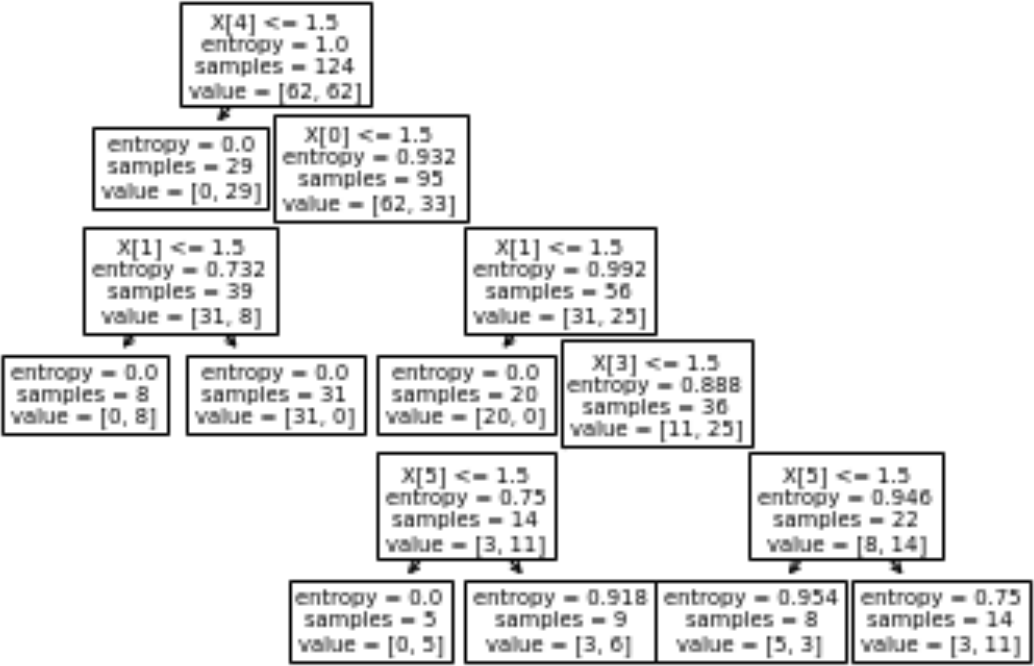


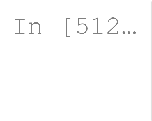


decision\_tree\_5d **=** DecisionTreeClassifier(criterion**=**'entropy', max\_depth **=** 5) tree**.**plot\_tree(decision\_tree\_5d)









y\_pred\_5d **=** [decision\_tree\_5d**.**predict([x]) **for** x **in** Xtst] print("Confusion Matrix (Depth = 5):") print(confusion\_matrix(ytst, y\_pred\_5d))









M **=** np**.**genfromtxt('./part\_E\_Dataset.txt', missing\_values**=**0, skip\_header**=**0)



*#We want to find the mean of each attribute, not of all the attributes mushed*

trpose **=** np**.**transpose(M) Mat **=** []

**for** row **in** trpose: mean **=** np**.**mean(row) temp **=** []

**for** point **in** row:

**if** (point **<=** mean): temp**.**append(0)

**else**:

temp**.**append(1) Mat**.**append(temp)

Mat **=** np**.**asarray(np**.**transpose(Mat)) np**.**random**.**shuffle(Mat)



*#5000 rows in M*

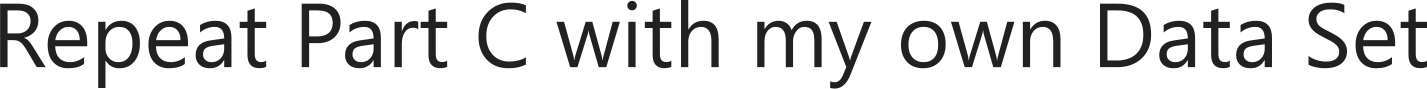
ytrn **=** Mat[:3751, 0] *#75% of dataset*

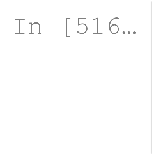
Xtrn **=** Mat[:3751, 1:]

ytst **=** Mat[3751:, 0] Xtst **=** Mat[3751:, 1:]

np**.**unique(ytst)





decision\_tree\_1ec **=** id3(Xtrn, ytrn, max\_depth**=**1)

*# Pretty print it to console*

pretty\_print(decision\_tree\_1ec)

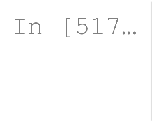












y\_pred\_1ec **=** [predict\_example(x, decision\_tree\_1ec) **for** x **in** Xtst] print("Confusion Matrix (Depth = 1):") print(confusion\_matrix(ytst, y\_pred\_1ec))









decision\_tree\_3ec **=** id3(Xtrn, ytrn, max\_depth**=**3)

*# Pretty print it to console*

pretty\_print(decision\_tree\_3ec)













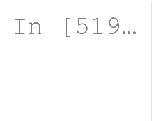












y\_pred\_3ec **=** [predict\_example(x, decision\_tree\_3ec) **for** x **in** Xtst] print("Confusion Matrix (Depth = 3):") print(confusion\_matrix(ytst, y\_pred\_3ec))









decision\_tree\_5ec **=** id3(Xtrn, ytrn, max\_depth**=**5)

*# Pretty print it to console*

pretty\_print(decision\_tree\_5ec)































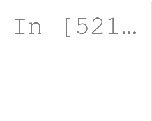












y\_pred\_5ec **=** [predict\_example(x, decision\_tree\_5ec) **for** x **in** Xtst] print("Confusion Matrix (Depth = 5):") print(confusion\_matrix(ytst, y\_pred\_5ec))





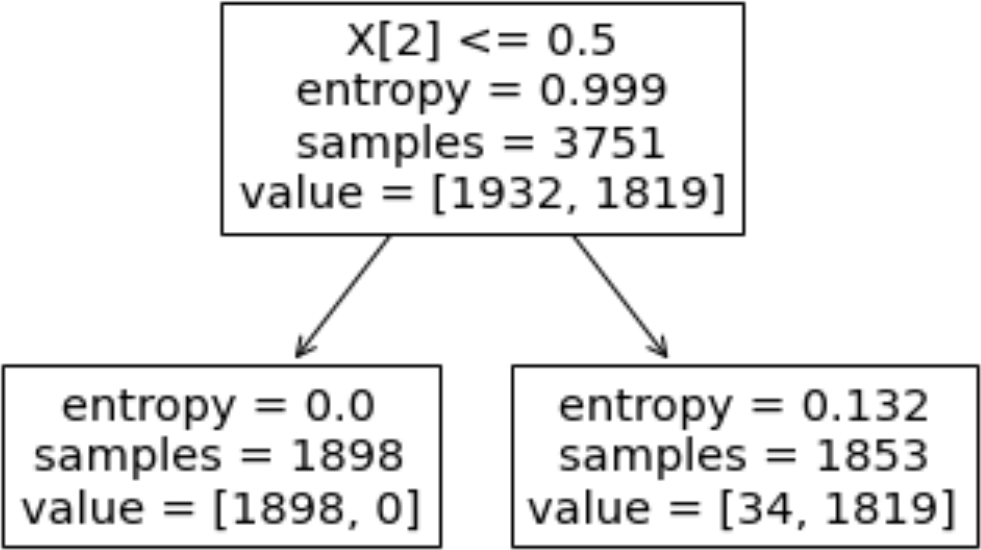


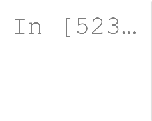


decision\_tree\_1ed **=** DecisionTreeClassifier(criterion**=**'entropy', max\_depth **=** 1 tree**.**plot\_tree(decision\_tree\_1ed)

# 







y\_pred\_1ed **=** [decision\_tree\_1ed**.**predict([x]) **for** x **in** Xtst] print("Confusion Matrix (Depth = 1):") print(confusion\_matrix(ytst, y\_pred\_1ed))





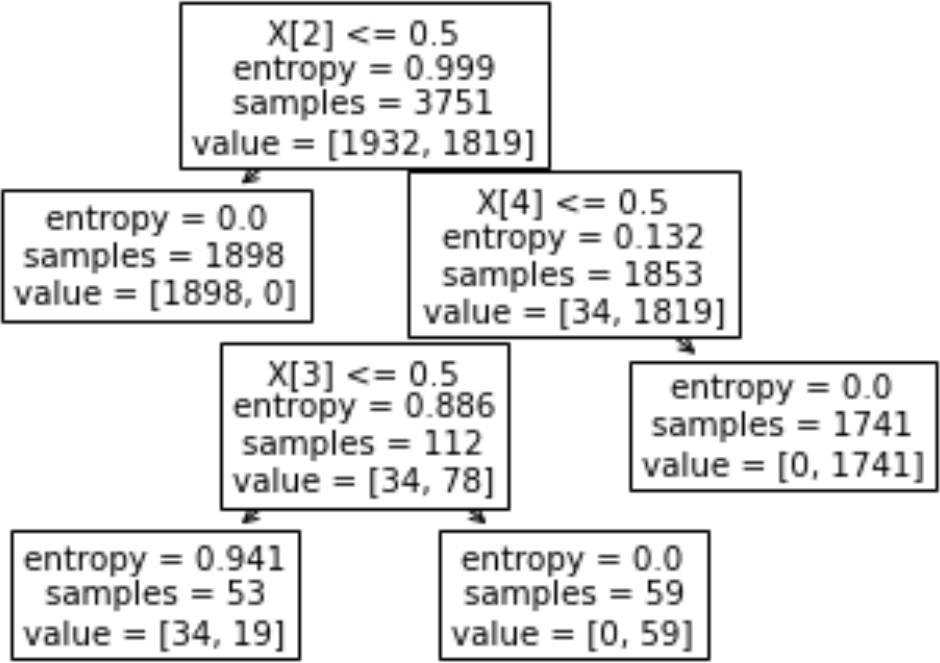


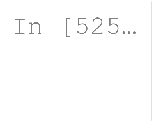


decision\_tree\_3ed **=** DecisionTreeClassifier(criterion**=**'entropy', max\_depth **=** 3 tree**.**plot\_tree(decision\_tree\_3ed)







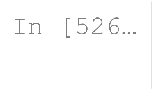


y\_pred\_3ed **=** [decision\_tree\_1ed**.**predict([x]) **for** x **in** Xtst] print("Confusion Matrix (Depth = 3):") print(confusion\_matrix(ytst, y\_pred\_3ed))





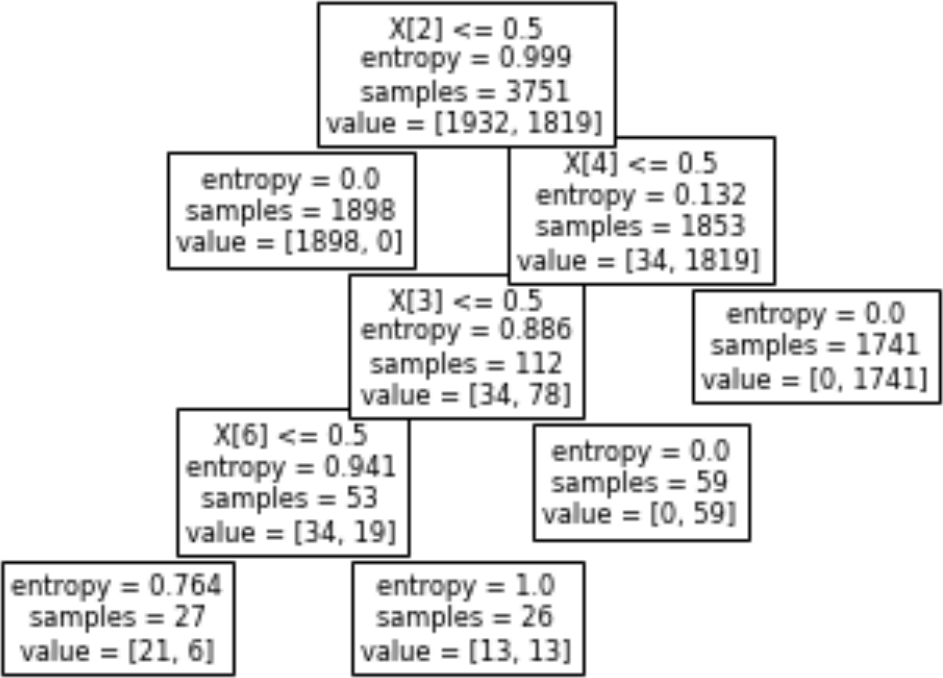


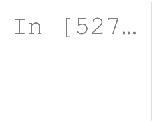


decision\_tree\_5ed **=** DecisionTreeClassifier(criterion**=**'entropy', max\_depth **=** 5 tree**.**plot\_tree(decision\_tree\_5ed)









y\_pred\_5ed **=** [decision\_tree\_1ed**.**predict([x]) **for** x **in** Xtst] print("Confusion Matrix (Depth = 5):") print(confusion\_matrix(ytst, y\_pred\_5ed))







