# CS6301 MACHINE LEARNING LAB WEEK – 7 DECISION TREES

## SRIHARI. S – 2018103601

**Date**: 05-04-2021 Monday

**Aim**: To implement ID3, C4.5 AND CART algorithms and classify handwritten digits of MNIST dataset using Decision Tree algorithms.

**IMPLEMENTATION OF ID3 ALGORITHM**

**DATASET USED:** Party Dataset. We aim to identify the activity done by a person based on his/her dealines, laziness and wish to party. The activity can be either study or pub.

**INPUT:**

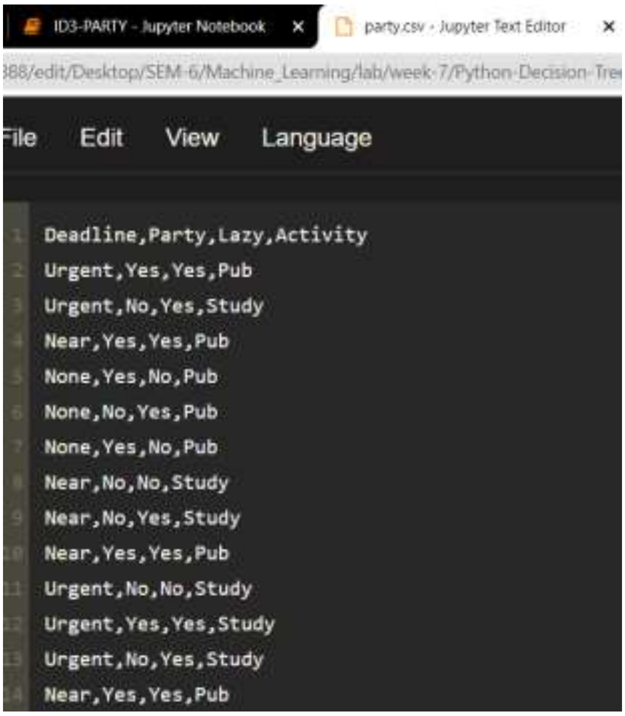
**Features:**

Deadline – Takes values Urgent, Near, None

Party – Takes values Yes,No

Lazy  – Takes values Yes,No

Output Feature : Activity – Pub, Study




```
Deadline,Party,Lazy,Activity
Urgent,Yes,Yes,Pub
Urgent,No,Yes,Study
Near,Yes,Yes,Pub
None,Yes,No,Pub
None,No,Yes,Pub
None,Yes,No,Pub
Near,No,No,Study
Near,No,Yes,Study
Near,Yes,Yes,Pub
Urgent,No,No,Study
Urgent,Yes,Yes,Study
Urgent,No,Yes,Study
Near,Yes,Yes,Pub
```

**OUTPUT:**

TOTAL ENTROPY = 0.98

For the root:

INFO GAIN (DEADLINE) = 0.357

INFO GAIN (PARTY) = 0.449

INFO GAIN (LAZY)= 0.008

Similarly ID3 algorithm is used for subsequent depths.

```
Classes: Counter({'Pub': 17, 'Study': 13})

 Number of Instances of the Current Sub Class is 30.0:

 Probabilities of Class Pub is 0.43333333333333335:

 Probabilities of Class Study is 0.5666666666666667:

 Total Entropy of Party Data Set: 0.9871377743721863
```

```
Information Gain Calculation of  Deadline

Classes: Counter({'Pub': 6, 'Study': 6})

 Number of Instances of the Current Sub Class is 12.0:

 Probabilities of Class Pub is 0.5:

 Probabilities of Class Study is 0.5:

Classes: Counter({'Pub': 9})

 Number of Instances of the Current Sub Class is 9.0:

 Probabilities of Class Pub is 1.0:

 Probabilities of Class Pub is 1.0:

Classes: Counter({'Study': 7, 'Pub': 2})

 Number of Instances of the Current Sub Class is 9.0:

 Probabilities of Class Pub is 0.2222222222222222:

 Probabilities of Class Study is 0.7777777777777778:

Classes: Counter({'Pub': 17, 'Study': 13})

 Number of Instances of the Current Sub Class is 30.0:

 Probabilities of Class Pub is 0.43333333333333335:

 Probabilities of Class Study is 0.5666666666666667:
Info-gain for Deadline is :0.35787642241960027
```

```
Information Gain Calculation of  Party

Classes: Counter({'Study': 12, 'Pub': 3})

 Number of Instances of the Current Sub Class is 15.0:

 Probabilities of Class Pub is 0.2:

 Probabilities of Class Study is 0.8:

Classes: Counter({'Pub': 14, 'Study': 1})

 Number of Instances of the Current Sub Class is 15.0:

 Probabilities of Class Pub is 0.06666666666666667:

 Probabilities of Class Study is 0.9333333333333333:

Classes: Counter({'Pub': 17, 'Study': 13})

 Number of Instances of the Current Sub Class is 30.0:

 Probabilities of Class Pub is 0.43333333333333335:

 Probabilities of Class Study is 0.5666666666666667:

 Info-gain for Party is: 0.4494940594177945
```

```
Information Gain Calculation of  Lazy

Classes: Counter({'Pub': 6, 'Study': 6})

 Number of Instances of the Current Sub Class is 12.0:

 Probabilities of Class Pub is 0.5:

 Probabilities of Class Study is 0.5:

Classes: Counter({'Pub': 11, 'Study': 7})

 Number of Instances of the Current Sub Class is 18.0:

 Probabilities of Class Pub is 0.3888888888888889:

 Probabilities of Class Study is 0.6111111111111112:

Classes: Counter({'Pub': 17, 'Study': 13})

 Number of Instances of the Current Sub Class is 30.0:

 Probabilities of Class Pub is 0.43333333333333335:

 Probabilities of Class Study is 0.5666666666666667:

 Info-gain for Lazy is:0.008690515487248751
```

**Train**

```
              precision    recall  f1-score   support

         Pub       1.00      0.88      0.94        17
       Study       0.87      1.00      0.93        13

    accuracy                           0.93        30
   macro avg       0.93      0.94      0.93        30
weighted avg       0.94      0.93      0.93        30
```

```
Name: predicted, dtype: object
[[15  2]
 [ 0 13]]
Accuracy =  93.33333333333333 %
```

**Test**

```
              precision    recall  f1-score   support

         Pub       1.00      0.83      0.91         6
       Study       0.80      1.00      0.89         4

    accuracy                           0.90        10
   macro avg       0.90      0.92      0.90        10
weighted avg       0.92      0.90      0.90        10
```

```
[[7 1]
 [0 7]]
Accuracy =  93.33333333333333 %
```

**CODE:**

```
import pandas as pd
from sklearn import metrics
df_tennis = pd.read_csv('party.csv')
print("\n Given Party Data Set:\n\n", df_tennis)

def entropy(probs):
    import math
    return sum( [-prob*math.log(prob, 2) for prob in probs] )
```

```
def entropy_of_list(a_list):
    from collections import Counter
    cnt = Counter(x for x in a_list)
    print("\nClasses:",cnt)
    num_instances = len(a_list)*1.0
    print("\n Number of Instances of the Current Sub Class is {0}:".format(num_instances ))
    probs = [x / num_instances for x in cnt.values()]  # x means no of YES/NO
    print(" \n Probabilities of Class {0} is {1}:".format(min(cnt),min(probs)))
    print(" \n Probabilities of Class {0} is {1}:".format(max(cnt),max(probs)))
    return entropy(probs)
print("\n  INPUT DATA SET FOR ENTROPY CALCULATION:\n", df_tennis['Activity'])
total_entropy = entropy_of_list(df_tennis['Activity'])
print("\n Total Entropy of Party Data Set:",total_entropy)

def information_gain(df, split_attribute_name, target_attribute_name, trace=0):
    print("Information Gain Calculation of ",split_attribute_name)
    df_split = df.groupby(split_attribute_name)
    nobs = len(df.index) * 1.0
    df_agg_ent = df_split.agg({target_attribute_name : [entropy_of_list, lambda x: len(x)/nobs]
})[target_attribute_name]
    df_agg_ent.columns = ['Entropy', 'PropObservations']
    new_entropy = sum( df_agg_ent['Entropy'] * df_agg_ent['PropObservations'] )
    old_entropy = entropy_of_list(df[target_attribute_name])
    return old_entropy - new_entropy


print('Info-gain for Deadline is :'+str( information_gain(df_tennis, 'Deadline', 'Activity')),"\n")
print('\n Info-gain for Party is: ' + str( information_gain(df_tennis, 'Party', 'Activity')),"\n")
print('\n Info-gain for Lazy is:' + str( information_gain(df_tennis, 'Lazy', 'Activity')),"\n")

def id3(df, target_attribute_name, attribute_names, default_class=None):
    from collections import Counter
    cnt = Counter(x for x in df[target_attribute_name])
    if len(cnt) == 1:
        return next(iter(cnt))  # next input data set, or raises StopIteration when EOF is hit.
    elif df.empty or (not attribute_names):
        return default_class
    else:
        default_class = max(cnt.keys()) #No of YES and NO Class
        gainz = [information_gain(df, attr, target_attribute_name) for attr in attribute_names] #
        index_of_max = gainz.index(max(gainz))
        best_attr = attribute_names[index_of_max]
        tree = {best_attr:{}} # Iniiate the tree with best attribute as a node
        remaining_attribute_names = [i for i in attribute_names if i != best_attr]
        for attr_val, data_subset in df.groupby(best_attr):
            subtree = id3(data_subset,
                        target_attribute_name,
                        remaining_attribute_names,
                        default_class)
            tree[best_attr][attr_val] = subtree
```

```
        return tree
attribute_names = list(df_tennis.columns)
print("List of Attributes:", attribute_names)
attribute_names.remove('Activity') #Remove the class attribute
print("Predicting Attributes:", attribute_names)

from pprint import pprint
tree = id3(df_tennis,'Activity',attribute_names)
print("\n\nThe Resultant Decision Tree is :\n")
pprint(tree)
attribute = next(iter(tree))
print("Best Attribute :\n",attribute)
print("Tree Keys:\n",tree[attribute].keys())

def classify(instance, tree, default=None):
    attribute = next(iter(tree)) # Outlook/Humidity/Wind
    print("Key:",tree.keys())  # [Outlook,Humidity,Wind ]
    print("Attribute:",attribute) # [Key /Attribute Both are same ]
    if instance[attribute] in tree[attribute].keys(): # Value of the attributs in  set of Tree keys
        result = tree[attribute][instance[attribute]]
        print("Instance Attribute:",instance[attribute],"TreeKeys :",tree[attribute].keys())
        if isinstance(result, dict): # this is a tree, delve deeper
            return classify(instance, result)
        else:
            return result # this is a label
    else:
        return default
df_tennis['predicted'] = df_tennis.apply(classify, axis=1, args=(tree,'No') )
print(df_tennis['predicted'])
confusion = metrics.confusion_matrix(y_true = test_data['Activity'], y_pred =
test_data['predicted2'])
print(confusion)
print("Accuracy = ",metrics.accuracy_score(y_true = test_data['Activity'], y_pred =
test_data['predicted2'])*100,"%")
df_tennis[['Activity', 'predicted']]
confusion = metrics.confusion_matrix(y_true = df_tennis['Activity'], y_pred = df_tennis['predicted'])
print(confusion)
print("Accuracy = ",metrics.accuracy_score(y_true = df_tennis['Activity'], y_pred = df_tennis['predict
ed'])*100,"%")
class_wise = metrics.classification_report(y_true = df_tennis['Activity'], y_pred =
df_tennis['predicted'])
print(class_wise)
training_data = df_tennis.iloc[1:-10] # all but last four instances
test_data  = df_tennis.iloc[-10:] # just the last four
train_tree = id3(training_data, 'Activity', attribute_names)
test_data['predicted2'] = test_data.apply( classify, axis=1,
args=(train_tree,'Yes') )
print ('\n\n Accuracy is : ' + str( sum(test_data['Activity']==test_data['predicted2'] ) /
(1.0*len(test_data.index)) ))
class_wise = metrics.classification_report(y_true = test_data['Activity'], y_pred =
test_data['predicted2'])
```

```
print(class_wise)
```

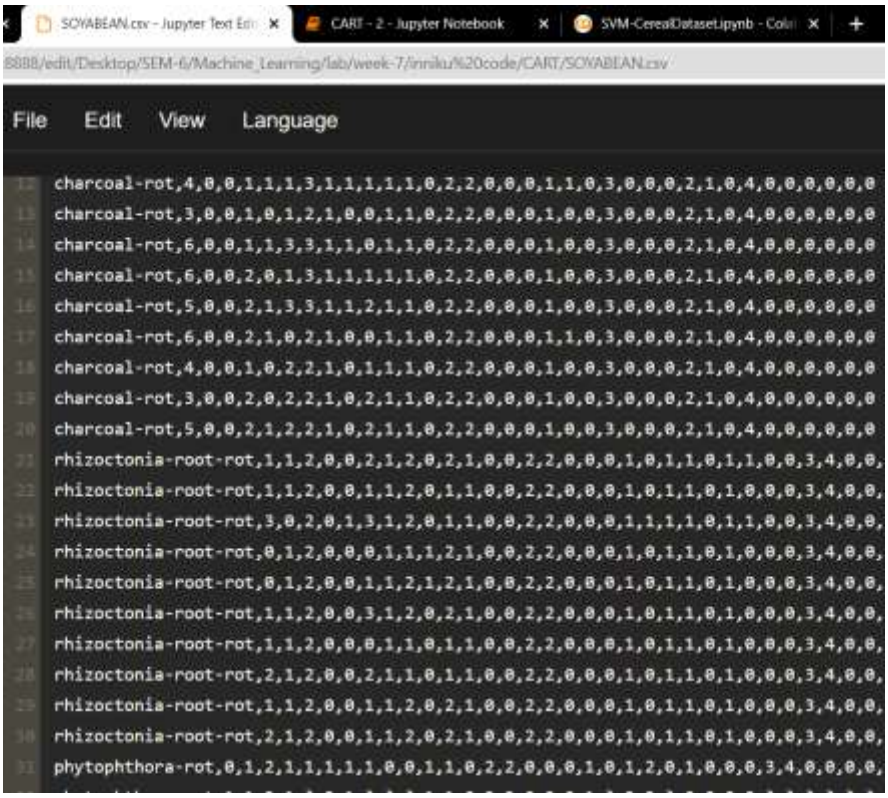
## CART ALGORITHM

## DATASET – Soyabean dataset

## Url:

https://archive.ics.uci.edu/ml/datasets/Soybean+(Large)#:~:text=Data%20Set%20Information%3A&text=There%20are%2035%20categorical%20attributes,values%20is%20encoded%20as%20%22%3F"

We aim to identify the 35 different classes of Soyabean from its features like germination, plant growth, temperature, leaves, mycelium etc.




**OUTPUT:**

```
Predictive accuracy for k =  12  is  0.9523809523809523
[[1 0 0 0 1 0 0 0 0 0 0 0]
 [0 2 0 0 0 0 0 0 0 2 0 0]
 [0 0 1 1 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 3 0 0 0 0 0 0 0]
 [0 0 0 0 0 2 0 0 0 0 0 0]
 [0 0 0 0 0 0 2 0 0 0 0 0]
 [0 0 0 0 0 0 0 4 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 2]]
Precision =  0.8541666666666666
Recall =  0.7916666666666666
F1-score =  0.7936507936507936
```

**Code:**

```
import csv
from collections import defaultdict
import pandas as pd
from sklearn.metrics import precision_recall_fscore_support
from sklearn.utils import resample
from sklearn.utils import shuffle
class DecisionTree:
   def __init__(self, col=-1, value=None, trueBranch=None, falseBranch=None, results=None,
summary=None):
       self.col = col
       self.value = value
       self.trueBranch = trueBranch
       self.falseBranch = falseBranch
       self.results = results
       self.summary = summary

def kfold(data,k):
   X= shuffle(data,random_state=42)
   X=X.to_numpy()
   n = len(data)/k
   if(n>int(n)):
      n= (int(n)+1)
   trainingData = X[0:n*(k-1)]
   test = X[n*(k-1):len(data)]
   testData = test[:,:test.shape[1]-1]
   y_test = test[:,test.shape[1]-1:]
   decisionTree = growTree(trainingData, evaluationFunction=gini)
   prune(decisionTree, 0.8, notify=True)
   count=0
   count1=0
   true=[]
   pred=[]
   for i in range(testData.shape[0]):
```

```
        count1 +=1
        t = classify(testData[i], decisionTree)
        for key, value in t.items():
            pred.append(key)
            true.append(y_test[i])
            if(key==y_test[i]):
                count +=1
    print("\nPredictive accuracy for k = ",k," is ",count/count1)
    print(confusion_matrix(true,pred))
    a,b,c,d = precision_recall_fscore_support(true, pred, average="macro")
    print("Precision = ",a, "\nRecall = ",b," \nF1-score = ",c)

def bootstrap(data,n):
    data = data.to_numpy()
    for j in range(n):
        trainingData = resample(data,n_samples=250)
        testData = resample(data,n_samples=50)
        y_test = testData[:,testData.shape[1]-1:]
        testData = testData[:,:testData.shape[1]-1]
        decisionTree = growTree(trainingData, evaluationFunction=gini)
        prune(decisionTree, 0.8, notify=True)
        count=0
        count1=0
        true=[]
        pred=[]
        for i in range(testData.shape[0]):
            count1 +=1
            t = classify(testData[i], decisionTree)
            for key, value in t.items():
                pred.append(key)
                true.append(y_test[i])
                if(key==y_test[i]):
                    count +=1
        print("\nPredictive accuracy for Bootstrap = ",j+1," is ",count/count1)
        print(confusion_matrix(true,pred))
        a,b,c,d = precision_recall_fscore_support(true, pred, average='macro')
        print("Precision = ",a, "\nRecall = ",b,"\nF1-score = ",c)

def Unique_Counts(rows):
    results_ = {}
    for row in rows:
        r = row[-1]
        if r not in results_: results_[r] = 0
        results_[r] += 1
    return results_
def entropy(rows):
    from math import log
    log2 = lambda x: log(x)/log(2)
    results_ = Unique_Counts(rows)
    entropy_value = 0.0
    for r in results_:
```

```
        prob = float(results_[r])/len(rows)
        entropy_value -= prob*log2(prob)
    return entropy_value

def divideSet(trows, column_, val):
    splitFn = None
    if isinstance(val, int) or isinstance(val, float):
        splitFn = lambda row : row[column_] >= val
    else:
        splitFn = lambda row : row[column_] == val
    lista = [row for row in trows if splitFn(row)]
    listb = [row for row in trows if not splitFn(row)]
    return (lista, listb)

def gini(trows):
    total = len(trows)
    count = Unique_Counts(trows)
    imp_val = 0.0
    for ka in count:
        pa = float(count[ka])/total
        for kb in count:
            if ka == kb: continue
            pb = float(count[kb])/total
            imp_val += (pa*pb)
    return imp_val

def growTree(rows, evaluationFunction=entropy):
    if len(rows) == 0: return DecisionTree()
    currScore = evaluationFunction(rows)
    gain_best = 0.0
    bestAttribute = None
    bestSets = None
    columnCount = len(rows[0]) - 1
    for col_ in range(0, columnCount):
        columnValues = [row_[col_] for row_ in rows]
        lsUnique = list(set(columnValues))
        for value in lsUnique:
            (seta, setb) = divideSet(rows, col_, value)
            prob = float(len(seta)) / len(rows)
            gain = currScore - prob*evaluationFunction(seta) - (1-prob)*evaluationFunction(setb)
            if gain>gain_best and len(seta)>0 and len(setb)>0:
                gain_best = gain
                bestAttribute = (col_, value)
                bestSets = (seta, setb)

    dcY = {'impurity' : '%.3f' % currScore, 'samples' : '%d' % len(rows)}
    if gain_best > 0:
        trueBranch = growTree(bestSets[0], evaluationFunction)
        falseBranch = growTree(bestSets[1], evaluationFunction)
        return DecisionTree(col=bestAttribute[0], value=bestAttribute[1], trueBranch=trueBranch,
                    falseBranch=falseBranch, summary=dcY)
```

```
    else:
        return DecisionTree(results=Unique_Counts(rows), summary=dcY)

def prune(tree, minGain, evaluationFunction=entropy, notify=False):
    if tree.trueBranch.results == None: prune(tree.trueBranch, minGain, evaluationFunction, notify)
    if tree.falseBranch.results == None: prune(tree.falseBranch, minGain, evaluationFunction, notify)
    if tree.trueBranch.results != None and tree.falseBranch.results != None:
        ta, fa = [], []
        for v_, c_ in tree.trueBranch.results.items(): ta += [[v_]] * c_
        for v_, c_ in tree.falseBranch.results.items(): fa += [[v_]] * c_
        prob = float(len(ta)) / len(ta + fa)
        delta_val = evaluationFunction(ta+fa) - prob*evaluationFunction(ta) - (1-
prob)*evaluationFunction(fa)
        if delta_val < minGain:
            tree.trueBranch, tree.falseBranch = None, None
            tree.results = Unique_Counts(ta + fa)

def classify(obs, tree):
    def classify_(obs, tree):
        if tree.results != None:
            return tree.results
        else:
            val = obs[tree.col]
            branch_ = None
            if isinstance(val, int) or isinstance(val, float):
                if val >= tree.value: branch_ = tree.trueBranch
                else: branch_ = tree.falseBranch
            else:
                if val == tree.value: branch_ = tree.trueBranch
                else: branch_ = tree.falseBranch
        return classify_(obs, branch_)
    return classify_(obs, tree)

if __name__ == '__main__':
        from sklearn.metrics import confusion_matrix
        data=pd.read_csv("SOYABEAN.csv",header=None,index_col=None)
        target = data.iloc[:,0]
        data = data.drop(data.columns[0],axis = 1)
        data = data.assign(target1=target)
        data.columns = range(data.shape[1])
        for i in range(2,13):
            kfold(data,i)
        bootstrap(data,1)
```

## C4.5 ALGORITHM

**Dataset : Iris**

**Url**: https://archive.ics.uci.edu/ml/datasets/iris

**Description**: The **Iris Dataset** contains four features (length and width of sepals and petals) of 50 samples of three species of **Iris** (**Iris** setosa, **Iris** virginica and **Iris** versicolor).

**Input**: The following 4 attributes

- sepal length in cm,
- sepal width in cm,
- petal length in cm,
- petal width in cm,

jupyter iris_proc.data 21 hours ago

File Edit View Language

```
5.1,3.5,1.4,0.2,0
4.9,3.0,1.4,0.2,0
4.7,3.2,1.3,0.2,0
4.6,3.1,1.5,0.2,0
5.0,3.6,1.4,0.2,0
5.4,3.9,1.7,0.4,0
4.6,3.4,1.4,0.3,0
5.0,3.4,1.5,0.2,0
4.4,2.9,1.4,0.2,0
4.9,3.1,1.5,0.1,0
5.4,3.7,1.5,0.2,0
```

**Output**: decision: Multiclass classification among 3 classes of flowers: Iris Setosa, Iris Versicolour, Iris Virginica.

**Output:**
**Train:**

```
predictions = clf.predict(X_train)
from sklearn import metrics
confusion = metrics.confusion_matrix(y_true = y_train, y_pred = predictions)
confusion
```

```
array([[26,  0,  0],
       [ 0, 30,  0],
       [ 0,  0, 26]])
```

```
print("Accuracy = ",metrics.accuracy_score(y_true=y_train, y_pred=predictions)*100)
```

```
Accuracy =  100.0
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        26
           1       1.00      1.00      1.00        30
           2       1.00      1.00      1.00        26

    accuracy                           1.00        82
   macro avg       1.00      1.00      1.00        82
weighted avg       1.00      1.00      1.00        82
```

**Test:**

```
from sklearn import metrics
confusion = metrics.confusion_matrix(y_true = y_test, y_pred = predictions)
confusion
```

```
array([[24,  0,  0],
       [ 1, 16,  3],
       [ 0,  1, 23]])
```

```
print("Accuracy = ",metrics.accuracy_score(y_true=y_test, y_pred=predictions)*100)
```

```
Accuracy =  92.64705882352942
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        24
           1       0.94      0.80      0.86        20
           2       0.88      0.96      0.92        24

    accuracy                           0.93        68
   macro avg       0.93      0.92      0.92        68
weighted avg       0.93      0.93      0.92        68
```

**Code:**

```
import math
from xml.etree import ElementTree as ET

def prettify(elem, level=0):
   i = "\n" + level*"  "
   if len(elem):
      if not elem.text or not elem.text.strip():
```

```
            elem.text = i + "  "
        for e in elem:
            prettify(e, level+1)
        if not e.tail or not e.tail.strip():
            e.tail = i
    if level and (not elem.tail or not elem.tail.strip()):
        elem.tail = i
    return elem

def isnum(attr):
    for x in set(attr):
        if not x=="?":
            try:
                x=float(x)
                return isinstance(x,float)
            except ValueError:
                return False
    return True

def entropy(x):
    ent=0
    for k in set(x):
        p_i=float(x.count(k))/len(x)
        ent=ent-p_i* math.log(p_i,2)
    return ent

def gain_ratio(category,attr):
    s=0
    cat=[]
    att=[]
    for i in range(len(attr)):
        if not attr[i]=="?":
            cat.append(category[i])
            att.append(attr[i])
    for i in set(att):
        p_i=float(att.count(i))/len(att)
        cat_i=[]
        for j in range(len(cat)):
            if att[j]==i:
                cat_i.append(cat[j])
        s=s+p_i*entropy(cat_i)
    gain=entropy(cat)-s
    ent_att=entropy(att)
    if ent_att==0:
        return 0
    else:
        return gain/ent_att

def gain(category,attr):
    cats=[]
    for i in range(len(attr)):
```

```
        if not attr[i]=="?":
            cats.append([float(attr[i]),category[i]])
    cats=sorted(cats, key=lambda x:x[0])

    cat=[cats[i][1] for i in range(len(cats))]
    att=[cats[i][0] for i in range(len(cats))]
    if len(set(att))==1:
        return 0
    else:
        gains=[]
        div_point=[]
        for i in range(1,len(cat)):
            if not att[i]==att[i-1]:
                gains.append(entropy(cat[:i])*float(i)/len(cat)+entropy(cat[i:])*(1-float(i)/len(cat)))
                div_point.append(i)
        gain=entropy(cat)-min(gains)

        p_1=float(div_point[gains.index(min(gains))])/len(cat)
        ent_attr= -p_1*math.log(p_1,2)-(1-p_1)*math.log((1-p_1),2)
        return gain/ent_attr

def division_point(category,attr):
    cats=[]
    for i in range(len(attr)):
        if not attr[i]=="?":
            cats.append([float(attr[i]),category[i]])
    cats=sorted(cats, key=lambda x:x[0])

    cat=[cats[i][1] for i in range(len(cats))]
    att=[cats[i][0] for i in range(len(cats))]
    gains=[]
    div_point=[]
    for i in range(1,len(cat)):
        if not att[i]==att[i-1]:
            gains.append(entropy(cat[:i])*float(i)/len(cat)+entropy(cat[i:])*(1-float(i)/len(cat)))
            div_point.append(i)
    return att[div_point[gains.index(min(gains))]]

def grow_tree(data,category,parent,attrs_names):
    if len(set(category))>1:

        division=[]
        for i in range(len(data)):
            if set(data[i])==set("?"):
                division.append(0)
            else:
                if (isnum(data[i])):
                    division.append(gain(category,data[i]))
                else:
                    division.append(gain_ratio(category,data[i]))
        if max(division)==0:
```

```
            num_max=0
            for cat in set(category):
                num_cat=category.count(cat)
                if num_cat>num_max:
                    num_max=num_cat
                    most_cat=cat
            parent.text=most_cat
        else:
            index_selected=division.index(max(division))
            name_selected=str(attrs_names[index_selected])
            if isnum(data[index_selected]):
                div_point=division_point(category,data[index_selected])
                r_son_data=[[] for i in range(len(data))]
                r_son_category=[]
                l_son_data=[[] for i in range(len(data))]
                l_son_category=[]
                for i in range(len(category)):
                    if not data[index_selected][i]=="?":
                        if float(data[index_selected][i])<float(div_point):
                            l_son_category.append(category[i])
                            for j in range(len(data)):
                                l_son_data[j].append(data[j][i])
                        else:
                            r_son_category.append(category[i])
                            for j in range(len(data)):
                                r_son_data[j].append(data[j][i])
                if len(l_son_category)>0 and len(r_son_category)>0:
                    p_l=float(len(l_son_category))/(len(data[index_selected])-
data[index_selected].count("?"))

son=ET.SubElement(parent,name_selected,{'value':str(div_point),"flag":"l","p":str(round(p_l,3))})
                    grow_tree(l_son_data,l_son_category,son,attrs_names)

son=ET.SubElement(parent,name_selected,{'value':str(div_point),"flag":"r","p":str(round(1-p_l,3))})
                    grow_tree(r_son_data,r_son_category,son,attrs_names)
                else:
                    num_max=0
                    for cat in set(category):
                        num_cat=category.count(cat)
                        if num_cat>num_max:
                            num_max=num_cat
                            most_cat=cat
                    parent.text=most_cat
            else:
                for k in set(data[index_selected]):
                    if not k=="?":
                        son_data=[[] for i in range(len(data))]
                        son_category=[]
                        for i in range(len(category)):
                            if data[index_selected][i]==k:
                                son_category.append(category[i])
```

```
                for j in range(len(data)):
                    son_data[j].append(data[j][i])

son=ET.SubElement(parent,name_selected,{'value':k,"flag":"m",'p':str(round(float(len(son_category)
)/(len(data[index_selected])-data[index_selected].count("?")),3))})
            grow_tree(son_data,son_category,son,attrs_names)
    else:
        parent.text=category[0]

def add(d1,d2):
    d=d1
    for i in d2:
        if d.has_key(i):
            d[i]=d[i]+d2[i]
        else:
            d[i]=d2[i]
    return d

def decision(root,obs,attrs_names,p):
    if root.hasChildNodes():
        att_name=root.firstChild.nodeName
        if att_name=="#text":

            return decision(root.firstChild,obs,attrs_names,p)
        else:
            att=obs[attrs_names.index(att_name)]
            if att=="?":
                d={}
                for child in root.childNodes:
                    d=add(d,decision(child,obs,attrs_names,p*float(child.getAttribute("p"))))
                return d
            else:
                for child in root.childNodes:
                    if child.getAttribute("flag")=="m" and child.getAttribute("value")==att or \
                        child.getAttribute("flag")=="l" and float(att)<float(child.getAttribute("value")) or \
                        child.getAttribute("flag")=="r" and float(att)>=float(child.getAttribute("value")):
                        return decision(child,obs,attrs_names,p)
    else:
        return {root.nodeValue:p}

import math
from xml.dom import minidom
from xml.etree import ElementTree as ET

from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils.validation import check_array, check_is_fitted, check_X_y


class C45(BaseEstimator, ClassifierMixin):
    def __init__(self, attrNames=None):
        if attrNames is not None:
```

```
            attrNames = [''.join(i for i in x if i.isalnum()).replace(' ', '_') for x in attrNames]
        self.attrNames = attrNames

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        self.X_ = X
        self.y_ = y
        self.resultType = type(y[0])
        if self.attrNames is None:
            self.attrNames = [f'attr{x}' for x in range(len(self.X_[0]))]

        assert(len(self.attrNames) == len(self.X_[0]))

        data = [[] for i in range(len(self.attrNames))]
        categories = []

        for i in range(len(self.X_)):
            categories.append(str(self.y_[i]))
            for j in range(len(self.attrNames)):
                data[j].append(self.X_[i][j])
        root = ET.Element('DecisionTree')
        grow_tree(data,categories,root,self.attrNames)
        self.tree_ = ET.tostring(root, encoding="unicode")
        return self

    def predict(self, X):
        check_is_fitted(self, ['tree_', 'resultType', 'attrNames'])
        X = check_array(X)
        dom = minidom.parseString(self.tree_)
        root = dom.childNodes[0]
        prediction = []
        for i in range(len(X)):
            answerlist = decision(root,X[i],self.attrNames,1)
            answerlist = sorted(answerlist.items(), key=lambda x:x[1], reverse = True )
            answer = answerlist[0][0]
            prediction.append((self.resultType)(answer))
        return prediction

    def printTree(self):
        check_is_fitted(self, ['tree_'])
        dom = minidom.parseString(self.tree_)
        print(dom.toprettyxml(newl="\r\n"))

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
clf = C45(attrNames=iris.feature_names)
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.45)
clf.fit(X_train, y_train)
```

```
predictions = clf.predict(X_train)
from sklearn import metrics
confusion = metrics.confusion_matrix(y_true = y_train, y_pred = predictions)
confusion
print("Accuracy = ",metrics.accuracy_score(y_true=y_train, y_pred=predictions)*100)
class_wise = metrics.classification_report(y_true=y_train, y_pred=predictions)
print(class_wise)
predictions = clf.predict(X_test)
#predictions
from sklearn import metrics
confusion = metrics.confusion_matrix(y_true = y_test, y_pred = predictions)
confusion
print("Accuracy = ",metrics.accuracy_score(y_true=y_test, y_pred=predictions)*100)
class_wise = metrics.classification_report(y_true=y_test, y_pred=predictions)
print(class_wise)
```

**Dataset:** MNIST Dataset

The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems which contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset.

**Input:**

train - Notepad

File Edit Format View Help

```
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255
 247 127   0   0   0   0   0   0   0   0   0   0   0   0  30  36  94 154
 170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0   0   0
   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82
  82  56  39   0   0   0   0   0   0   0   0   0   0   0   0  18 219 253
 253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  35 241
 225 160 108   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187
```

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn import tree
import os
print(os.listdir("/content/drive/MyDrive"))
df_train = pd.read_csv("/content/drive/MyDrive/mnist_train.csv")
df_test = pd.read_csv("/content/drive/MyDrive/mnist_test.csv")
print(df_train.shape)
print(df_test.shape)
X = []
y = []
for row in df_train.iterrows() :
    label = row[1][0] # label (the number visible in the image)
    image = list(row[1][1:]) # image information as list, without label
    image = np.array(image) / 255
    X.append(image)
    y.append(label)
X = np.array(X)
y = np.array(y)
print(len(X))
print(len(y))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
print(len(X_train), len(y_train))
```

```
print(X_train[1].shape)
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
y_predt = clf.predict(X_train)
print(y_predt[0:20], ".....")
print(y_train[0:20], ".....")
print(metrics.accuracy_score(y_train, y_predt))
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(y_pred[0:20], ".....")
print(y_test[0:20], ".....")
print(metrics.accuracy_score(y_test, y_pred))
```

**Output:**

**Train**

```
[0 1 1 9 0 2 1 8 7 2 8 6 3 5 0 9 3 3 6 7] .....
[0 1 1 9 0 2 1 8 7 7 8 6 1 5 0 9 2 3 6 7] .....
Accuracy =  92.2
```

```
array([[4305,    2,   15,    9,    8,   28,   18,    9,   26,    8],
       [   1, 4952,   33,   19,    6,   13,    8,   20,   33,    8],
       [  34,   18, 4106,   61,   46,   28,   35,   68,   74,   17],
       [  17,   19,   61, 4084,   13,  189,    6,   66,   74,   84],
       [   9,   25,   30,   14, 3926,   11,   25,   36,   48,  275],
       [  40,   21,   16,   98,   24, 3697,   25,   11,   45,   61],
       [  35,   33,   28,   14,   51,   61, 4152,    5,   53,    4],
       [   4,   14,   60,   27,   40,    8,    2, 4290,   33,  152],
       [  27,   54,   54,   52,   26,   53,   38,   24, 3982,   96],
       [  14,   13,   19,   69,  147,   72,    1,   73,   66, 3996]])
```

```
              precision    recall  f1-score   support

           0       0.96      0.97      0.97      4428
           1       0.96      0.97      0.97      5093
           2       0.93      0.92      0.92      4487
           3       0.92      0.89      0.90      4613
           4       0.92      0.89      0.90      4399
           5       0.89      0.92      0.90      4038
           6       0.96      0.94      0.95      4436
           7       0.93      0.93      0.93      4630
           8       0.90      0.90      0.90      4406
           9       0.85      0.89      0.87      4470

    accuracy                           0.92     45000
   macro avg       0.92      0.92      0.92     45000
weighted avg       0.92      0.92      0.92     45000
```

**Test:**

```
9 print("Accuracy = ",metrics.accuracy_score(y_test, y_predtt)*100)
```

```
[7 3 8 9 3 9 7 7 5 4 4 5 6 9 1 7 4 8 0 7] .....
[7 3 8 9 3 9 7 7 5 4 2 5 6 8 1 7 4 8 0 7] .....
Accuracy =  87.11333333333333
```

```
array([[1377,    0,   17,    9,    8,   18,   35,    5,   20,    6],
       [   1, 1598,    7,   13,    6,    7,    5,    4,    7,    1],
       [  18,   18, 1224,   35,   20,   15,   32,   45,   50,   14],
       [  10,   12,   43, 1262,   16,   71,   14,   25,   36,   29],
       [   9,    5,    9,   10, 1257,   10,   26,   22,   19,   76],
       [  23,    8,   17,   57,   14, 1155,   33,   10,   38,   28],
       [  16,    3,   32,   10,   17,   27, 1342,    4,   28,    3],
       [   6,   20,   47,   15,   20,    4,    2, 1482,    9,   30],
       [  23,   29,   45,   52,   33,   39,   27,   12, 1155,   30],
       [  12,    7,   17,   43,   89,   20,   13,   33,   30, 1215]])
```

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.92 | 0.92 | 0.92 | 1495 |
| 1 | 0.94 | 0.97 | 0.95 | 1649 |
| 2 | 0.84 | 0.83 | 0.84 | 1471 |
| 3 | 0.84 | 0.83 | 0.83 | 1518 |
| 4 | 0.85 | 0.87 | 0.86 | 1443 |
| 5 | 0.85 | 0.84 | 0.84 | 1383 |
| 6 | 0.88 | 0.91 | 0.89 | 1482 |
| 7 | 0.90 | 0.91 | 0.90 | 1635 |
| 8 | 0.83 | 0.80 | 0.81 | 1445 |
| 9 | 0.85 | 0.82 | 0.83 | 1479 |
| accuracy | | | 0.87 | 15000 |
| macro avg | 0.87 | 0.87 | 0.87 | 15000 |
| weighted avg | 0.87 | 0.87 | 0.87 | 15000 |

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

# TABULAR INFERENCE

| ALGORITHM | ID3 | C4.5 | CART |
|---|---|---|---|
| PRECISION | 0.90 | 0.93 | 0.85 |
| RECALL | 0.92 | 0.92 | 0.79 |
| F1-SCORE | 0.90 | 0.92 | 0.79 |
| ACCURACY | 93.3 % | 92.64 % | 95.2% |

## ALGORITHM 1 -ID3

| Dataset | ACCURACY (%) | PRECISION | RECALL | F1-SCORE |
|---|---|---|---|---|
| **Train** | 93.3 | 0.93 | 0.94 | 0.93 |
| **Test** | 93.3 | 0.90 | 0.92 | 0.90 |

## ALGORITHM 2 – C4.5

| Dataset | ACCURACY (%) | PRECISION | RECALL | F1-SCORE |
|---|---|---|---|---|
| **Train** | 100 | 1 | 1 | 1 |
| **Test** | 92.64 | 0.93 | 0.92 | 0.92 |

## ALGORITHM 3 – CART

| Dataset | ACCURACY (%) | PRECISION | RECALL | F1-SCORE |
|---|---|---|---|---|
| **Test** | 95.2% | 0.85 | 0.79 | 0.79 |

## MNIST DATASET

| Dataset | ACCURACY (%) | PRECISION | RECALL | F1-SCORE |
|---|---|---|---|---|
| **Train** | 92.2 | 0.92 | 0.92 | 0.92 |
| **Test** | 87.11 | 0.87 | 0.87 | 0.87 |

## MNIST DATASET - CRITERION USED

| CRITERION | ACCURACY(%) |
|---|---|
| **Gini** | **87.11** |
| **Entropy** | **86.7** |

## MNIST DATASET - SPLITTER USED

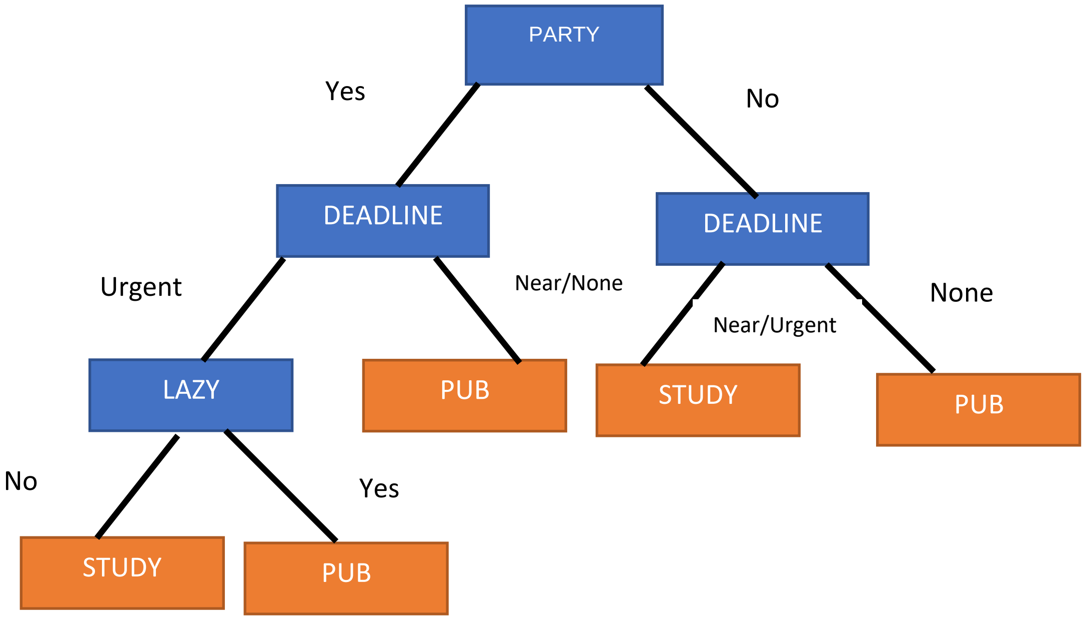
| SPLITTER | ACCURACY(%) |
|---|---|
| **Best** | **87.11** |
| **Random** | **85.4** |

## MNIST DATASET – MAX DEPTH USED

| MAX DEPTH | ACCURACY(%) |
|---|---|
| **None** | **86.54** |
| **13** | **87.11** |
| **18** | **85.4** |

**Inference: Thus, Decision tree algorithms ID3, C4.5 and CART were implemented. It was applied on the MNIST dataset as well. The accuracy obtained by varying different hyperparameters are tabulated. The tree obtained is shown.**

## Expected Tree for Party Dataset of ID3 Algorithm:




## Obtained Tree for Iris Dataset of C4.5 Algorithm: