**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:**

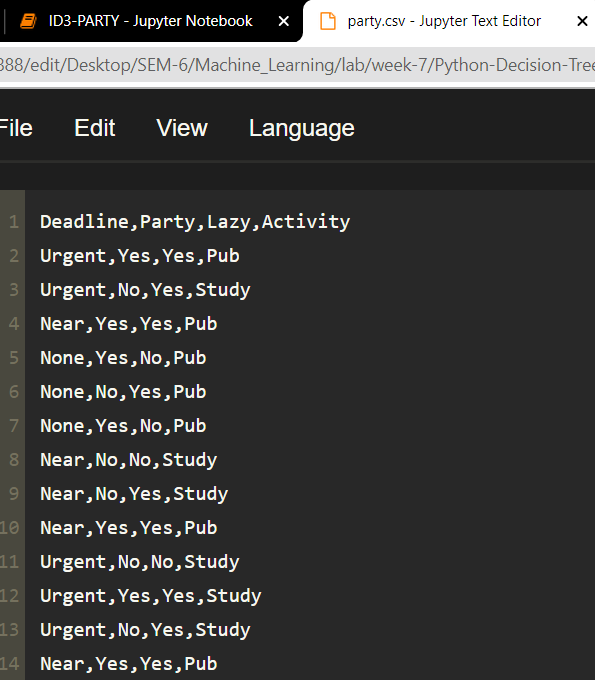
**Features:**

Deadline – Takes values Urgent, Near, None

Party – Takes values Yes,No

Lazy – Takes values Yes,No

Output Feature : Activity – Pub, Study



**OUTPUT:**

TOTAL ENTROPY = 0.98

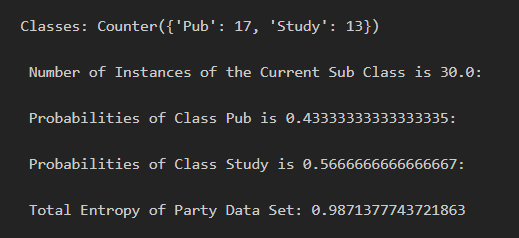
For the root:

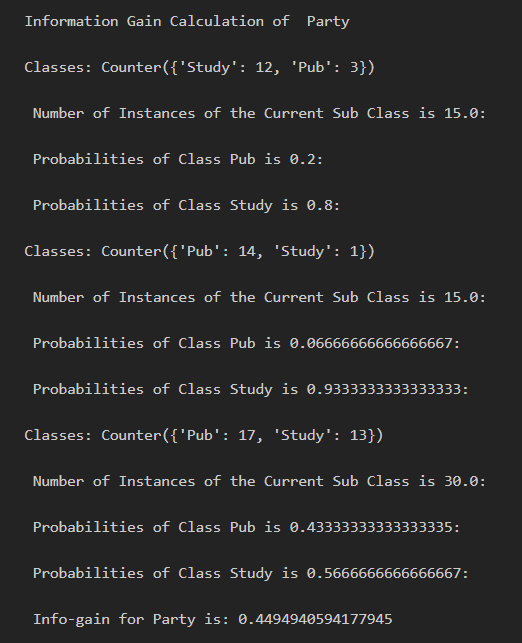
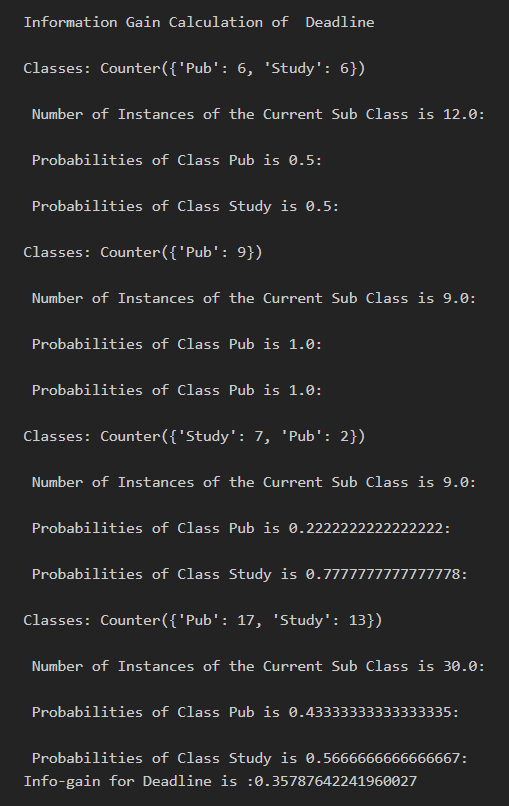
INFO GAIN (DEADLINE) = 0.357

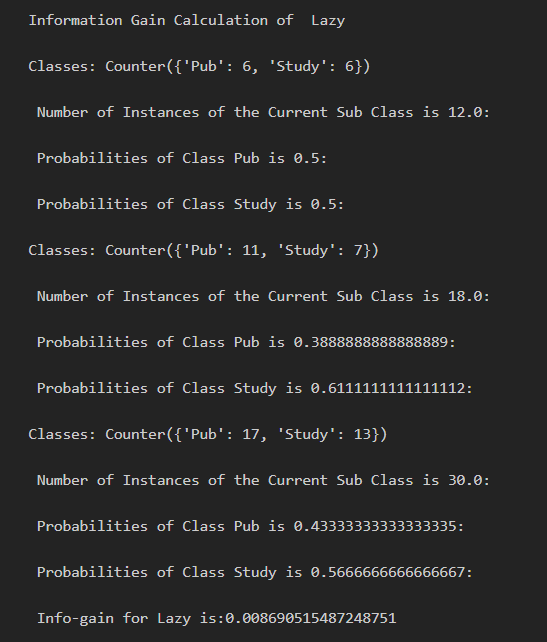
INFO GAIN (PARTY) = 0.449

INFO GAIN (LAZY)= 0.008

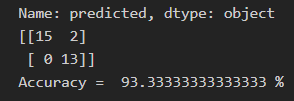
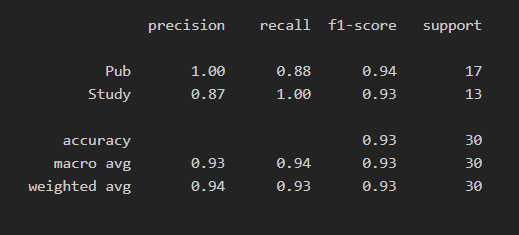
Similarly ID3 algorithm is used for subsequent depths.



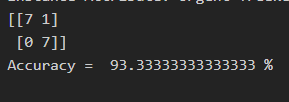
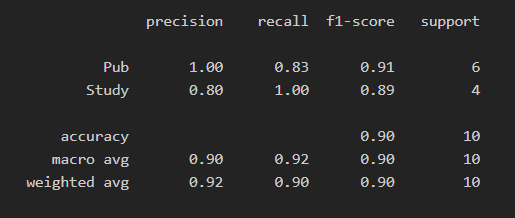




**Train**



**Test**



**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['predicted'])\*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'])

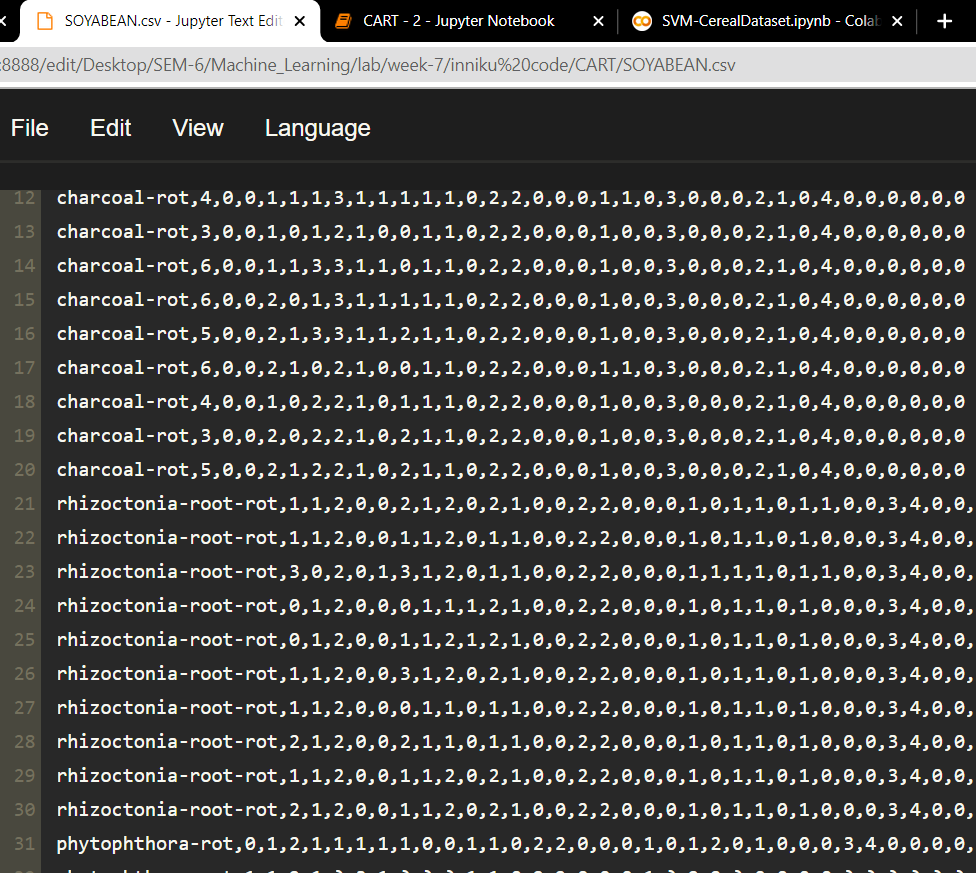
print(class\_wise)

**CART ALGORITHM**

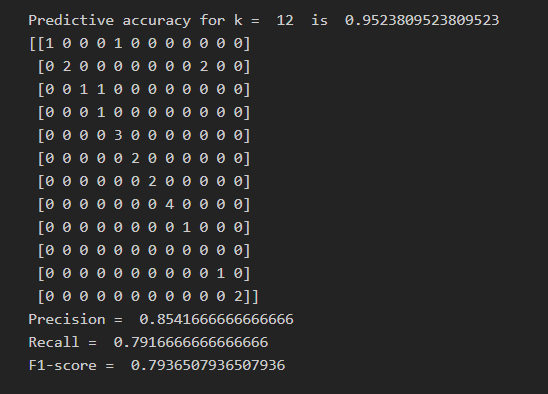
**DATASET – Soyabean dataset**

**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:**



**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**

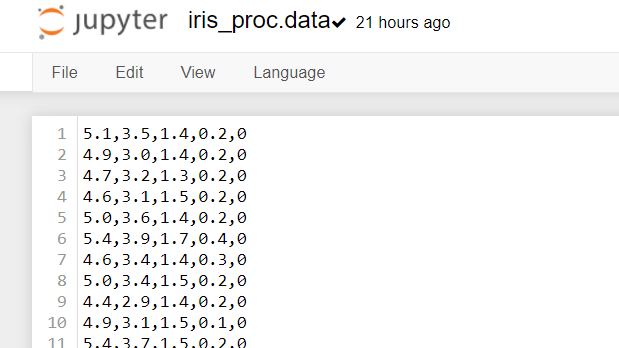
**Dataset : Iris**

**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

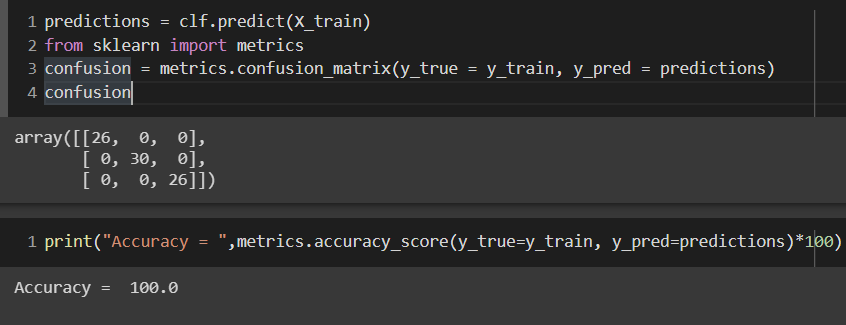
* sepal length in cm,
* sepal width in cm,
* petal length in cm,
* petal width in cm,

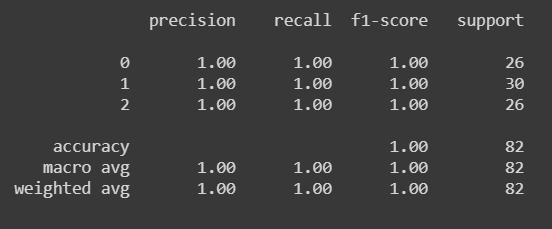


**Output**: decision: Multiclass classification among 3 classes of flowers: Iris Setosa, Iris Versicolour, Iris Virginica.

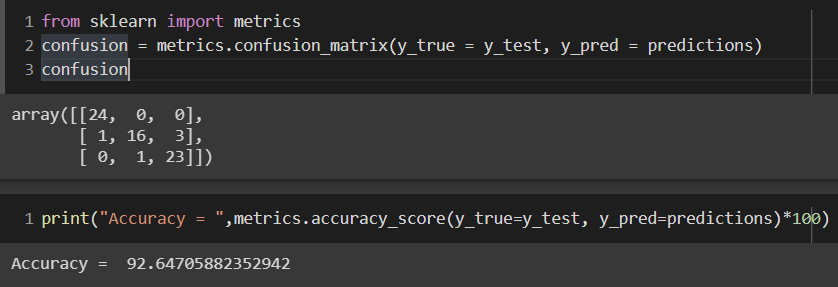
**Output:**

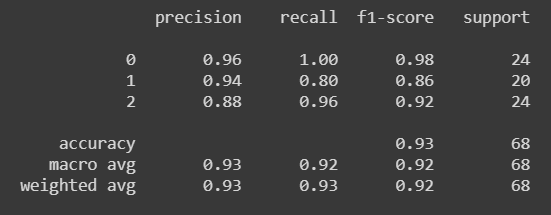
**Train:**





**Test:**





**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)

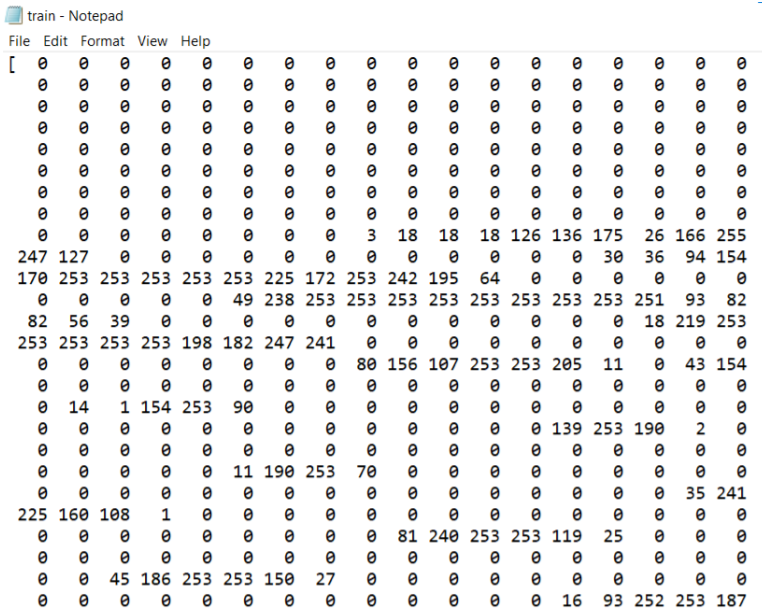
class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=predictions)

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:**



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], ".....")

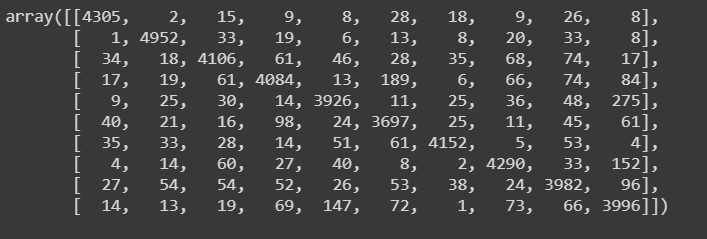
print(y\_test[0:20], ".....")

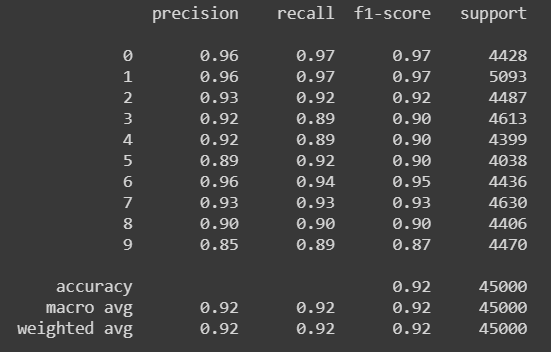
print(metrics.accuracy\_score(y\_test, y\_pred))

**Output:**

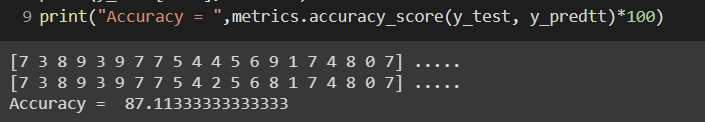
**Train**

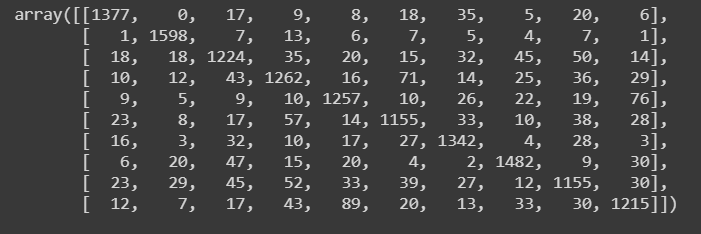


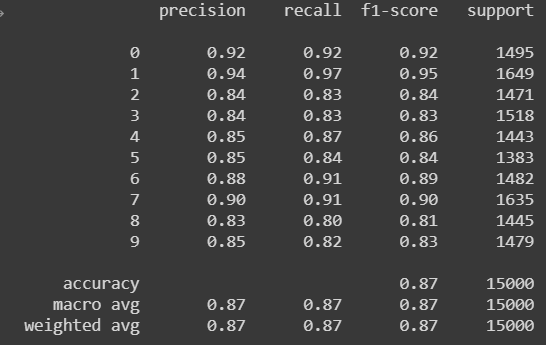


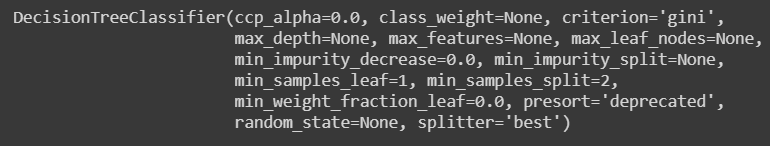


**Test:**









**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:**

PUB

STUDY

PUB

STUDY

PUB

LAZY

DEADLINE

DEADLINE

PARTY

No

Yes

Near/None

None

Urgent

Near/Urgent

No

Yes

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

Yes

Yes

Yes

Yes

No

No

No

No

No

Yes

Iris Versicolor

Iris Versicolor

Iris Verginica

SepalWidth = 3.2cm

Iris Verginica

Iris Verginica

PetalLength = 4.7 cm

PetalLength = 5.1 cm

Iris Setosa

PetaWidth = 1.7 cm

PetalLength = 3.3 cm