Decision Tree Classifier from Scratch

Assignment 1

Emirhan Koc
emirhan.koc@bilkent.edu.tr

Electrical and Electronics Engineering, Bilkent University

March 14, 2022

Contents

1 Description

In this assignment, I worked on implementation of **Decision Tree Classifier** and conducted experiments on *Thyroid data set* which is taken from UCI repository. This data set contains separate training ("ann-train.data") and test ("ann-test.data") sets. The training set contains 3772 instances and the test set contains 3428 instances. There are a total of 3 classes. In the data files, each line corresponds to an instance that has 21 features (15 binary and 6 continuous features) and 1 class label. In the first part of this homework, I implemented a decision tree using ML packages. In the second part, I implemented the decision tree from scratch .In the third part of this homework, I modified the decision tree in second part considering the cost of using (extracting) the features. The cost of using each feature is given in another file ("ann-thyroid.cost"). It does not include the cost of the 21st feature because it is a combination of the other features. The 21st feature is defined using the 19th and 20th features. This means that I do not incur additional cost for this feature if the 19th and 20th features have already been extracted. Otherwise, I have to pay for the cost of the unextracted feature(s).

2 Question 1

2.1 Non-Pruning vs Pruning

Pruning is a data compression method that reduces the size of decision trees to avoid overfitting by removing sections of the tree that are non-critical and redundant to classify instances. Pruning diminishes the complexity of the final classifier, and thus develop the inference performance by the reduction of overfitting. In this part, I created different models with pruning and non-pruning by setting *min impurity decrease* option in *sklearn*. You can see how pruning works in this package:

The weighted impurity decrease equation is the following:

$N_t/N * (impurity - N_{tR}/N_t * right - impurity - N_{tL}/N_t * left - impurity)$ where N_t is the number of samples at the current node, N_{tL} is the number of samples in the left child, and N_{tR} is the number of samples in the right child.

In this part, I set *random state* as 12 and impurity criterion as *entropy*. I also set the *maximum depth* as 8.

Here, we can see that test accuracies differ significantly with pruning compared to withour pruning. It shows that the model was overfitting before and we avoid overfitting. You can see

Options	Train	Test
Pruning	100	99.27
Non-Pruning	99.33	98.8

Table 1: Pruning vs Non-Pruning Accuracy

results in Appendix.

2.2 Normalization

Here, I standardized to data make all features in the form similar measure in high dimensional space. It is scaling technique where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation.

$\tilde{X} = \frac{(X-\mu)}{\sigma}$ is the representation of standardizing the data. After standardizing data, I did not see significant change in both training and test accuracies. You can see results in Appendix.

2.3 Handling Imbalanced Distribution

Imbalanced distribution of classes are a subtle problem in model creation in machine learning application. The model may tend to make predictions in favor of the overwhelming class as the accuracy will nevertheless be high even other class predictions are wrong. To solve this problem, I created a new dataset taking equal amount of data from each class and trained the model with this data. I set the amount of data from each class by setting it to number of instances of the smallest class and it was 93. In this setting, even the training accuracy remains similar, the test accuracy diminished considerably. You can see results in Appendix.

3 Question 2

3.1 Implementation of Decision Tree with Prepruning from Scratch

Prepruninig in decision tree make stop the tree earlier before it overfits the training sample. There are two options for pruning tree such that depth based and information gain based. In this question, I implemented considering gain such that if the information gain for the current node, I stop growing the tree and make it leaf node. In this case, labeling is made in favor of dominating class. I would like to discuss the algorithmic details:

Firstly, I choice an impurity options from these two options: **Entropy** and **Gini Index** :

$$I(m)_{gini} = 1 - \sum_{i=1}^{i=N_c}(P(C_i)_m)^2$$
$$I(m)_{entropy} = -\sum_{i=1}^{i=N_c}(P(C_i)_m * logP(C_i)_m)$$

where N_c is the number of instances of a class in the curret node.

3.2 Logic Behind Tree Creation

Imagine you are in a certain node in the tree and would like the tree to grow further. First, we need to decide which feature to choose and which value to compare. It is worth note that this is the most exhaustive and computationally costly part.I created an exhaustive search algorithm to choose the best option for further growing the node. Here, what I did is as follows:

```
def exhaustive_search(attributes,label,impurity_c

    splitCheckAll= np.Inf
    gainAll=0
    valueSplitAll=0

    featureIndexSplit=0

    for ft in range(0,attributes.shape[1]):

        splitCheck= np.Inf

        for val in attributes[:,ft]:
            isSplit,gainCheck = LeftRightSplit(at

            if isSplit < splitCheck:
                splitCheck=isSplit
                tempSplitCheck=splitCheck
                splitValue= val
                gainSplit= gainCheck


        if tempSplitCheck < splitCheckAll:
            splitCheckAll=tempSplitCheck
            featureIndexSplit=ft
            valueSplitAll= splitValue
            gainAll=gainSplit


    return splitCheckAll,gainAll,valueSplitAll,fe
```

For all the attributes and values, search for the value that gives the smallest average left-right split impurity. Afterward, according to comparison of value with all values in its attributes,

we create left and right nodes and grow three. In fact, this process is applied from root node to leaf nodes. Additionaly, leaf node may be decided whether the desire maximum depth is reached or the node is pure.

Also, as it is state above, I stop the tree to grow by prepruning according comparison of information gain in current node with pruning value. If the gain is less than pruning value, I set the node as leaf node and label in favor of outnumbered class.

3.3 Results

I implemented a decision tree with prepruning options from scratch and results made me surprised as it was very close to results obtained in the first question. I obtained results below:

	Train	Test
Total	99.97	99.3
Class 1	100	95.89
Class 2	100	99.4
Class 3	99.97	99.4

Table 2: Total and Class Based Accuracy

4 Cost Sensitive Decision Tree Classifier from Scratch

In the description part, it is stated that features have some cost to be extracted and the last feature is the mixture of 19th and 20th features. In this case, abovementioned splitting criteria, which it was choosing the minimum average splitting impurity, is revisited and modified. A score is defined such that it equals to square of gain divided by cost in certain node for certain feature. Instead of using maximum value of this score, I multiplied it with -1 and consider negative it to compare according to minimum value. You can see the results as well as average cost for classes.

4.1 Results

	Train	Test
Total	99.17	98.59
Class 1	88.17	84.93
Class 2	100	100
Class 3	99.42	98.83

Table 3: Total and Class Based Accuracy

	Train	Test
Class 1	58.35	57.6
Class 2	49.37	49.6
Class 3	23.0	23.15

Table 4: Class Based Costs

Acknowledgements

In this assignment, I benefited informations mostly from course slides, Tom Mitchell's lecture videos and notes in CMU and a blog site Analytics Vidhya.

Question_1

March 14, 2022

[97]:
```
import pandas as pd
import numpy as np
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.preprocessing import StandardScaler
from random import sample
import os
```

[98]:
```
cur_dir = os.getcwd()
train_file_path = os.path.join(cur_dir, 'ann-train.data')
test_file_path = os.path.join(cur_dir, 'ann-test.data')
feature_file_path = os.path.join(cur_dir, 'ann-thyroid.cost')
```

[99]:
```
# Organize features

features = pd.read_csv(feature_file_path, sep =":", header =None)
features = features[features.columns[:1]]
features = features.append(pd.Series("mixed"),ignore_index= True)
features_list = np.squeeze(features.to_numpy())
features = features.to_dict()[0]
```

[100]:
```
# Prepare tranining data
data_train = pd.read_csv(train_file_path, sep =" ", header =None)
train_data= data_train[data_train.columns[0:21]]
train_data = train_data.rename(columns=features,inplace =False)
label_train = data_train[data_train.columns[21]].astype("category")
```

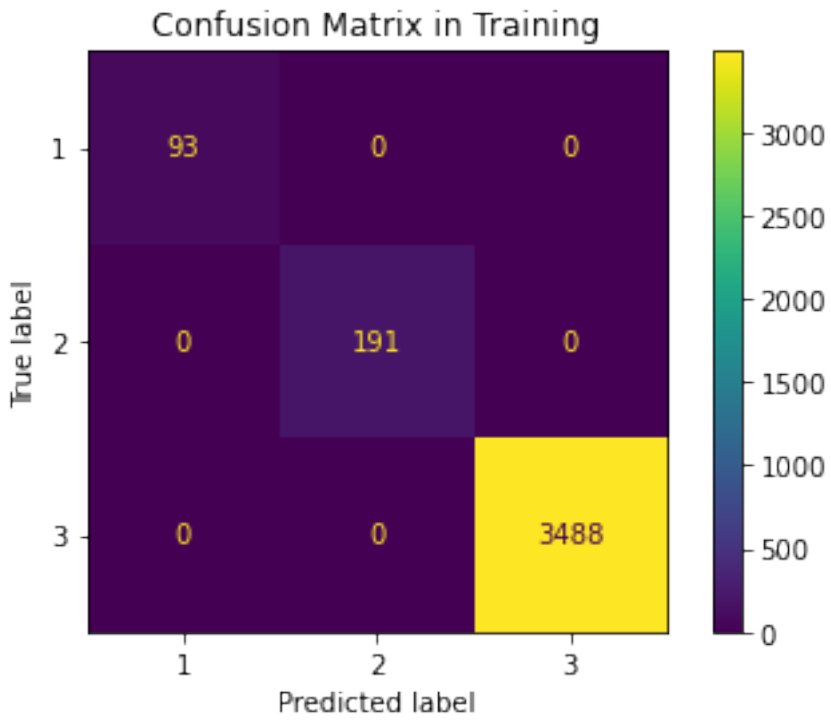
[101]:
```
# Prepare test data

data_test = pd.read_csv(test_file_path, sep =" ", header =None)
test_data= data_test[data_test.columns[0:21]]
test_data = test_data.rename(columns=features,inplace =False)
label_test=data_test[data_train.columns[21]].astype("category")
```

[108]:
```
# Create Decision Tree Classifier


def createTree(data,label, criterion="gini",splitter="best",random_state =3,␣
 ↪max_depth=15,inpurity=0):
    decision_tree=DecisionTreeClassifier(criterion= criterion,␣
 ↪splitter=splitter, random_state=random_state,␣
 ↪max_depth=max_depth,min_impurity_decrease=inpurity)
    decision_tree = decision_tree.fit(data,label)

    fig, ax = plt.subplots(figsize=(20,20))
    tree.plot_tree(decision_tree,filled= True,feature_names= features_list)
    plt.show()
    return decision_tree



def classificationAccuracy(model, data,label, mode = "Training"):


    label= label.astype(int)
    preds=model.predict(data).astype(int)
    #print(preds.shape)
    pred_accuracy = accuracy_score(preds,label)

    print( mode, "  accuracy is : ", 100*pred_accuracy)

    #class based accuracy
    class_labels = np.sort(label.unique())
    class_accuracy = []

    for cl in class_labels:
        class_pred= model.predict(data.iloc[label.index[label==cl]]).astype(int)
        class_acc= accuracy_score(class_pred, label.iloc[label.index[label==cl]])
        print(mode, " accuracy for class ", cl, "is : ", class_acc*100 )
        class_accuracy.append(class_acc)



    # Confusion Matrix Creation

    plot_confusion_matrix(model, data, label)
    plt.title("Confusion Matrix in " +  mode)
    plt.show()
```

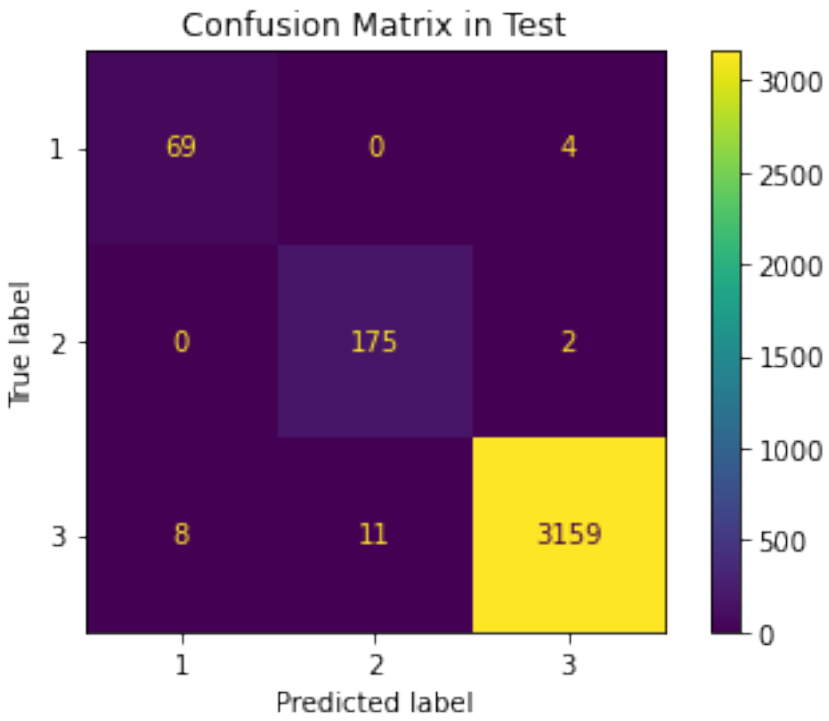
1 Desicion Tree without Pruning

[109]:
```
# Without pruning
decision_tree= createTree(train_data,label_train,"entropy","best",12,8,0.0)
classificationAccuracy(decision_tree,train_data,label_train,"Training")
classificationAccuracy(decision_tree,test_data,label_test,"Test")
```


```
Training   accuracy is :  100.0
Training  accuracy for class  1 is :  100.0
Training  accuracy for class  2 is :  100.0
Training  accuracy for class  3 is :  100.0
```

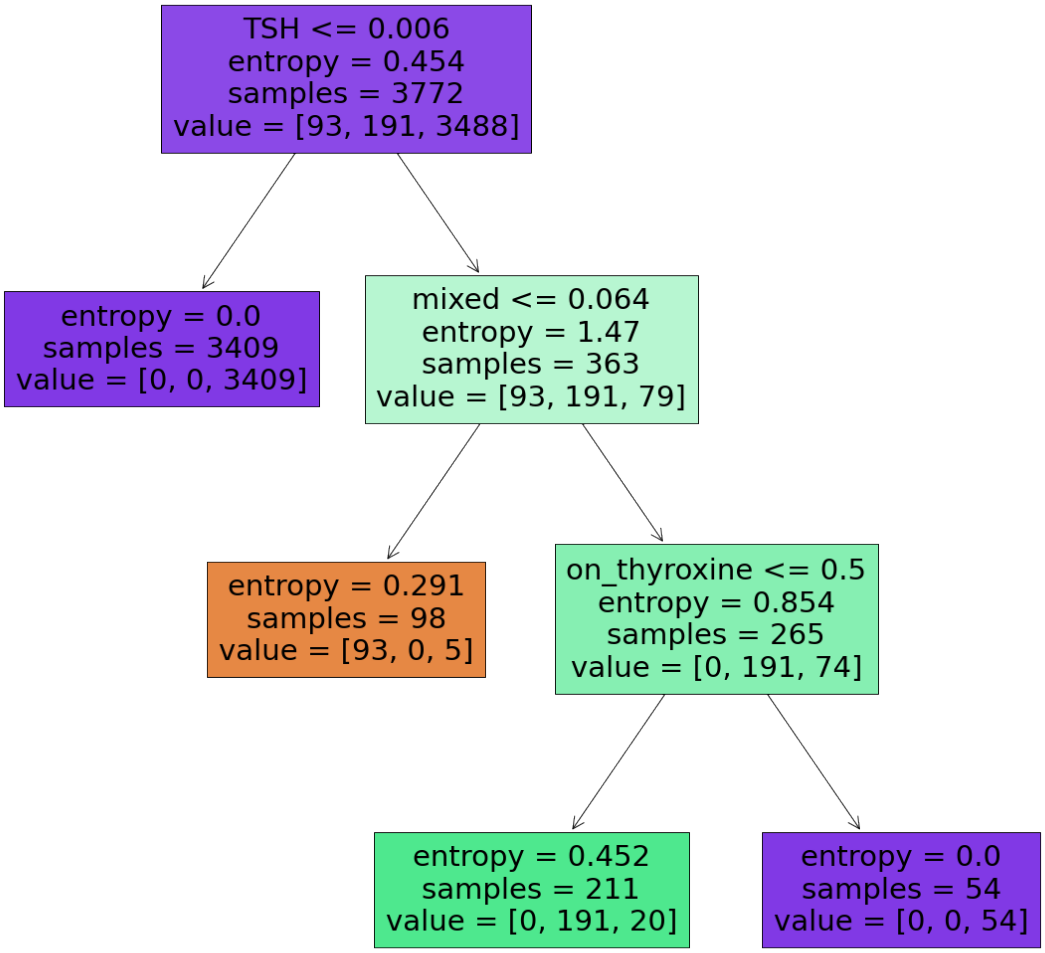



```
Test   accuracy is :  99.27071178529755
Test  accuracy for class  1 is :  94.52054794520548
Test  accuracy for class  2 is :  98.87005649717514
Test  accuracy for class  3 is :  99.40213971050976
```

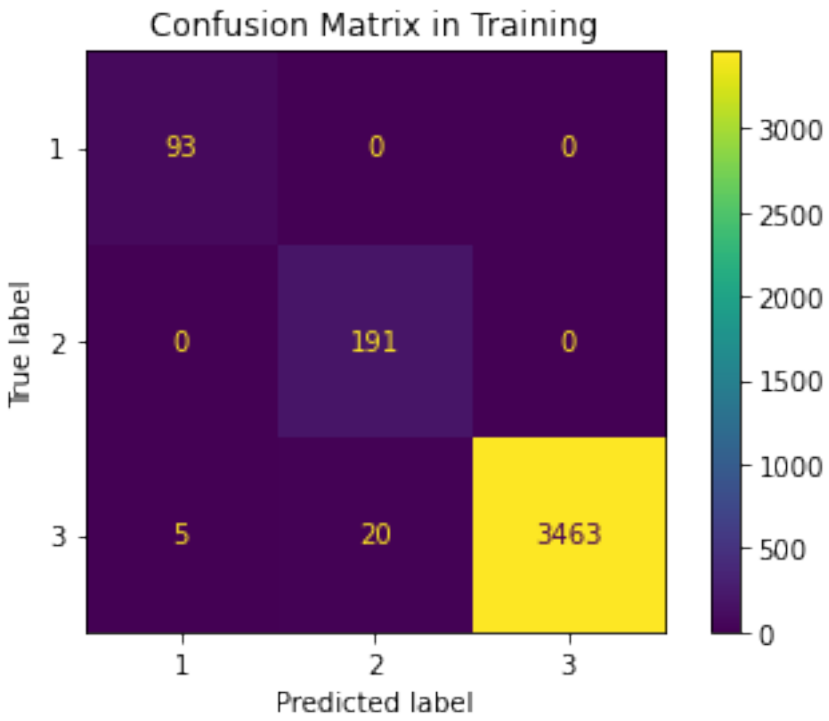


2 Decision Tree with Pruning

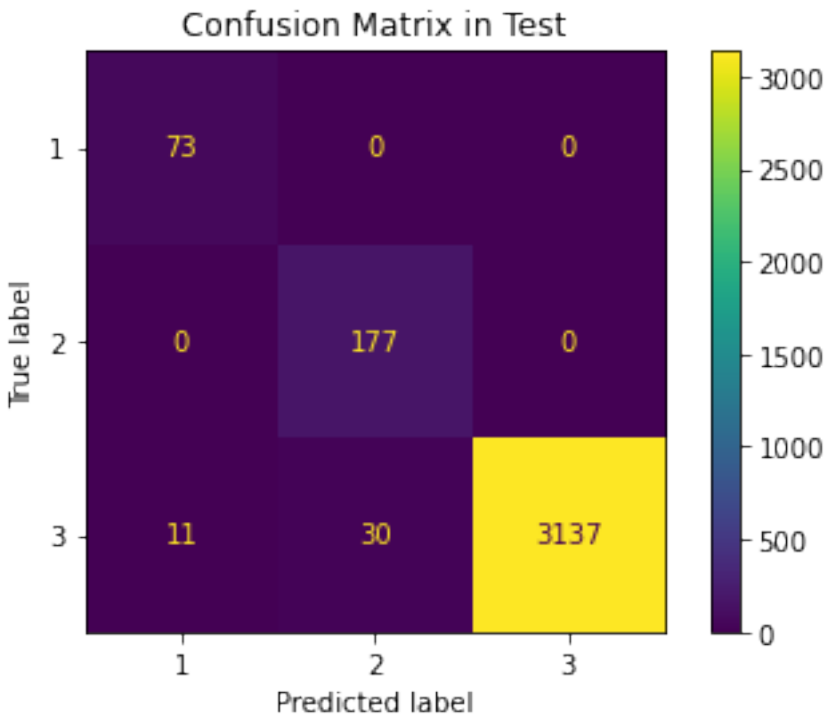
[110]:
```
# With pruning
decision_tree= createTree(train_data,label_train,"entropy","best",12,8,0.025)
classificationAccuracy(decision_tree,train_data,label_train,"Training")
classificationAccuracy(decision_tree,test_data,label_test,"Test")
```




```
Training   accuracy is :  99.33722163308589
Training  accuracy for class  1 is :  100.0
Training  accuracy for class  2 is :  100.0
Training  accuracy for class  3 is :  99.28325688073394
```



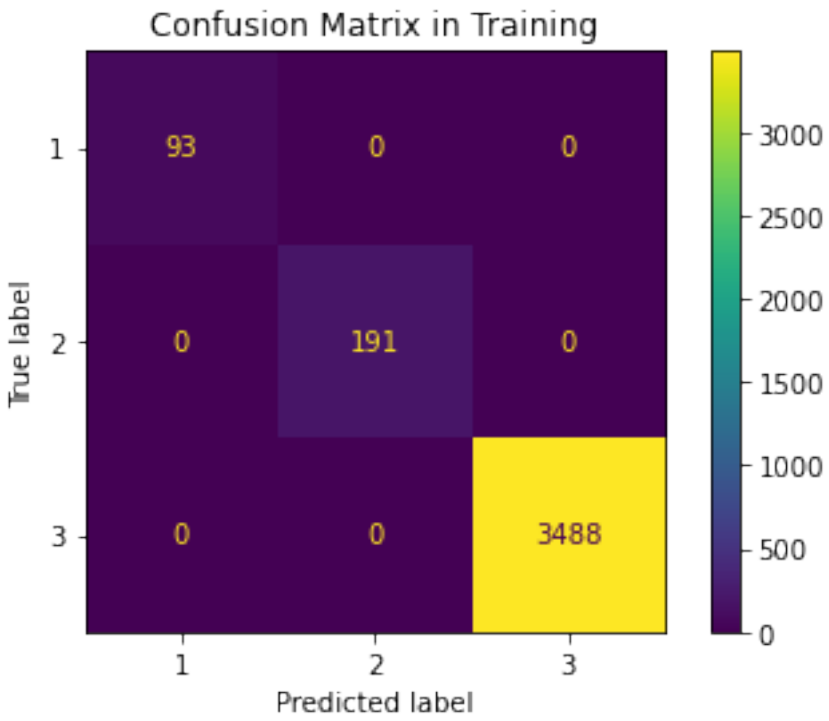

```
Test   accuracy is :  98.80396732788797
Test  accuracy for class  1 is :  100.0
Test  accuracy for class  2 is :  100.0
Test  accuracy for class  3 is :  98.7098804279421
```



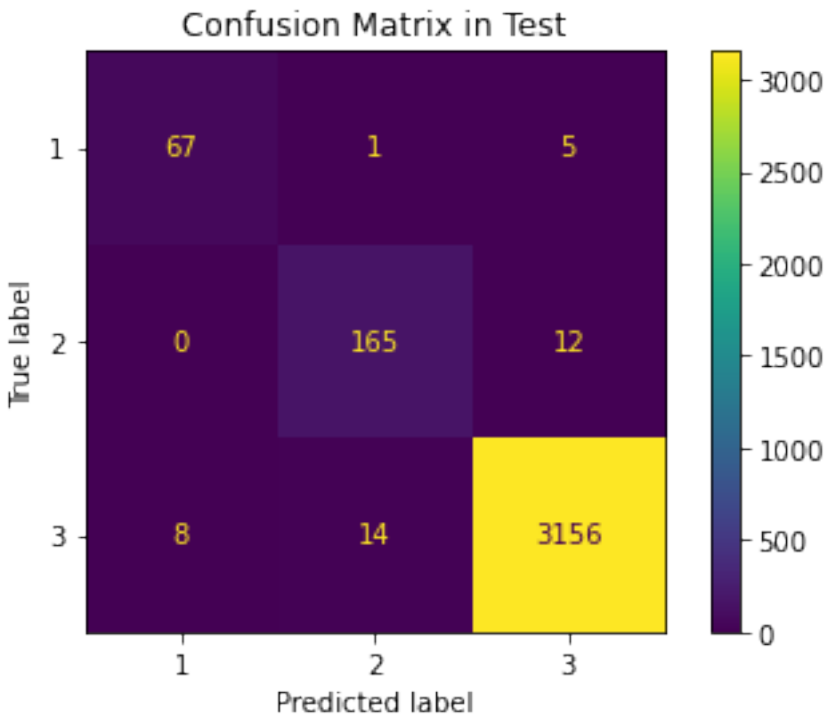
3 Normalized Decision Tree

[111]:
```
# Normalized
scaler = StandardScaler()
train_norm = scaler.fit_transform(train_data)
train_norm= pd.DataFrame(train_norm)
test_norm = scaler.fit_transform(test_data)
test_norm= pd.DataFrame(test_norm)


decision_tree= createTree(train_norm,label_train,"entropy","best",5,10,0)
classificationAccuracy(decision_tree,train_norm,label_train,"Training")
classificationAccuracy(decision_tree,test_norm,label_test,"Test")
```


```
Training   accuracy is :  100.0
Training  accuracy for class  1 is :  100.0
Training  accuracy for class  2 is :  100.0
Training  accuracy for class  3 is :  100.0
```



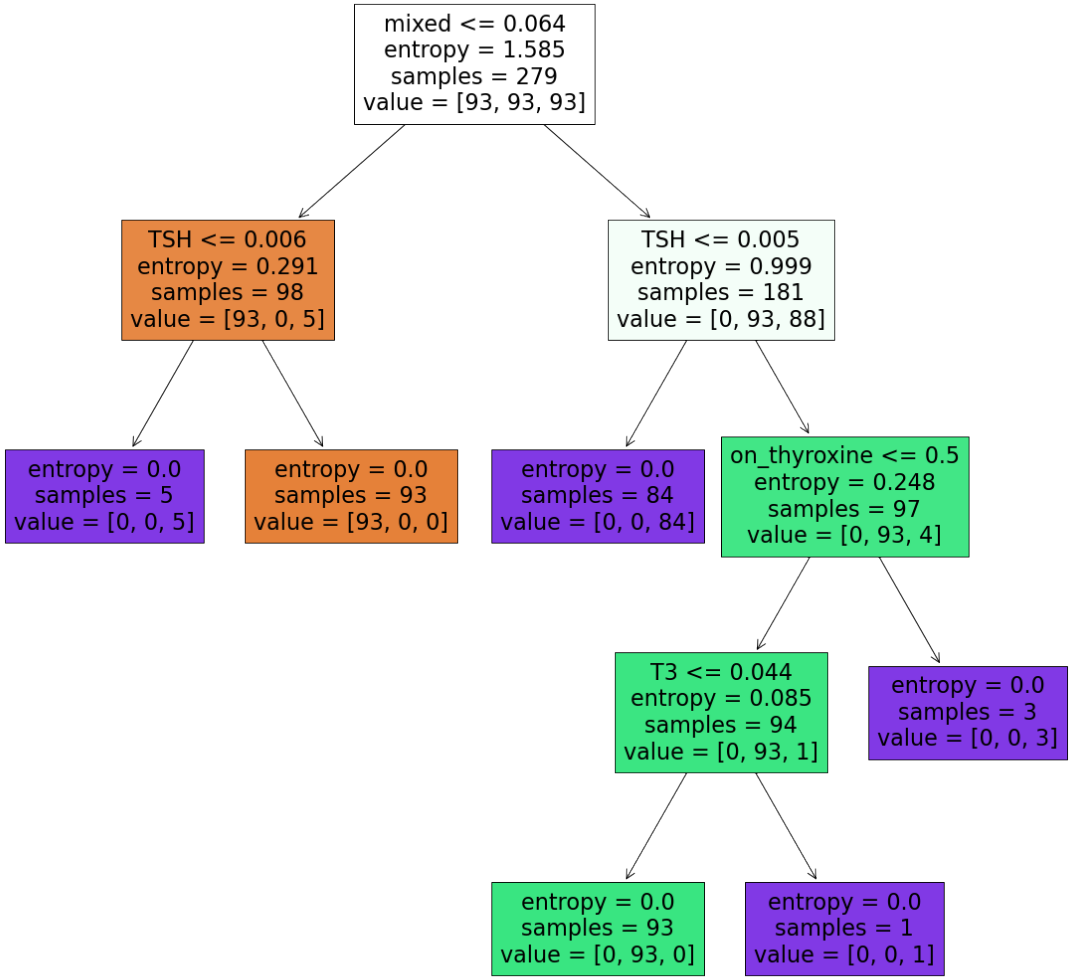

```
Test   accuracy is :  98.83313885647608
Test  accuracy for class  1 is :  91.78082191780823
Test  accuracy for class  2 is :  93.22033898305084
Test  accuracy for class  3 is :  99.30774071743235
```



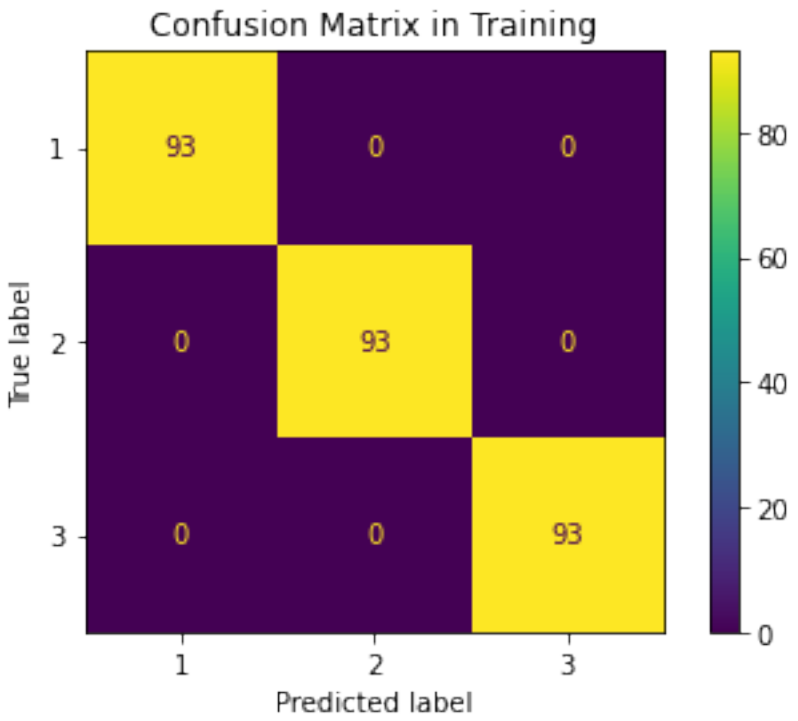
[106]:
```
# Balanced Classes

def balanceData(data,label):


    min_class = label.value_counts().min()
    numClass = len(label.value_counts())
    class_labels = np.sort(label.unique())
    balanced = pd.DataFrame(columns=features_list)
    balanced_label = np.repeat(np.array([[1],[2],[3]]),min_class)
    balanced_label= pd.Series(balanced_label)



    for cl in class_labels:
        perm = sample(list(label.index[label==cl]),min_class)
        d=data.iloc[perm]
        balanced= balanced.append(d,ignore_index=True)

    return balanced,balanced_label
```

Balanced Model Decision Tree

[112]:
```
train_data_bal,label_bal= balanceData(train_data,label_train)
decision_tree= createTree(train_data_bal,label_bal,"entropy","best",12,8,0)
classificationAccuracy(decision_tree,train_data_bal,label_bal,"Training")
classificationAccuracy(decision_tree,test_data,label_test,"Test")
```




```
Training   accuracy is :  100.0
Training  accuracy for class  1 is :  100.0
Training  accuracy for class  2 is :  100.0
Training  accuracy for class  3 is :  100.0
```




```
Test   accuracy is :  97.87047841306884
Test  accuracy for class  1 is :  100.0
Test  accuracy for class  2 is :  100.0
Test  accuracy for class  3 is :  97.70295783511642
```


Confusion Matrix in Test
73
0
0
0
177
0
13
60
3105
1
2
3
True label
1
2
3
Predicted label
3000
2500
2000
1500
1000
500
0

End

[ ]:

[ ]:

[ ]:

Question_2

March 14, 2022

Question 2

[59]:
```
import pandas as pd
import numpy as np
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score,ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.preprocessing import StandardScaler
from random import sample
import os
```

[60]:
```
cur_dir = os.getcwd()
train_file_path = os.path.join(cur_dir, 'ann-train.data')
test_file_path = os.path.join(cur_dir, 'ann-test.data')
feature_file_path = os.path.join(cur_dir, 'ann-thyroid.cost')
```

[61]:
```
# Organize features

features = pd.read_csv(feature_file_path, sep =":", header =None)
features = features[features.columns[:1]]
features = features.append(pd.Series("mixed"),ignore_index= True)
features_list = np.squeeze(features.to_numpy())
features = features.to_dict()[0]
```

[62]:
```
# Prepare tranining data
data_train = pd.read_csv(train_file_path, sep =" ", header =None)
train_data= data_train[data_train.columns[0:21]]
train_data = train_data.rename(columns=features,inplace =False) # This is pandas␣
 ↪version

np_train_data= train_data.to_numpy() # This is numpy version
label_train = data_train[data_train.columns[21]].astype("category")
np_label_train = label_train.to_numpy()
# Prepare test data
```

```
data_test = pd.read_csv(test_file_path, sep =" ", header =None)
test_data= data_test[data_test.columns[0:21]]
test_data = test_data.rename(columns=features,inplace =False)

np_test_data = test_data.to_numpy()
label_test=data_test[data_train.columns[21]].astype("category")
np_label_test = label_test.to_numpy()
```

[63]:
```
#Define some functions
def impurityCalculation(impurity_choice, labelsOfnode):

    labels, numLabels = np.unique(labelsOfnode, return_counts = True)
    total_labels= np.sum(numLabels)
    p_numLabels = numLabels/total_labels

    if impurity_choice == "entropy":
        entropy = 0
        total_entropy = np.sum([entropy+ (-np.log2(ii)*ii)  for ii in␣
 ↪p_numLabels])
        total_impurity = total_entropy


    elif impurity_choice == "gini":
        entropy = 0
        gini= (1/2)*(1-np.sum(np.square(p_numLabels)))
        total_impurity = gini

    else:
        raise Exception("Sorry", impurity_choice, "is not an impurity type. ")

    return total_impurity

def LeftRightSplit(attribute, label, value,impurity_choice):


    total_entropy = impurityCalculation(impurity_choice,label)

    R_split = label[attribute>value]
    L_split = label[attribute<=value]
    L_prop= len(L_split)/len(label)
    R_prop= 1- L_prop

    L_impurity = impurityCalculation(impurity_choice,L_split)
    R_impurity = impurityCalculation(impurity_choice,R_split)

    split_impurity = L_prop*L_impurity + R_prop*R_impurity
    informationGain= total_entropy-split_impurity
```

```
    return split_impurity, informationGain


def exhaustive_search(attributes,label,impurity_choice):

    splitCheckAll= np.Inf
    gainAll=0
    valueSplitAll=0

    featureIndexSplit=0

    for ft in range(0,attributes.shape[1]):

        splitCheck= np.Inf

        for val in attributes[:,ft]:
            isSplit,gainCheck = LeftRightSplit(attributes[:
 ↪,ft],label,val,impurity_choice)

            if isSplit < splitCheck:
                splitCheck=isSplit
                tempSplitCheck=splitCheck
                splitValue= val
                gainSplit= gainCheck


        if tempSplitCheck < splitCheckAll:
            splitCheckAll=tempSplitCheck
            featureIndexSplit=ft
            valueSplitAll= splitValue
            gainAll=gainSplit


    return splitCheckAll,gainAll,valueSplitAll,featureIndexSplit


def splitNode(attributes,label,impurity_choice):

    bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex =␣
 ↪exhaustive_search(attributes,label,impurity_choice)

    node_impurity = impurityCalculation(impurity_choice,label)

    left_node = attributes[attributes[:,bestFeatureIndex]<=bestValue]
    left_node_labels = label[attributes[:,bestFeatureIndex]<=bestValue]
```

```
    right_node = attributes[attributes[:,bestFeatureIndex]>bestValue]
    right_node_labels = label[attributes[:,bestFeatureIndex]>bestValue]


    return left_node,left_node_labels, right_node,right_node_labels,␣
↪node_impurity,bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex
```

[64]:
```
# Here I made a tree implementation
```

[65]:
```
class Node():
    def __init__(self,parent,depth):

        self.parent = parent
        self.depth=depth
        self.childen = []

        self.sampleNumber = None
        self.labelNumbers = []
        self.labelNames= []
        self.bestSplitValue= None
        self.bestSplitIndex= None
        self.splitImpurity = None
        self.isLeaf = 0
        self.leafLabelName= None


def Tree(attributes,label,node,prun,impurity_choice):

    node_labels, labelNumbers = np.unique(label,return_counts= True)
    node.labelNames= node_labels
    node.labelNumbers=labelNumbers
    node.sampleNumber= np.sum(labelNumbers)

    if len(node_labels)==1: # it is pure
        node.isLeaf=1
        node.leafLabelName=node_labels[0]
        return


    else :
        left_node_Feat,left_node_labels, right_node_Feat,right_node_labels,␣
↪node_impurity,bestSplitImpurity,\
        bestSplitInfoGain,bestValue,bestFeatureIndex =␣
↪splitNode(attributes,label,impurity_choice)
        #Preprunning
```

```
        if bestSplitInfoGain <= prun:

            labelname,labelNum= np.unique(label, return_counts=True)
            node.leafLabelName= labelname[np.argmax(labelNum)]
            node.isLeaf = 1
            return
        # You can also add another pruning methods

        node.bestSplitValue=bestValue

        node.bestSplitIndex= bestFeatureIndex
        node.splitImpurity=bestSplitImpurity


        node.L_child= Node(node,node.depth+1)
        node.R_child= Node(node,node.depth+1)
        Tree(left_node_Feat,left_node_labels,node.L_child, prun,impurity_choice)
        Tree(right_node_Feat,right_node_labels,node.R_child,prun,impurity_choice)




def TraverseTree(node,data):

    if node.isLeaf==1:
        prediction = node.leafLabelName

    else:
        if data[node.bestSplitIndex] <= node.bestSplitValue:
            prediction = TraverseTree(node.L_child,data)
        else:
            prediction = TraverseTree(node.R_child,data)
    return prediction
```

[66]:
```
class DecisionTreeClassifier():

    def __init__(self,criterion,isPrunned="yes"):
        self.beginTree= None
        self.impurity_choice=criterion
        self.isPrunned=isPrunned


    def fit(self,data,label,prun):
```

```
        rootNode = Node(None,0)
        if self.isPrunned == "yes":
            Tree(data,label,rootNode,prun,self.impurity_choice)

        else:
            Tree(data,label,rootNode,0,self.impurity_choice)

        self.beginTree=rootNode

    def predict(self,data):

        if self.beginTree==None:
            print(" You need to create a tree !")

        else:
            prediction_list =[]
            for ii in range(0,data.shape[0]):
                prediction_list.append(TraverseTree(self.beginTree,data[ii,:]))

            prediction = np.asarray(prediction_list)
            return prediction
```

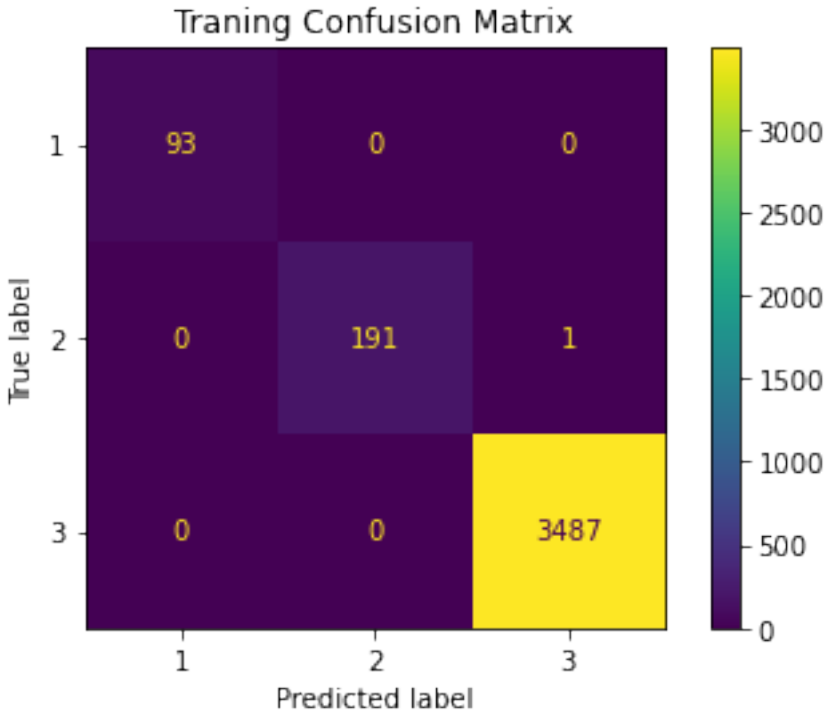
[67]:
```
def classificationAccuracy(model, data,label, mode = "Training"):

    preds= model.predict(data)
    pred_accuracy = np.sum(preds==label)/data.shape[0]
    print( mode, "  accuracy is : ", 100*pred_accuracy)




    #class based accuracy
    class_labels = np.unique(label,return_counts=False)
    class_accuracy = []

    for cl in class_labels:
        class_pred= model.predict(data[label==cl])
        class_acc= np.sum(class_pred==label[label==cl])/len(label[label==cl])
        print(mode, " accuracy for class ", cl, "is : ", class_acc*100 )
        class_accuracy.append(class_acc)



    # Confusion Matrix Creation
```

```
    cm= confusion_matrix(preds, label)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title(mode + " Confusion Matrix")
    plt.show()



# Here I will implement graph plotting
```

1 Confusion Matrix for Test and Train

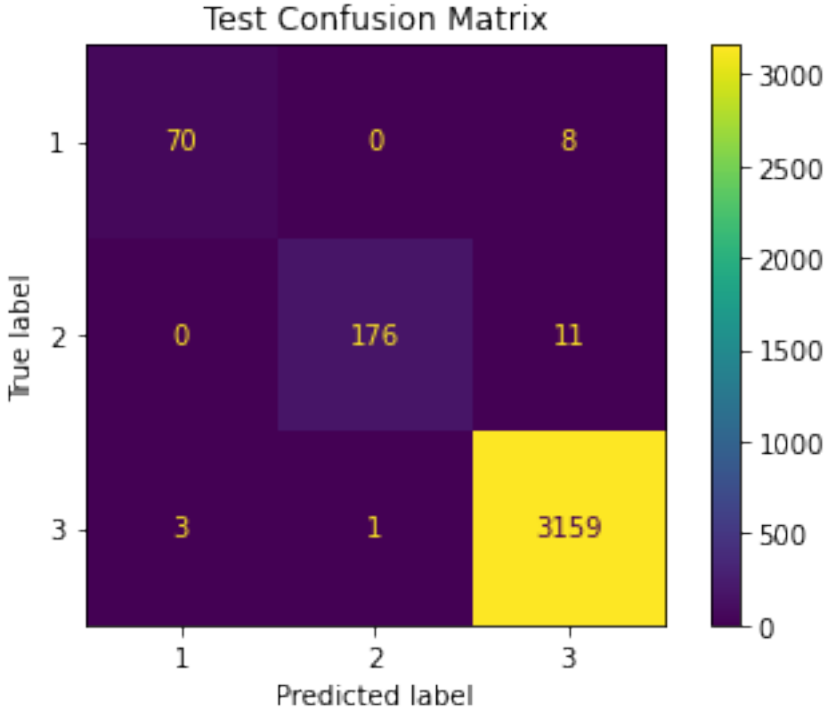
[68]:
```
decision_tree= DecisionTreeClassifier("entropy","yes")
decision_tree.fit(np_train_data,np_label_train,0.035)
classificationAccuracy(decision_tree,np_train_data,np_label_train,mode="Traning")
classificationAccuracy(decision_tree,np_test_data,np_label_test,mode="Test")
```

```
Traning   accuracy is :  99.97348886532343
Traning  accuracy for class  1 is :  100.0
Traning  accuracy for class  2 is :  100.0
Traning  accuracy for class  3 is :  99.97133027522935
```




```
Test   accuracy is :  99.32905484247374
```

```
Test  accuracy for class  1 is :  95.8904109589041
Test  accuracy for class  2 is :  99.43502824858757
Test  accuracy for class  3 is :  99.40213971050976
```



2 DONE

[ ]:

Question_3

March 14, 2022

1 Question 3

[375]:
```
import pandas as pd
import numpy as np
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score,ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.preprocessing import StandardScaler
from random import sample
import os
```

[376]:
```
cur_dir = os.getcwd()
train_file_path = os.path.join(cur_dir, 'ann-train.data')
test_file_path = os.path.join(cur_dir, 'ann-test.data')
feature_file_path = os.path.join(cur_dir, 'ann-thyroid.cost')
```

[377]:
```
# Organize features

features = pd.read_csv(feature_file_path, sep =":", header =None)
costs= list(features[features.columns[1]])
extraction= costs[18]+costs[19]
costs.append(extraction)
features = features[features.columns[:1]]
features = features.append(pd.Series("mixed"),ignore_index= True)
features_list = np.squeeze(features.to_numpy())
features = features.to_dict()[0]
```

[378]:
```
# Prepare tranining data
data_train = pd.read_csv(train_file_path, sep =" ", header =None)
train_data= data_train[data_train.columns[0:21]]
train_data = train_data.rename(columns=features,inplace =False) # This is pandas␣
 ↪version

np_train_data= train_data.to_numpy() # This is numpy version
```

```
label_train = data_train[data_train.columns[21]].astype("category")
np_label_train = label_train.to_numpy()
# Prepare test data

data_test = pd.read_csv(test_file_path, sep =" ", header =None)
test_data= data_test[data_test.columns[0:21]]
test_data = test_data.rename(columns=features,inplace =False)

np_test_data = test_data.to_numpy()
label_test=data_test[data_train.columns[21]].astype("category")
np_label_test = label_test.to_numpy()
```

[379]:
```
#Define some functions
def impurityCalculation(impurity_choice, labelsOfnode):

    labels, numLabels = np.unique(labelsOfnode, return_counts = True)
    total_labels= np.sum(numLabels)
    p_numLabels = numLabels/total_labels

    if impurity_choice == "entropy":
        entropy = 0
        total_entropy = np.sum([entropy+ (-np.log2(ii)*ii)  for ii in␣
 ↪p_numLabels])
        total_impurity = total_entropy


    elif impurity_choice == "gini":
        entropy = 0
        gini= (1/2)*(1-np.sum(np.square(p_numLabels)))
        total_impurity = gini

    else:
        raise Exception("Sorry", impurity_choice, "is not an impurity type. ")

    return total_impurity


def InfoGain(attribute, label, value,impurity_choice):
    total_entropy = impurityCalculation(impurity_choice,label)
    R_split = label[attribute>value]
    L_split = label[attribute<=value]
    L_prop= len(L_split)/len(label)
    R_prop= 1- L_prop

    L_impurity = impurityCalculation(impurity_choice,L_split)
    R_impurity = impurityCalculation(impurity_choice,R_split)
```

```
    split_impurity = L_prop*L_impurity + R_prop*R_impurity
    informationGain= total_entropy-split_impurity
    return informationGain

def LeftRightSplit(attribute, label, value,impurity_choice,feature_cost):


    total_entropy = impurityCalculation(impurity_choice,label)

    R_split = label[attribute>value]
    L_split = label[attribute<=value]
    L_prop= len(L_split)/len(label)
    R_prop= 1- L_prop

    L_impurity = impurityCalculation(impurity_choice,L_split)
    R_impurity = impurityCalculation(impurity_choice,R_split)

    informationGain=InfoGain(attribute,label,value,impurity_choice)
    split_impurity = L_prop*L_impurity + R_prop*R_impurity


    if feature_cost!=0:
        score= -(informationGain**2)/feature_cost
        return score
    elif feature_cost ==0:
        return split_impurity


def␣
 ↪cost_sensitive_exhaustive_search(attributes,label,impurity_choice,costVector):

    splitCheckAll= np.Inf
    gainAll=0
    valueSplitAll=0

    featureIndexSplit=0
    tempSplitCheck=0
    splitValue=0
    gainSplit=0


    for ft in range(0,attributes.shape[1]):

        feature_cost=costVector[ft]
        splitCheck= np.Inf

        for val in attributes[:,ft]:
```

```
            isSplit= LeftRightSplit(attributes[:
↪,ft],label,val,impurity_choice,feature_cost)
            gainCheck =InfoGain(attributes[:,ft], label, val,impurity_choice)
            if  isSplit < splitCheck:
                splitCheck=isSplit
                tempSplitCheck=splitCheck
                splitValue= val
                gainSplit= gainCheck


        if tempSplitCheck < splitCheckAll:
            splitCheckAll=tempSplitCheck
            featureIndexSplit=ft
            valueSplitAll= splitValue
            gainAll=gainSplit
    cost = costVector[featureIndexSplit]
    if featureIndexSplit==20:

        costVector[20]=0
        costVector[19]=0
        costVector[0]=0
    elif featureIndexSplit==19:

        costVector[20]=costVector[18]
        costVector[19]=0
    elif featureIndexSplit==18:
        costVector[20]=costVector[19]
        costVector[18]=0
    else:
        costVector[featureIndexSplit]=0
    costVectorUpdate= costVector

    print("Updated cost:" ,costVectorUpdate)

    return␣
↪splitCheckAll,gainAll,valueSplitAll,featureIndexSplit,costVectorUpdate,cost


def splitNode(attributes,label,impurity_choice,costVector):

    bestSplitScore,␣
↪bestSplitInfoGain,bestValue,bestFeatureIndex,costVectorUpdate,cost =␣
↪cost_sensitive_exhaustive_search(attributes,label,impurity_choice,costVector)

    node_impurity = impurityCalculation(impurity_choice,label)
```

```
    left_node = attributes[attributes[:,bestFeatureIndex]<=bestValue]
    left_node_labels = label[attributes[:,bestFeatureIndex]<=bestValue]


    right_node = attributes[attributes[:,bestFeatureIndex]>bestValue]
    right_node_labels = label[attributes[:,bestFeatureIndex]>bestValue]


    return left_node,left_node_labels, right_node,right_node_labels,␣
 ↪node_impurity,bestSplitScore,␣
 ↪bestSplitInfoGain,bestValue,bestFeatureIndex,costVectorUpdate,cost
```

[380]:
```
# Here I made a tree implementation
```

[381]:
```
class Node():
    def __init__(self,parent,depth):

        self.parent = parent
        self.depth=depth
        self.childen = []

        self.sampleNumber = None
        self.labelNumbers = []
        self.labelNames= []
        self.bestSplitValue= None
        self.bestSplitIndex= None
        self.splitImpurity = None
        self.isLeaf = 0
        self.leafLabelName= None
        self.cost=0


def Tree(attributes,label,node,prun,impurity_choice,costVector):

    node_labels, labelNumbers = np.unique(label,return_counts= True)
    node.labelNames= node_labels
    node.labelNumbers=labelNumbers
    node.sampleNumber= np.sum(labelNumbers)

    if len(node_labels)==1: # it is pure
        node.isLeaf=1
        node.leafLabelName=node_labels[0]
        return
```

```
    else :
        left_node_Feat,left_node_labels, right_node_Feat,right_node_labels,␣
↪node_impurity,bestSplitImpurity,\
        bestSplitInfoGain,bestValue,bestFeatureIndex,costVectorUpdate,cost =␣
↪splitNode(attributes,label,impurity_choice,costVector)
        #Preprunning
        if bestSplitInfoGain <= prun:

            labelname,labelNum= np.unique(label, return_counts=True)
            node.leafLabelName= labelname[np.argmax(labelNum)]
            node.isLeaf = 1
            return
        # You can also add another pruning methods

        node.bestSplitValue=bestValue

        node.bestSplitIndex= bestFeatureIndex
        node.splitImpurity=bestSplitImpurity
        node.cost=cost

        node.L_child= Node(node,node.depth+1)
        node.R_child= Node(node,node.depth+1)
        Tree(left_node_Feat,left_node_labels,node.L_child,␣
↪prun,impurity_choice,costVectorUpdate)
        Tree(right_node_Feat,right_node_labels,node.
↪R_child,prun,impurity_choice,costVectorUpdate)



def TraverseTree(node,data):

    if node.isLeaf==1:
        prediction = node.leafLabelName

    else:
        if data[node.bestSplitIndex] <= node.bestSplitValue:
            prediction = TraverseTree(node.L_child,data)
        else:
            prediction = TraverseTree(node.R_child,data)
    return prediction


def SumCost(node,sample,init_total_cost=0):

    if node.isLeaf==1:
        return node.cost
```

```
    else:
        if sample[node.bestSplitIndex] <= node.bestSplitValue:
            init_total_cost+=node.cost
            init_total_cost+= SumCost(node.L_child,sample)
            return init_total_cost
        else:
            init_total_cost+=node.cost
            init_total_cost+=SumCost(node.R_child,sample)
            return init_total_cost
```

[382]:
```
class DecisionTreeClassifier():

    def __init__(self,criterion,isPrunned="yes"):
        self.beginTree= None
        self.impurity_choice=criterion
        self.isPrunned=isPrunned


    def fit(self,data,label,prun,costVector):

        rootNode = Node(None,0)
        if self.isPrunned == "yes":
            Tree(data,label,rootNode,prun,self.impurity_choice,costVector)

        else:
            Tree(data,label,rootNode,0,self.impurity_choice,costVector)

        self.beginTree=rootNode

    def predict(self,data):

        if self.beginTree==None:
            print(" You need to create a tree !")

        else:
            prediction_list =[]
            for ii in range(0,data.shape[0]):
                prediction_list.append(TraverseTree(self.beginTree,data[ii,:]))

            prediction = np.asarray(prediction_list)
            return prediction
```

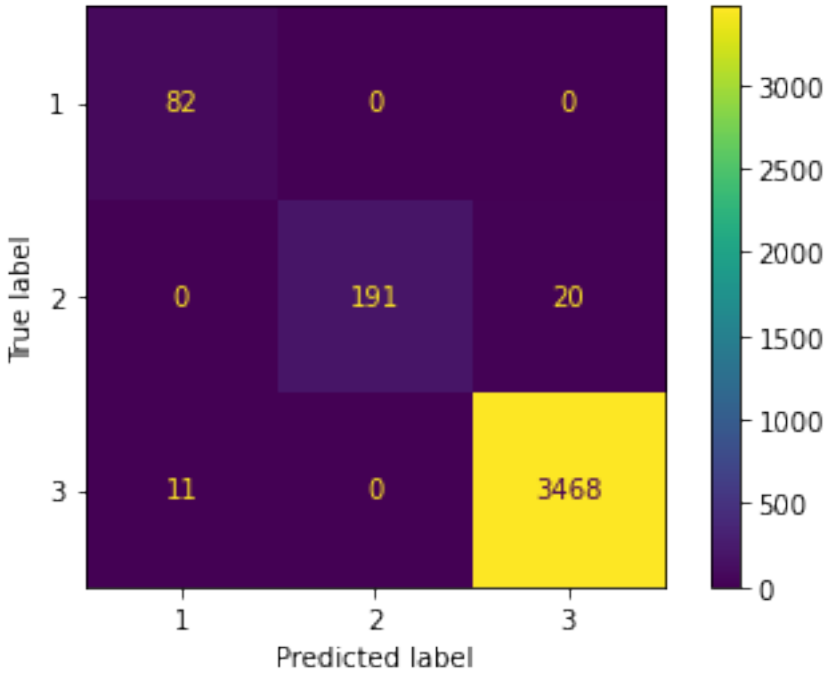
```
    def cost(self,data):
        init_cost=0

        for ii in range(0,data.shape[0]):
            init_cost+= SumCost(self.beginTree,data[ii,:])
        return init_cost
```

[383]:
```
def classificationAccuracy(model, data,label, mode = "Training"):

    preds= model.predict(data)
    pred_accuracy = np.sum(preds==label)/data.shape[0]
    print( mode, "  accuracy is : ", 100*pred_accuracy)




    #class based accuracy
    class_labels = np.unique(label,return_counts=False)
    class_accuracy = []


    for cl in class_labels:
        class_pred= model.predict(data[label==cl])
        class_acc= np.sum(class_pred==label[label==cl])/len(label[label==cl])
        class_avg_cost = model.cost(data[label==cl])/len(label[label==cl])
        print("Average", mode, " cost for class", cl, "is",class_avg_cost)
        print(mode, " accuracy for class ", cl, "is : ", class_acc*100 )
        class_accuracy.append(class_acc)




    # Confusion Matrix Creation

    cm= confusion_matrix(preds, label)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.show()


def Decision_Tree_Plot(model,attribute_list,class_names):

    decision_tree= model.beginTree

    tree=Digraph(name)
```

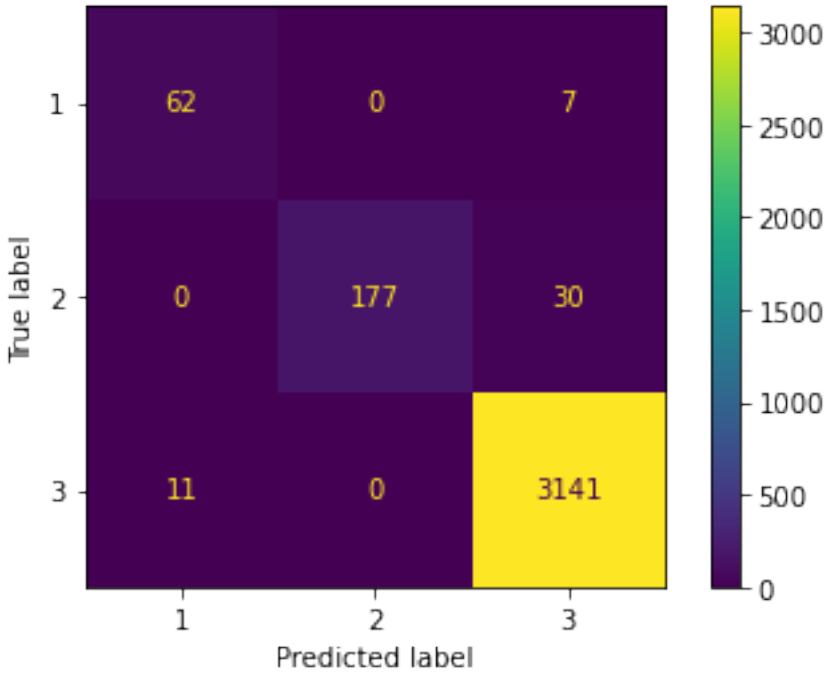
2 Confusion Matrix for Test and Train with Average Costs

[384]:
```
decision_tree= DecisionTreeClassifier("entropy","yes")
decision_tree.fit(np_train_data,np_label_train,0.025,costs)
classificationAccuracy(decision_tree,np_train_data,np_label_train,mode="Train")
classificationAccuracy(decision_tree,np_test_data,np_label_test,mode="Test")
```

```
Updated cost: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 0, 11.41, 14.51, 11.41, 25.92]
Updated cost: [1.0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 0, 11.41, 14.51, 11.41, 25.92]
Updated cost: [0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 0, 11.41, 14.51, 0, 0]
Updated cost: [0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 0, 11.41, 14.51, 0, 0]
Updated cost: [0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 0, 0, 14.51, 0, 0]
Updated cost: [0, 0, 0, 1.0, 1.0, 1.0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 0, 0, 14.51, 0, 0]
Updated cost: [0, 0, 0, 1.0, 1.0, 1.0, 1.0, 0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 0, 0, 14.51, 0, 0]
Updated cost: [0, 0, 0, 1.0, 1.0, 1.0, 1.0, 0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 0, 0, 0, 0, 0]
Updated cost: [0, 0, 0, 0, 1.0, 1.0, 1.0, 0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 0, 0, 0, 0, 0]
Updated cost: [0, 0, 0, 0, 1.0, 0, 1.0, 0, 1.0, 0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
0, 0, 0, 0, 0]
Updated cost: [0, 0, 0, 0, 1.0, 0, 1.0, 0, 0, 0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
0, 0, 0, 0, 0]
Train   accuracy is :  99.17815482502651
Average Train  cost for class 1 is 58.35430107526878
Train  accuracy for class  1 is :  88.17204301075269
Average Train  cost for class 2 is 49.699999999999996
Train  accuracy for class  2 is :  100.0
Average Train  cost for class 3 is 23.00142488532023
Train  accuracy for class  3 is :  99.42660550458714
```




```
Test   accuracy is :  98.59976662777129
Average Test  cost for class 1 is 57.69273972602743
Test  accuracy for class  1 is :  84.93150684931507
Average Test  cost for class 2 is 49.69999999999994
Test  accuracy for class  2 is :  100.0
Average Test  cost for class 3 is 23.150305223410108
Test  accuracy for class  3 is :  98.83574575204531
```


62
0
7
0
177
30
11
0
3141
1
2
3
True label
Predicted label
3000
2500
2000
1500
1000
500
0

[ ]:

[ ]:

3 Done

[ ]: