In [1]:
```
#Entropy(S) = - ∑ pᵢ * log₂(pᵢ) ; i = 1 to n
#IG(S, A) = Entropy(S) - ∑((|Sᵥ| / |S|) * Entropy(Sᵥ))
# !pwd
# !ls ../Data
```

Importing Necessary Libraries

In [2]:
```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, roc_curve, confusion_matrix,auc
from sklearn.metrics import RocCurveDisplay
```

Performing data preprocessing to binarize the features present in the data.

In [3]:
```
fileInpTrain = "../Data/preProcess/car-data-train.csv"
fileInpTest = "../Data/preProcess/car-data-test.csv"


df = pd.read_csv(fileInpTrain, header=None)

df.iloc[:, 0] = df.iloc[:, 0].apply(lambda val: "low"
                                    if val == "med"
                                    else "high" if val == "vhigh"
                                    else val)

df.iloc[:, 1] = df.iloc[:, 1].apply(lambda val: "low"
                                    if val == "med"
                                    else "high" if val == "vhigh"
                                    else val)

# Created >2 other doors into one bin and leave 2 doors as 1 bin #uneven feature binning
# Check for other binning strategy
df.iloc[:, 2] = df.iloc[:, 2].apply(lambda val: val
                                    if val == '2'
                                    else '>2')


# Integer Division
FirstHalfIdx = df.iloc[:, 3][df.iloc[:, 3] == "4"].index[:len(df.iloc[:, 3][df.iloc[:, 3] == "4"].index)//2 + 1]
SecondHalfIdx = df.iloc[:, 3][df.iloc[:, 3] == "4"].index[len(df.iloc[:, 3][df.iloc[:, 3] == "4"].index)//2 + 1:]

# Number of person. Half of 4 in 2 and other half in more
df.iloc[:, 3] = df.apply(lambda row: "2" if row.name in FirstHalfIdx
                                         else "more"
                                         if row.name in SecondHalfIdx
                                         else row[3], axis=1)


# Integer Division
FirstHalfIdx = df.iloc[:, 4][df.iloc[:, 4] == "med"].index[:len(df.iloc[:, 4][df.iloc[:, 4] == "med"].index)//2 + 1
SecondHalfIdx = df.iloc[:, 4][df.iloc[:, 4] == "med"].index[len(df.iloc[:, 4][df.iloc[:, 4] == "med"].index)//2 + 1

#Lug Boot. Half of med added to small and other half med in more
df.iloc[:, 4] = df.apply(lambda row: "small" if row.name in FirstHalfIdx
                                             else "big"
                                             if row.name in SecondHalfIdx
                                             else row[4], axis=1)

# Integer Division
FirstHalfIdx = df.iloc[:, 5][df.iloc[:, 5] == "med"].index[:len(df.iloc[:, 5][df.iloc[:, 5] == "med"].index)//2 + 1
SecondHalfIdx = df.iloc[:, 5][df.iloc[:, 5] == "med"].index[len(df.iloc[:, 5][df.iloc[:, 5] == "med"].index)//2 + 1

# Number of person. Half of 4 in 2 and other half in more
df.iloc[:, 5] = df.apply(lambda row: "low" if row.name in FirstHalfIdx
                                             else "high"
                                             if row.name in SecondHalfIdx
                                             else row[5], axis=1)


# Count the binnings
value_counts = [df.iloc[:, i].value_counts() for i in range(len(df.columns))]

df.columns = ["buying", "maint", "doors", "persons", "lug_boot", "safety", "label"]

fileOutTrain = "../Data/postProcess/train.csv"
fileOutTest = "../Data/postProcess/test.csv"

df.to_csv(fileOutTrain, index=None)
```

Creating the Decision tree class and a complimentary node class.

We have shown how the two algorithms for making a decision tree work such as cart and Id3 which are the fundamental algorithms used for designing a decision tree.

In [4]:

```
'''
Node class acts as a supplimentary class
which creates the most basic element for
building a tree.

'''
class Node:
    def __init__(self, data, maxdepth):
        self.data = data
        self.maxdepth = maxdepth
        self.key = None #splitting attribute
        self.infogain = None
        self.label = None #classifier
        self.left = None
        self.right = None
        self.parent = None
        self.value = None #attribute value (ex. 'y' or 'n')

'''
Gini Index is a concise way to measure income disparity.
Incorporating the detailed share data into a single statistic,
the Gini coefficient captures the income distribution over the
full income distribution.

'''

def gini_index(labels):
    totalNumofLabels = len(labels)
    classes, countOfClasses = np.unique(labels, return_counts=True)
    if len(classes) <= 1:
        return 0
    ## computation
    probs = countOfClasses/totalNumofLabels
    gini_index = 0
    for p in probs:
        #calculation used to determine the gini index of the labels
        gini_index += p**2
    return 1 - gini_index

'''
Entropy is a metric used in information theory to
gauge how pure or uncertain a set of observations is.
It controls the way a decision tree decides how to
divide data.
'''

def entropy(labels):
    totalNumofLabels = len(labels)
    classes, countOfClasses = np.unique(labels, return_counts=True)
    #if arity of the labels is 1 then the entropy is 1
    if len(classes) <= 1:
        return 0
    ##computation
    probs = countOfClasses/totalNumofLabels
    ent = 0
    for p in probs:
        #calculation used to determine the entropy of the labels
        ent += p*math.log2(p)
    return -1 * ent

'''
These are helper functions. They are used to find mutual information value.

'''

def mutualInfo(data, position, switch):
    if switch == "entropy":
        ent = entropy(data[:,-1])
        lengthOfValues = len(data[:, position])
        classes, countOfClasses = np.unique(data[:, position], return_counts=True)
        if len(classes) <= 1:
            return 0
        #calculate probabilities used in conditional entropy
        probs = countOfClasses/lengthOfValues
        #calculate specific conditional entropies
        specCondEnts = []
        for c in classes:
            specCondEnt = 0
            boolean = data[:, position] == c
            subset = data[boolean]
            lengthOfSubset = len(subset)
            subsetClasses, subsetCountofClasses = np.unique(subset[:,-1], return_counts=True)
            subsetClasses, subsetCountofClasses
            probsOfSubset = subsetCountofClasses/lengthOfSubset
            for p in probsOfSubset:
                if p != 0:
```

```
                    specCondEnt += p*math.log2(p)
                else:
                    specCondEnt += 0
            specCondEnts.append(-1*specCondEnt)
        #calculate conditional entropy
        condEnt = sum(p*specCondEnts[idx] for idx, p in enumerate(probs))
        #calculate info gain
        infoGain = ent - condEnt
    elif switch == "gini" :
        ent = gini_index(data[:, -1])
        lengthOfValues = len(data[:, position])
        classes, countOfClasses = np.unique(data[:, position], return_counts=True)
        if len(classes) <= 1:
            return 0
        #calculate probabilities used in conditional entropy
        probs = countOfClasses/lengthOfValues
        #calculate specific conditional entropies
        specCondEnts = []
        for c in classes:
            specCondEnt = 0
            boolean = data[:, position] == c
            subset = data[boolean]
            lengthOfSubset = len(subset)
            subsetClasses, subsetCountofClasses = np.unique(subset[:,-1], return_counts=True)
            subsetClasses, subsetCountofClasses
            probsOfSubset = subsetCountofClasses/lengthOfSubset
            for p in probsOfSubset:
                if p != 0:
                    specCondEnt += p**2
                else:
                    specCondEnt += 0
            specCondEnts.append(1 - specCondEnt)
        #calculate conditional entropy
        condEnt = sum(p*specCondEnts[idx] for idx, p in enumerate(probs))
        #calculate info gain
        infoGain = ent - condEnt
    else:
        raise Exception("No other criterion. Only entropy or gini")

    return infoGain

'''
These are helper functions. They are used to split the
data value at each Node.

'''

def splitData(data, idx):
    classes = np.unique(data[:,idx])
    subsets = []
    for c in classes:
        boolean = data[:, idx] == c
        subset = data[boolean]
        subsets.append(subset)

    return subsets

'''
These are helper functions. Function returns attribute that
obtains the highest mutual information at each node.

'''

def bestMutualInfo(data, attributes, switch): #optimize this to work better
    if len(attributes) == 1: #unnecessary
        return attributes[0]
    elif len(attributes) == 0:
        return None
    else:
        atts = attributes.copy()[:len(attributes)-1]
        bestInfoGain = -np.inf
        bestatt = None
        for idx, att in enumerate(atts):
            if mutualInfo(data, idx, switch) >= bestInfoGain:
                bestInfoGain = mutualInfo(data, idx, switch=switch)
                bestatt = atts[idx]

        return bestInfoGain, bestatt

'''
These are helper functions. Function need to pass last column
which is the target variable calculates the majority class
for the labels.

'''
```

```
def maj_classifer(data):
    labels = data.copy()
    classes, countOfClasses = np.unique(labels, return_counts=True)
    if len(classes) == 1:
        return classes[0]
    counter = 0
    maxValue = -np.inf
    best_label = None
    while counter < len(classes):
        for idx, count in enumerate(countOfClasses):
            if count >= maxValue:
                maxValue = count
                best_label = classes[idx]
            counter += 1
    return best_label

'''
These are helper functions. Recursive Algorithm to build the tree

'''

def buildTree(traindata,feats, maxdepth, switch):
    if maxdepth == 0:
        return maj_classifer(traindata[:,-1])
    #maxdepth is limited by number of features
    if maxdepth > len(feats) + 1:
        maxdepth = len(feats) + 1
    #create the root node with all the training data and an initial max depth
    root = Node(traindata, maxdepth)
    #calculate the information gain at that node and the attribute that best splits the data at that node
    infoGainVal, bestAtt = bestMutualInfo(root.data, feats, switch=switch)
    #base case
    if infoGainVal <= 0:
        root.key = 'leaf'
        root.label = maj_classifer(root.data[:, -1])
        return root
    root.key = bestAtt
    root.label = maj_classifer(root.data[:,-1])
    root.infogain = infoGainVal
    #split the root node data
    rightData, leftData = splitData(root.data, feats.index(bestAtt))

    #recurse to the left subtree
    if maxdepth != 1 and root.left == None :
        root.left = buildTree(leftData, feats, maxdepth - 1, switch=switch)
        root.left.value = leftData[0,feats.index(bestAtt)]
        root.left.parent = bestMutualInfo(root.data, feats, switch=switch)[1]

    #recurse to the right subtree
    if maxdepth != 1 and root.right == None :
        root.right = buildTree(rightData, feats, maxdepth - 1, switch=switch)
        root.right.value = rightData[0,feats.index(bestAtt)]
        root.right.parent = bestMutualInfo(root.data, feats, switch=switch)[1]

    return root

'''
These are helper functions. Recursive Algorithm to print the tree

'''

def printPreorder(root, classOne, classTwo, counter = 0):
    if root:
        classes = [classOne, classTwo]
        _, countOfClasses = np.unique(root.data[:,-1], return_counts=True)
        if counter == 0:
            print('[{} {} /{} {}]\n'.format(countOfClasses[0], classes[0], countOfClasses[1], classes[1]))
        else:
            if len(countOfClasses) == 2:
                print('|'* counter + '{} = {}: [{} {} /{} {}] \n'.format(root.parent, root.value, countOfClasses[0
            elif len(countOfClasses) == 1 and _ == classes[0]:
                print('|' * counter +'{} = {}: [{} {} /{} {}] \n'.format(root.parent, root.value, countOfClasses[0
            else:
                print('|' * counter + '{} = {}: [{} {} /{} {}] \n'.format(root.parent, root.value, 0, classes[0], c

        # Then recur on left child
        printPreorder(root.left, classes[0], classes[1], counter+1)
        #Finally recur on right child
        printPreorder(root.right, classes[0], classes[1], counter+1)

'''
These are helper functions. Recursive Algorithm to print the tree

'''
def printPreorder2(tree=None, indent=" "):
    ''' function to print the tree '''
```

```
    if tree:
        print("X_"+str(tree.key), " ? ", str(tree.infogain))
        print("%sleft:" % (indent), end="")
        printPreorder2(tree.left, indent + indent)
        print("%sright:" % (indent), end="")
        printPreorder2(tree.right, indent + indent)


'''
These are helper functions. Recursive function that traverses the tree
and return the prediction of the query

'''

def prediction(tree, feats, row, maxdepth, currentdepth=1):
    #base case
    if tree.key == 'leaf':
        return tree.label
    #base case
    if maxdepth == currentdepth:
        return tree.label
    #recurse
    if any(tree.key == feat for feat in feats):
        idx = feats.index(tree.key)
        if row[idx] == tree.left.value:
            left = prediction(tree.left,feats,row, maxdepth, currentdepth + 1)
            return left
        if row[idx] == tree.right.value:
            right = prediction(tree.right,feats,row, maxdepth, currentdepth + 1)
            return right
```

Using the ID3 Algorithm

In [5]:

```
# Setting up the train and test data.
train = pd.read_csv("../Data/postProcess/train.csv")
test = pd.read_csv("../Data/postProcess/test.csv")

train = train.to_numpy()
test = test.to_numpy()
XTrain_Id3, yTrain_Id3 = train[: , :train.shape[1] - 1], train[:, -1]
XTest_Id3, yTest_Id3 = test[: , :test.shape[1] - 1], test[:, -1]

features = ["buying", "maint", "doors", "persons", "lug_boot", "safety", "label"]
maxDepth = 4

root = buildTree(train, features, maxDepth, switch="entropy")

#printout of trained tree
classes = np.unique(root.data[:,-1])
printPreorder(root, classes[0], classes[1])
# print()
# print(printPreorder2(root))

#Train on train and predict on Train
yTrainPred_Id3 = []
for row in XTrain_Id3:
    yTrainPred_Id3.append(prediction(root,  features, row, maxDepth))

#Train on train and predict on Test
yTestPred_Id3 = []
for row in XTest_Id3:
    yTestPred_Id3.append(prediction(root,  features, row, maxDepth))

yTrainPred_Id3 = np.array(yTrainPred_Id3)
yTestPred_Id3 = np.array(yTestPred_Id3)
```

```
[386 acc /910 unacc]

|safety = low: [83 acc /571 unacc]

||buying = low: [59 acc /272 unacc]

|||doors = >2: [48 acc /200 unacc]

|||doors = 2: [11 acc /72 unacc]

||buying = high: [24 acc /299 unacc]

|||maint = low: [20 acc /140 unacc]

|||maint = high: [4 acc /159 unacc]

|safety = high: [303 acc /339 unacc]

||persons = more: [239 acc /111 unacc]

|||buying = low: [160 acc /20 unacc]

|||buying = high: [79 acc /91 unacc]

||persons = 2: [64 acc /228 unacc]

|||lug_boot = small: [45 acc /103 unacc]

|||lug_boot = big: [19 acc /125 unacc]
```

Accuracy

Calculating the accuracy for Train Data on ID3 Algorithm

In [6]:

```
sum(yTrainPred_Id3 == yTrain_Id3)/len(yTrain_Id3)
```

Out[6]:

```
0.8101851851851852
```

Calculating the accuracy for Test Data on ID3

In [7]:
```
#Acc for Test Data on ID3
sum(yTestPred_Id3 == yTest_Id3)/len(yTest_Id3)
```

Out[7]:
```
0.8078703703703703
```

Confusion Matrix

A Confusion Matrix is a table called a confusion matrix is used to describe how well a classification system performs. the output of a classification algorithm is shown and summarized in a confusion matrix.

Confusion Matrix for ID3 Train data

In [8]:
```
confusionMatrix_Id3_Train = confusion_matrix(yTrain_Id3,yTrainPred_Id3)
confusionMatrix_Id3_Train
```

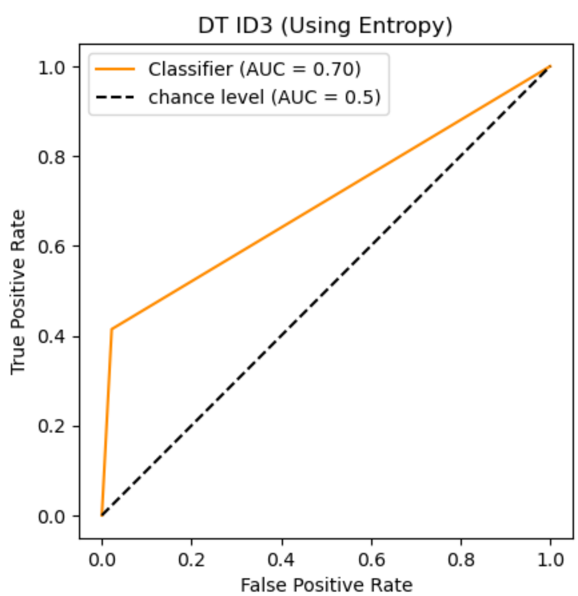
Out[8]:
```
array([[160, 226],
       [ 20, 890]])
```

ROC Curve for ID3 Train

In [9]:
```
binaryTransformation = lambda val: 0 if val == 'unacc' else 1
fn = np.vectorize(binaryTransformation)
yTrain_Id3 = fn(yTrain_Id3)
yTrainPred_Id3 = fn(yTrainPred_Id3)
```

In [10]:
```
rocCurve_ID3 = roc_curve(yTrain_Id3, yTrainPred_Id3, pos_label=1)
fpr, tpr, threshold = rocCurve_ID3
```

In [11]:
```
RocCurveDisplay.from_predictions(
    yTrain_Id3,
    yTrainPred_Id3,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT ID3 (Using Entropy)")
plt.legend()
plt.show()
```

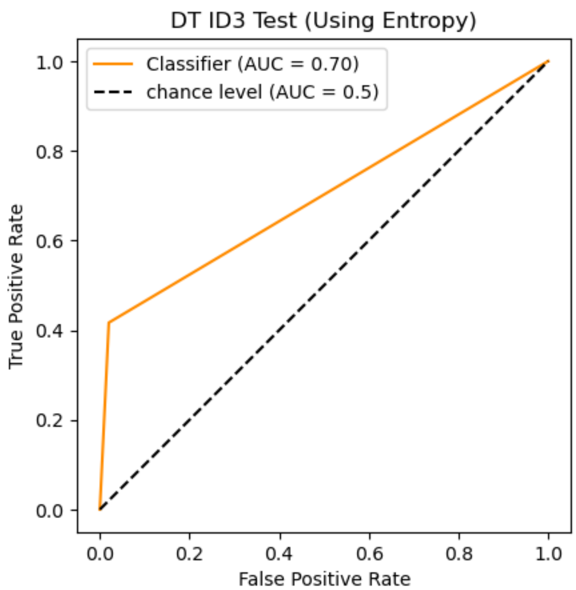


ROC Curve for ID3 Test

In [12]:
```
binaryTransformation = lambda val: 0 if val == 'unacc' else 1
fn = np.vectorize(binaryTransformation)
yTest_Id3 = fn(yTest_Id3)
yTestPred_Id3 = fn(yTestPred_Id3)
```

In [13]:
```
rocCurve_ID3Test = roc_curve(yTest_Id3, yTestPred_Id3, pos_label=1)
fpr, tpr, threshold = rocCurve_ID3Test
```

In [14]:
```
RocCurveDisplay.from_predictions(
    yTest_Id3,
    yTestPred_Id3,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT ID3 Test (Using Entropy)")
plt.legend()
plt.show()
```



Cart Algorithm on Car Evaluation

In [15]:
```
XTrain_Cart, yTrain_Cart = train[: , :train.shape[1] - 1], train[:, -1]
XTest_Cart, yTest_Cart = test[: , :test.shape[1] - 1], test[:, -1]

features = ["buying", "maint", "doors", "persons", "lug_boot", "safety", "label"]
maxDepth = 4

root = buildTree(train, features, maxDepth, switch="gini")

#printout of trained tree
classes = np.unique(root.data[:,-1])
printPreorder(root, classes[0], classes[1])

# printPreorder2(root)

#Train on train and predict on Train
yTrainPred_Cart = []
for row in XTrain_Cart:
    yTrainPred_Cart.append(prediction(root,  features, row, maxDepth))

#Train on train and predict on Test
yTestPred_Cart = []
for row in XTest_Cart:
    yTestPred_Cart.append(prediction(root,  features, row, maxDepth))

yTrainPred_Cart = np.array(yTrainPred_Cart)
yTestPred_Cart = np.array(yTestPred_Cart)
```

```
[386 acc /910 unacc]

|safety = low: [83 acc /571 unacc]

||buying = low: [59 acc /272 unacc]

|||doors = >2: [48 acc /200 unacc]

|||doors = 2: [11 acc /72 unacc]

||buying = high: [24 acc /299 unacc]

|||lug_boot = small: [5 acc /173 unacc]

|||lug_boot = big: [19 acc /126 unacc]

|safety = high: [303 acc /339 unacc]

||persons = more: [239 acc /111 unacc]

|||buying = low: [160 acc /20 unacc]

|||buying = high: [79 acc /91 unacc]

||persons = 2: [64 acc /228 unacc]

|||lug_boot = small: [45 acc /103 unacc]

|||lug_boot = big: [19 acc /125 unacc]
```

Calculating the accuracy for Train Data on CART Algorithm

In [16]:
```
#Acc for Train Data on CART
sum(yTrainPred_Cart == yTrain_Cart)/len(yTrain_Cart)
```

Out[16]: 0.8101851851851852

Calculating the accuracy for Test Data on CART Algorithm

In [17]:
```
#Acc for Test Data on CART
sum(yTestPred_Cart == yTest_Cart)/len(yTest_Cart)
```

Out[17]: 0.8078703703703703

In [18]:
```
df.head()
```

Out[18]:

	buying	maint	doors	persons	lug_boot	safety	label
0	high	low	2	more	small	high	unacc
1	low	low	>2	2	small	high	acc
2	low	high	>2	more	small	low	unacc
3	high	low	>2	2	big	high	acc
4	low	low	>2	more	small	high	acc

Confusion Matrix

A Confusion Matrix is a table called a confusion matrix is used to describe how well a classification system performs. the output of a classification algorithm is shown and summarized in a confusion matrix.

In [19]:
```
confusionMatrix_Cart_Train = confusion_matrix(yTrain_Cart,yTrainPred_Cart)
confusionMatrix_Cart_Train
```

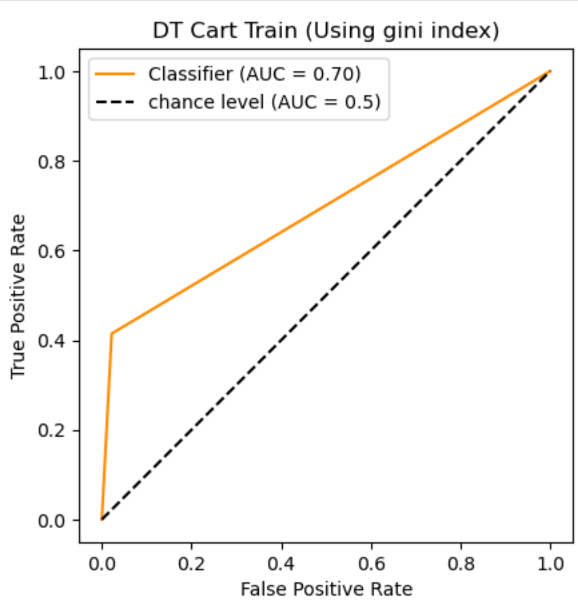
Out[19]:
```
array([[160, 226],
       [ 20, 890]])
```

ROC Curve for CART Train

In [20]:
```
binaryTransformation = lambda val: 0 if val == 'unacc' else 1
fn = np.vectorize(binaryTransformation)
yTrain_Cart = fn(yTrain_Cart)
yTrainPred_Cart = fn(yTrainPred_Cart)
```

In [21]:
```
rocCurve_Cart = roc_curve(yTrain_Cart, yTrainPred_Cart, pos_label=1)
fpr, tpr, threshold = rocCurve_Cart
```

In [22]:
```
RocCurveDisplay.from_predictions(
    yTrain_Cart,
    yTrainPred_Cart,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT Cart Train (Using gini index)")
plt.legend()
plt.show()
```

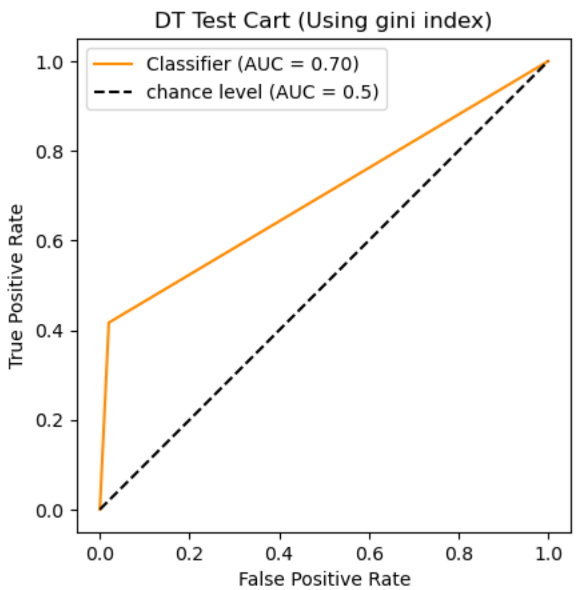


ROC for CART Test

In [23]:
```
binaryTransformation = lambda val: 0 if val == 'unacc' else 1
fn = np.vectorize(binaryTransformation)
yTest_Cart = fn(yTest_Cart)
yTestPred_Cart = fn(yTestPred_Cart)
```

In [24]:
```
rocCurve_CartTest = roc_curve(yTest_Id3, yTestPred_Id3, pos_label=1)
fpr, tpr, threshold = rocCurve_CartTest
```

In [25]:
```
RocCurveDisplay.from_predictions(
    yTest_Cart,
    yTestPred_Cart,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT Test Cart (Using gini index)")
plt.legend()
plt.show()
```



Helper function for converting the values into binary

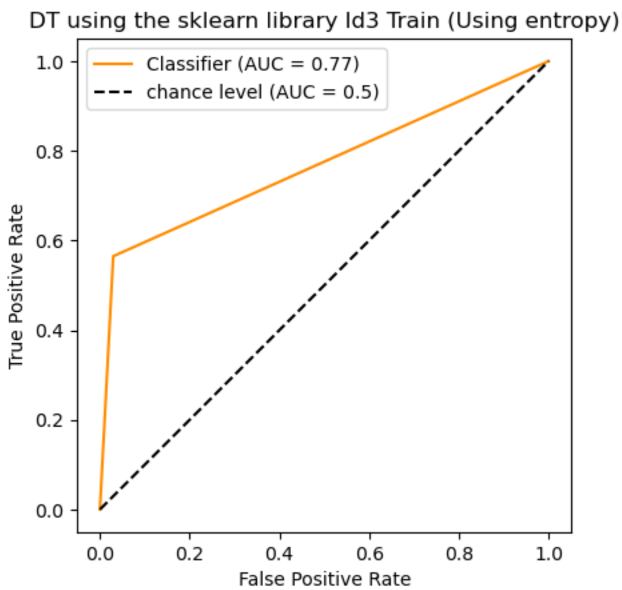
In [26]:
```
def conversion(XTrain_):
    for i in range(len(XTrain_)):
        for j in range(len(XTrain_[i])):
            if XTrain_[i][j] == 'high':
                XTrain_[i][j] = 1
            elif XTrain_[i][j] == 'low':
                XTrain_[i][j] = 0
            elif XTrain_[i][j] == '2':
                XTrain_[i][j] = 1
            elif XTrain_[i][j] == '>2':
                XTrain_[i][j] = 0
            elif XTrain_[i][j] == 'more':
                XTrain_[i][j] = 0
            elif XTrain_[i][j] == 'small':
                XTrain_[i][j] = 0
            elif XTrain_[i][j] == 'big':
                XTrain_[i][j] = 1
            elif XTrain_[i][j] == 'high':
                XTrain_[i][j] = 1
            elif XTrain_[i][j] == 'low':
                XTrain_[i][j] = 0
    return XTrain_
```

Decision tree using library of sklearn

Train ID3 data

In [27]:
```
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier(criterion='entropy', splitter='best', max_depth=None, min_samples_split=2)
xt = conversion(XTrain_Id3)
dt.fit(xt,yTrain_Id3)
xtt = conversion(XTest_Id3)
yTrainPred_Id3= dt.predict(XTrain_Id3)
```

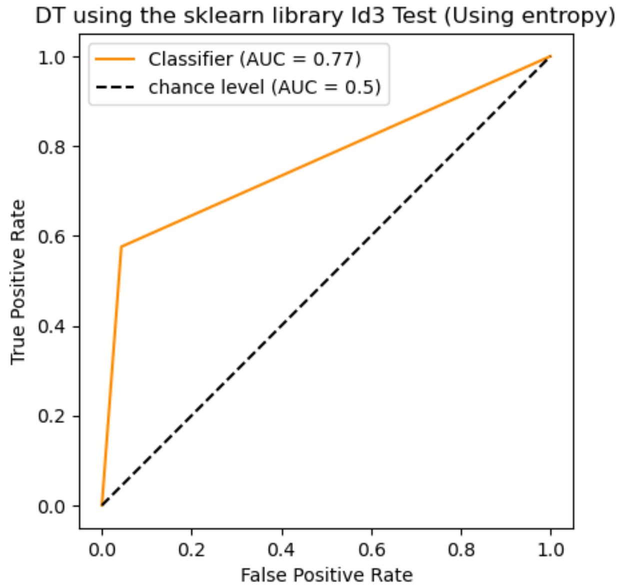
In [28]:
```
RocCurveDisplay.from_predictions(
#     yTest_Id3,y_pred,
    yTrain_Id3,yTrainPred_Id3,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT using the sklearn library Id3 Train (Using entropy)")
plt.legend()
plt.show()
```



Test ID3 data

In [29]:
```
dt = DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None, min_samples_split=2)
xt1 = conversion(XTest_Id3)
dt.fit(xt1,yTest_Id3)
# xtt = conversion(XTest_Id3)
yTestPred_Id3 = dt.predict(XTest_Id3)
```

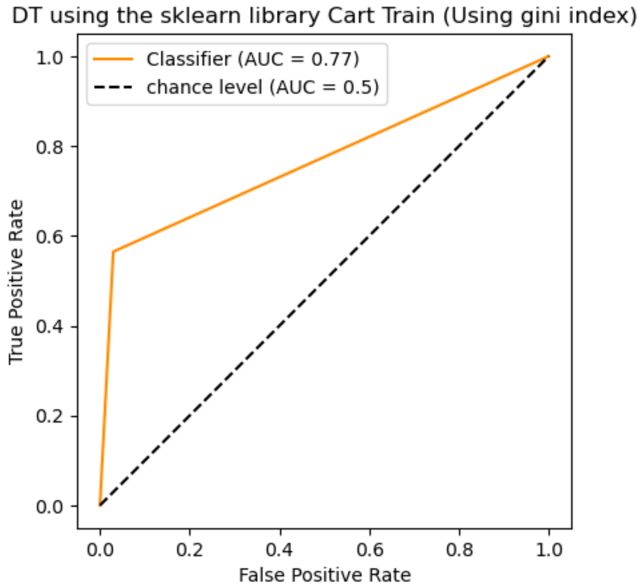
In [30]:
```
RocCurveDisplay.from_predictions(
    yTest_Id3,yTestPred_Id3,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT using the sklearn library Id3 Test (Using entropy)")
plt.legend()
plt.show()
```



Train data for Cart

In [31]:
```
dt = DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None, min_samples_split=2)
# xt2 = conversion(XTr_Cart)
dt.fit(XTrain_Cart,yTrain_Cart)
yTrainPred_Cart = dt.predict(XTrain_Cart)
```

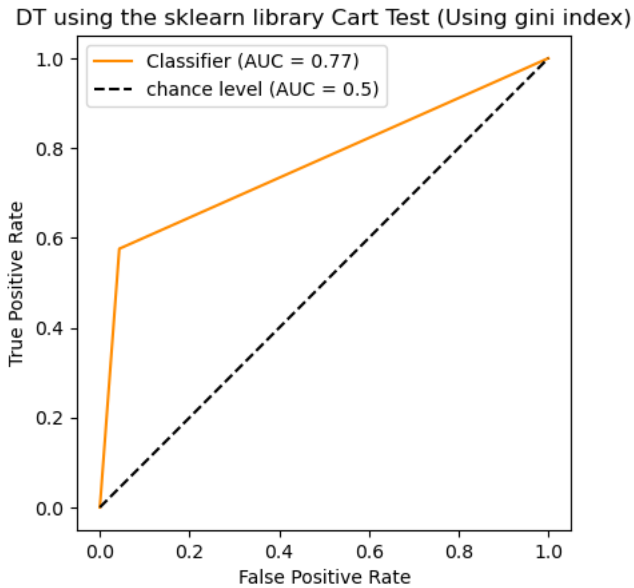
In [32]:
```
RocCurveDisplay.from_predictions(
    yTrain_Cart,yTrainPred_Cart,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT using the sklearn library Cart Train (Using gini index)")
plt.legend()
plt.show()
```



Test data for Cart

In [33]:
```
dt = DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None, min_samples_split=2)
# xt2 = conversion(XTr_Cart)
dt.fit(XTest_Cart,yTest_Cart)
yTestPred_Cart = dt.predict(XTest_Cart)
```

In [34]:
```
RocCurveDisplay.from_predictions(
    yTest_Cart,yTestPred_Cart,
    color="darkorange"
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("DT using the sklearn library Cart Test (Using gini index)")
plt.legend()
plt.show()
```



In [35]:
```
value_counts
```

Out[35]:
```
[low      660
 high     636
 Name: 0, dtype: int64,
 low      648
 high     648
 Name: 1, dtype: int64,
 >2     963
 2      333
 Name: 2, dtype: int64,
 2        655
 more     641
 Name: 3, dtype: int64,
 big      657
 small    639
 Name: 4, dtype: int64,
 low      654
 high     642
 Name: 5, dtype: int64,
 unacc    910
 acc      386
 Name: 6, dtype: int64]
```