Experiment 06 - Classification

| Roll No. |  |
| --- | --- |
| Name |  |
| Class | D15A |
| Subject | DS using Python Lab |
| LO Mapped | LO2: Understand the concept of Data science process and associated terminologies to solve real-world problems .  LO3: Analyze and apply the supervised machine learning techniques like Classification, Regression or Support Vector Machine on data for building the models of data and solve the problems. |
|  |  |

**Aim**:

To implement the classification algorithm using Python.

**Classification Algorithm**:

Classification algorithms are used to categorize data into a class or category. It can be performed on both structured or unstructured data. Classification can be of three types: binary classification, multiclass classification, multilabel classification.

Some of the classification algorithms are given below

**1. Naive Bayes**

Naive Bayes is based on Bayes’s theorem which gives an assumption of independence among predictors. This classifier assumes that the presence of a particular feature in a class is not related to the presence of any other feature/variable.

Naive Bayes Classifiers are of three types: Multinomial Naive Bayes, Bernoulli Naive Bayes, Gaussian Naive Bayes.

Pros:

* This algorithm works very fast.
* It can also be used to solve multi-class prediction problems as it’s quite useful with them.
* This classifier performs better than other models with less training data if the assumption of independence of features holds.

Cons:

* It assumes that all the features are independent. While it might sound great in theory, but in real life, anyone can hardly find a set of independent features.

Example:

| from sklearn.datasets import load\_iris from sklearn.model\_selection import train\_test\_split from sklearn.naive\_bayes import GaussianNB X, y = load\_iris(return\_X\_y=True) X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=142) Naive\_Bayes = GaussianNB() Naive\_Bayes.fit(X\_train, y\_train) prediction\_results = Naive\_Bayes.predict(X\_test)  print(prediction\_results) |
| --- |

Output:

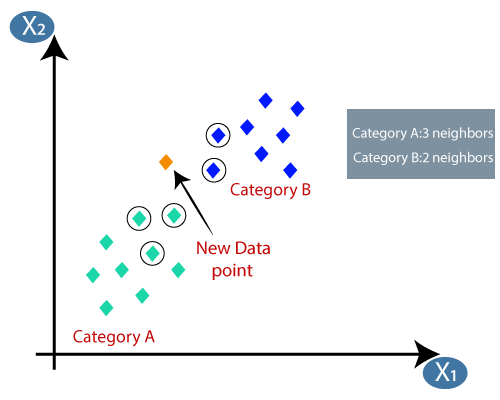
| array([0, 1, 1, 2, 1, 1, 0, 0, 2, 1, 1, 1, 2, 0, 1, 0, 2, 1, 1, 2, 2, 1,0, 1, 2, 1, 2, 2, 0, 1, 2, 1, 2, 1, 2, 2, 1, 2]) |
| --- |

These are the classes predicted for X\_test data by our naive Bayes model.

**2. K-Nearest Neighbor Algorithm**

KNN works on the very same principle. It classifies the new data points depending upon the class of the majority of data points amongst the K neighbor, where K is the number of neighbors to be considered. KNN captures the idea of similarity (sometimes called distance,

proximity, or closeness) with some basic mathematical distance formulas like euclidean distance, Manhattan distance, etc.



Choosing the right value for K

To select the K that’s right for the data you want to train, run the KNN algorithm several times with different values of K and choose that value of K which reduces the number of errors on unseen data.

Pros:

* KNN is simple and easiest to implement.
* There’s no need to build a model, tuning several parameters, or make additional assumptions like some of the other classification algorithms.
* It can be used for classification, regression, and search. So, it is flexible.

Cons:

* The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase.

Example:

| from sklearn.neighbors import KNeighborsClassifier X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=142) knn = KNeighborsClassifier(n\_neighbors=3) knn.fit(X\_train, y\_train) prediction\_results = knn.predict(X\_test[:5,:) print(prediction\_results) |
| --- |

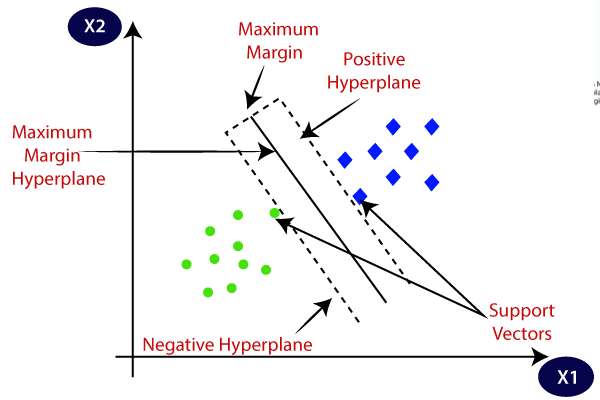
Output:

| array([0, 1, 1, 2, 1]) |
| --- |

We predicted our results for 5 sample rows. Hence we have 5 results in the array.

**3. SVM**

SVM stands for Support Vector Machine. This is a supervised machine learning algorithm that is very often used for both classification and regression challenges. However, it is mostly used in classification problems. The basic concept of the Support Vector Machine and how it works can be best understood by this simple example. So, just imagine you have two tags: green and blue, and our data has two features: x and y. We want a classifier that, given a pair of (x,y) coordinates, outputs if it’s either green or blue. Plot labeled training data on a plane and then try to find a plane (hyperplane of dimensions increases) that segregates data points of both colors very clearly.



But this is the case with data that is linear. But what if data is non-linear, then it uses kernel trick. So, to handle this we increase dimension, this brings data in space and now data becomes linearly separable in two groups.

Pros:

* SVM works relatively well when there is a clear margin of separation between classes.
* SVM is more effective in high-dimensional spaces.

Cons:

* SVM is not suitable for large data sets.
* SVM does not perform very well when the data set has more noise i.e. when target classes are overlapping. So, it needs to be handled.

Example:

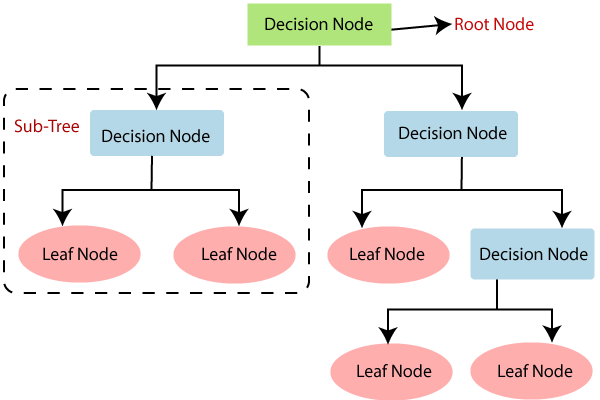
| from sklearn import svm svm\_clf = svm.SVC() X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=142) svm\_clf.fit(X\_train, y\_train) prediction\_results = svm\_clf.predict(X\_test[:7,:]) print(prediction\_results) |
| --- |

Output:

| array([0, 1, 1, 2, 1, 1, 0]) |
| --- |

**4. Decision Tree**

The decision tree is one of the most popular machine learning algorithms used. They are used for both classification and regression problems. Decision trees mimic human-level thinking so it’s so simple to understand the data and make some good intuitions and interpretations. They actually make you see the logic for the data to interpret. Decision trees are not like black-box algorithms like SVM, Neural Networks, etc.



For example, if we are classifying a person as fit or unfit then the decision tree looks somewhat like this above in the image.

So, in short, a decision tree is a tree where each node represents a

feature/attribute, each branch represents a decision, a rule, and each leaf represents an outcome. This outcome may be categorical or continuous. Categorical in case of classification and continuous in case of regression applications.

Pros:

* When compared to other algorithms, decision trees require less effort for data preparation while pre-processing.
* They do not require normalization of data and scaling as well.
* Model made on the decision tree is very intuitive and easy to explain to technical teams as well as to stakeholders also.

Cons:

* If even a small change is done in the data, that can lead to a large change in the structure of the decision tree causing instability.
* Sometimes calculation can go far more complex compared to other algorithms.
* Decision trees often take higher time to train the model.

Example:

| from sklearn import tree dtc = tree.DecisionTreeClassifier() X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=142) dtc.fit(X\_train, y\_train) prediction\_results = dtc.predict(X\_test[:7,:]) print(prediction\_results) |
| --- |

Output:

| array([0, 1, 1, 2, 1, 1, 0]) |
| --- |

**Python Library Function Used**:

Python library used for classification is scikit-learn

**sklearn.tree.DecisionTreeClassifier**

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

Some of the methods of DecisionTreeClassifier are given below

1. **fit(X, y[, sample\_weight, check\_input, ...]):**

Build a decision tree classifier from the training set (X, y).

1. **predict(X[, check\_input]):**

Predict class or regression value for X.

1. **score(X, y[, sample\_weight]):**

Return the mean accuracy on the given test data and labels.

1. **set\_params(\*\*params):**

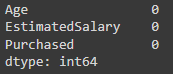
Set the parameters of this estimator.

**Data Modeling and Analysis**

**Dataset**: Social Network Ads

**Preprocessing**:

df.isnull().sum()



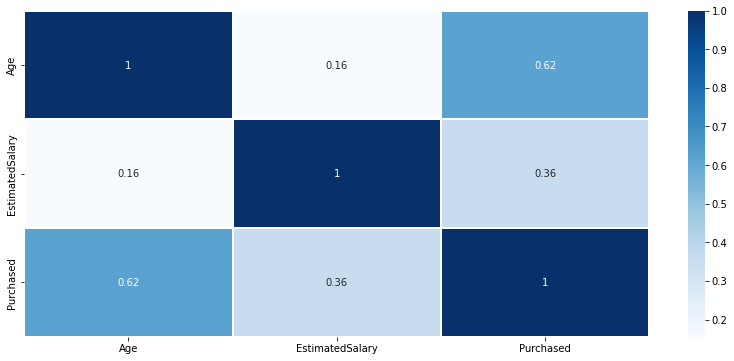
**Correlation Analysis**:

plt.figure(figsize=(14,6))

corr=abs(df.corr())

sns.heatmap(corr,annot=True,linewidth=1,cmap="Blues")

plt.show()



**Train Test Split**:

X = df.drop(columns=['Purchased'])

y = df['Purchased']

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3,random\_state=1)

**Feature Scaling**:

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

**Visualizing Classification in Training dataset**

from matplotlib.colors import ListedColormap

X\_set, y\_set = sc.inverse\_transform(X\_train), y\_train

X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 10, stop = X\_set[:, 0].max() + 10, step = 0.25),

np.arange(start = X\_set[:, 1].min() - 1000, stop = X\_set[:, 1].max() + 1000, step = 0.25))

plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape),

alpha = 0.75, cmap = ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)

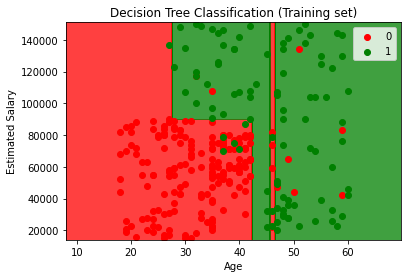
plt.title('Decision Tree Classification (Training set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()



**Visualizing Classification in Testing dataset**

from matplotlib.colors import ListedColormap

X\_set, y\_set = sc.inverse\_transform(X\_test), y\_test

X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 10, stop = X\_set[:, 0].max() + 10, step = 0.25),

np.arange(start = X\_set[:, 1].min() - 1000, stop = X\_set[:, 1].max() + 1000, step = 0.25))

plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape),

alpha = 0.75, cmap = ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)

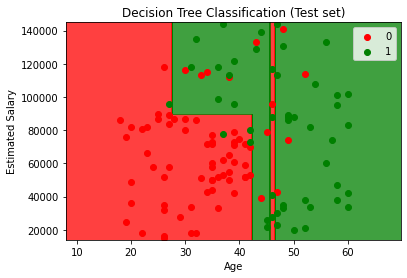
plt.title('Decision Tree Classification (Test set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()



**Code and Observation**:

**Classification using inbuilt library function**

from sklearn.tree import DecisionTreeClassifier

classifier = DecisionTreeClassifier(criterion = 'gini', random\_state = 0,max\_depth=3)

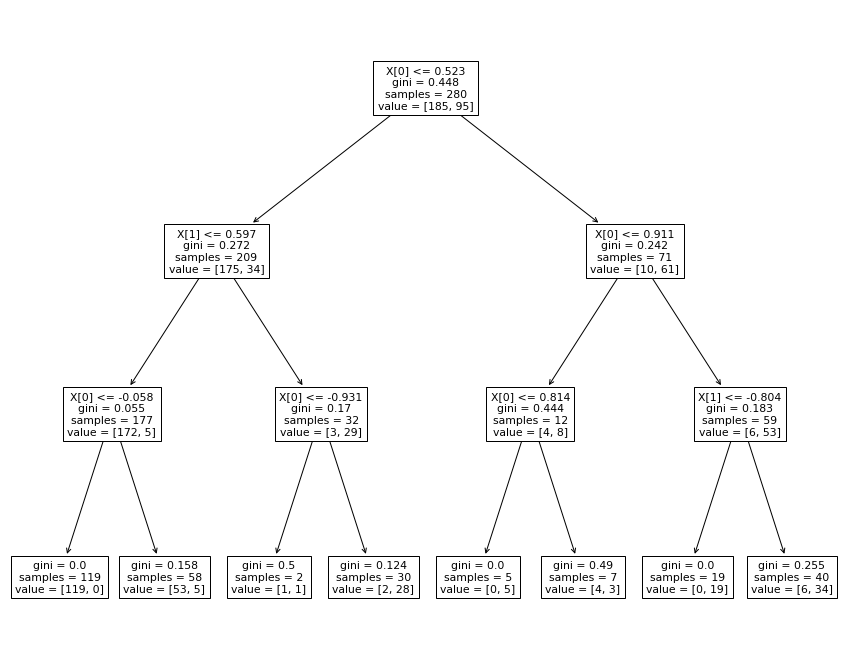
classifier.fit(X\_train, y\_train)

from sklearn import tree

import matplotlib.pyplot as plt

plt.figure(figsize=(15,12))

tree.plot\_tree(classifier)



y\_pred = classifier.predict(X\_test)

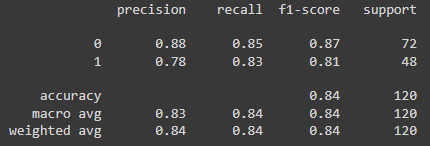
from sklearn.metrics import confusion\_matrix, accuracy\_score, classification\_report

cm = confusion\_matrix(y\_test, y\_pred)

print(cm)



print(classification\_report(y\_test, y\_pred))



accuracy\_score(y\_test, y\_pred)



**Classification using user-defined function**

from collections import Counter

class Node:

"""

Class for creating the nodes for a decision tree

"""

def \_\_init\_\_(

self,

Y: list,

X: pd.DataFrame,

min\_samples\_split=None,

max\_depth=None,

depth=None,

node\_type=None,

rule=None

):

# Saving the data to the node

self.Y = Y

self.X = X

# Saving the hyper parameters

self.min\_samples\_split = min\_samples\_split if min\_samples\_split else 20

self.max\_depth = max\_depth if max\_depth else 5

# Default current depth of node

self.depth = depth if depth else 0

# Extracting all the features

self.features = list(self.X.columns)

# Type of node

self.node\_type = node\_type if node\_type else 'root'

# Rule for spliting

self.rule = rule if rule else ""

# Calculating the counts of Y in the node

self.counts = Counter(Y)

# Getting the GINI impurity based on the Y distribution

self.gini\_impurity = self.get\_GINI()

# Sorting the counts and saving the final prediction of the node

counts\_sorted = list(sorted(self.counts.items(), key=lambda item: item[1]))

# Getting the last item

yhat = None

if len(counts\_sorted) > 0:

yhat = counts\_sorted[-1][0]

# Saving to object attribute. This node will predict the class with the most frequent class

self.yhat = yhat

# Saving the number of observations in the node

self.n = len(Y)

# Initiating the left and right nodes as empty nodes

self.left = None

self.right = None

# Default values for splits

self.best\_feature = None

self.best\_value = None

@staticmethod

def GINI\_impurity(y1\_count: int, y2\_count: int) -> float:

"""

Given the observations of a binary class calculate the GINI impurity

"""

# Ensuring the correct types

if y1\_count is None:

y1\_count = 0

if y2\_count is None:

y2\_count = 0

# Getting the total observations

n = y1\_count + y2\_count

# If n is 0 then we return the lowest possible gini impurity

if n == 0:

return 0.0

# Getting the probability to see each of the classes

p1 = y1\_count / n

p2 = y2\_count / n

# Calculating GINI

gini = 1 - (p1 \*\* 2 + p2 \*\* 2)

# Returning the gini impurity

return gini

@staticmethod

def ma(x: np.array, window: int) -> np.array:

"""

Calculates the moving average of the given list.

"""

return np.convolve(x, np.ones(window), 'valid') / window

def get\_GINI(self):

"""

Function to calculate the GINI impurity of a node

"""

# Getting the 0 and 1 counts

y1\_count, y2\_count = self.counts.get(0, 0), self.counts.get(1, 0)

# Getting the GINI impurity

return self.GINI\_impurity(y1\_count, y2\_count)

def best\_split(self) -> tuple:

"""

Given the X features and Y targets calculates the best split

for a decision tree

"""

# Creating a dataset for spliting

df = self.X.copy()

df['Y'] = self.Y

# Getting the GINI impurity for the base input

GINI\_base = self.get\_GINI()

# Finding which split yields the best GINI gain

max\_gain = 0

# Default best feature and split

best\_feature = None

best\_value = None

for feature in self.features:

# Droping missing values

Xdf = df.dropna().sort\_values(feature)

# Sorting the values and getting the rolling average

xmeans = self.ma(Xdf[feature].unique(), 2)

for value in xmeans:

# Spliting the dataset

left\_counts = Counter(Xdf[Xdf[feature]<value]['Y'])

right\_counts = Counter(Xdf[Xdf[feature]>=value]['Y'])

# Getting the Y distribution from the dicts

y0\_left, y1\_left, y0\_right, y1\_right = left\_counts.get(0, 0), left\_counts.get(1, 0), right\_counts.get(0, 0), right\_counts.get(1, 0)

# Getting the left and right gini impurities

gini\_left = self.GINI\_impurity(y0\_left, y1\_left)

gini\_right = self.GINI\_impurity(y0\_right, y1\_right)

# Getting the obs count from the left and the right data splits

n\_left = y0\_left + y1\_left

n\_right = y0\_right + y1\_right

# Calculating the weights for each of the nodes

w\_left = n\_left / (n\_left + n\_right)

w\_right = n\_right / (n\_left + n\_right)

# Calculating the weighted GINI impurity

wGINI = w\_left \* gini\_left + w\_right \* gini\_right

# Calculating the GINI gain

GINIgain = GINI\_base - wGINI

# Checking if this is the best split so far

if GINIgain > max\_gain:

best\_feature = feature

best\_value = value

# Setting the best gain to the current one

max\_gain = GINIgain

return (best\_feature, best\_value)

def grow\_tree(self):

"""

Recursive method to create the decision tree

"""

# Making a df from the data

df = self.X.copy()

df['Y'] = self.Y

# If there is GINI to be gained, we split further

if (self.depth < self.max\_depth) and (self.n >= self.min\_samples\_split):

# Getting the best split

best\_feature, best\_value = self.best\_split()

if best\_feature is not None:

# Saving the best split to the current node

self.best\_feature = best\_feature

self.best\_value = best\_value

# Getting the left and right nodes

left\_df, right\_df = df[df[best\_feature]<=best\_value].copy(), df[df[best\_feature]>best\_value].copy()

# Creating the left and right nodes

left = Node(

left\_df['Y'].values.tolist(),

left\_df[self.features],

depth=self.depth + 1,

max\_depth=self.max\_depth,

min\_samples\_split=self.min\_samples\_split,

node\_type='left\_node',

rule=f"{best\_feature} <= {round(best\_value, 3)}"

)

self.left = left

self.left.grow\_tree()

right = Node(

right\_df['Y'].values.tolist(),

right\_df[self.features],

depth=self.depth + 1,

max\_depth=self.max\_depth,

min\_samples\_split=self.min\_samples\_split,

node\_type='right\_node',

rule=f"{best\_feature} > {round(best\_value, 3)}"

)

self.right = right

self.right.grow\_tree()

def print\_info(self, width=4):

"""

Method to print the infromation about the tree

"""

# Defining the number of spaces

const = int(self.depth \* width \*\* 1.5)

spaces = "-" \* const

if self.node\_type == 'root':

print("Root")

else:

print(f"|{spaces} Split rule: {self.rule}")

print(f"{' ' \* const} | GINI impurity of the node: {round(self.gini\_impurity, 2)}")

print(f"{' ' \* const} | Class distribution in the node: {dict(self.counts)}")

print(f"{' ' \* const} | Predicted class: {self.yhat}")

def print\_tree(self):

"""

Prints the whole tree from the current node to the bottom

"""

self.print\_info()

if self.left is not None:

self.left.print\_tree()

if self.right is not None:

self.right.print\_tree()

def predict(self, X:pd.DataFrame):

"""

Batch prediction method

"""

predictions = []

for \_, x in X.iterrows():

values = {}

for feature in self.features:

values.update({feature: x[feature]})

predictions.append(self.predict\_obs(values))

return predictions

def predict\_obs(self, values: dict) -> int:

"""

Method to predict the class given a set of features

"""

cur\_node = self

while cur\_node.depth < cur\_node.max\_depth:

# Traversing the nodes all the way to the bottom

best\_feature = cur\_node.best\_feature

best\_value = cur\_node.best\_value

if cur\_node.n < cur\_node.min\_samples\_split:

break

if (values.get(best\_feature) < best\_value):

if self.left is not None:

cur\_node = cur\_node.left

else:

if self.right is not None:

cur\_node = cur\_node.right

return cur\_node.yhat

X = df.iloc[:, :-1].values

Y = df.iloc[:, -1].values.reshape(-1,1)

from sklearn.model\_selection import train\_test\_split

train, test = train\_test\_split(df, test\_size=0.2)

# Constructing the X and Y matrices

X = train[['EstimatedSalary', 'Age']]

Y = train['Purchased'].values.tolist()

# Initiating the Node

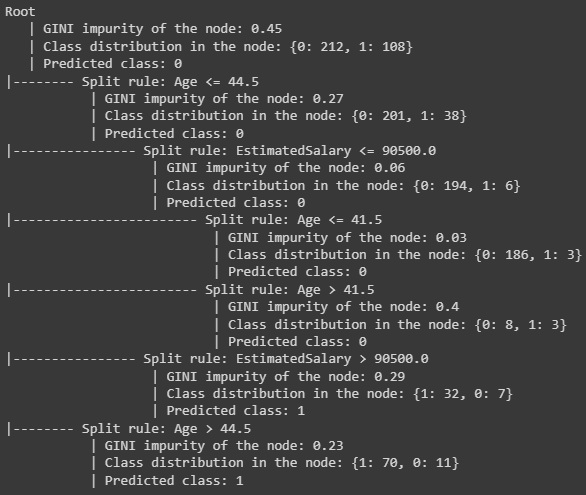
root = Node(Y, X, max\_depth=3, min\_samples\_split=100)

# Getting the best split

root.grow\_tree()

# Printing the tree information

root.print\_tree()



# Predicting

pred = root.predict(test)

actual = list(test['Purchased'])

accuracy = sum(1 for x,y in zip(actual,pred) if x == y) / len(actual)

print("Accuracy:",accuracy)



**Comparing Metrics:**

|  | Accuracy |
| --- | --- |
| Using Python Libraries | 84.167 |
| Using User-Defined Function | 88.750 |

Thus, classification using our defined class function gave better results, as compared to the classification using Python Libraries.

**Conclusion**:

Thus, we have learnt about classification and learnt how to implement it using python libraries and our own function.