Name : Neeraj Rijhwani Roll No. 64 Class:D15C

Experiment : 6

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)
```

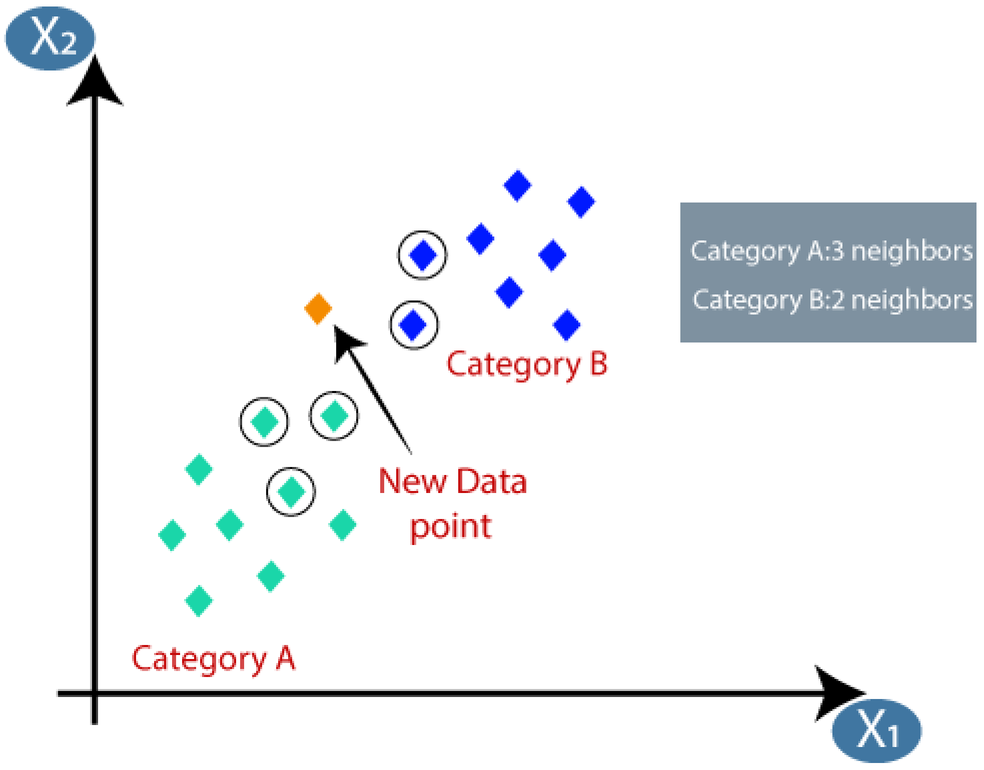
Output:

```
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)
```

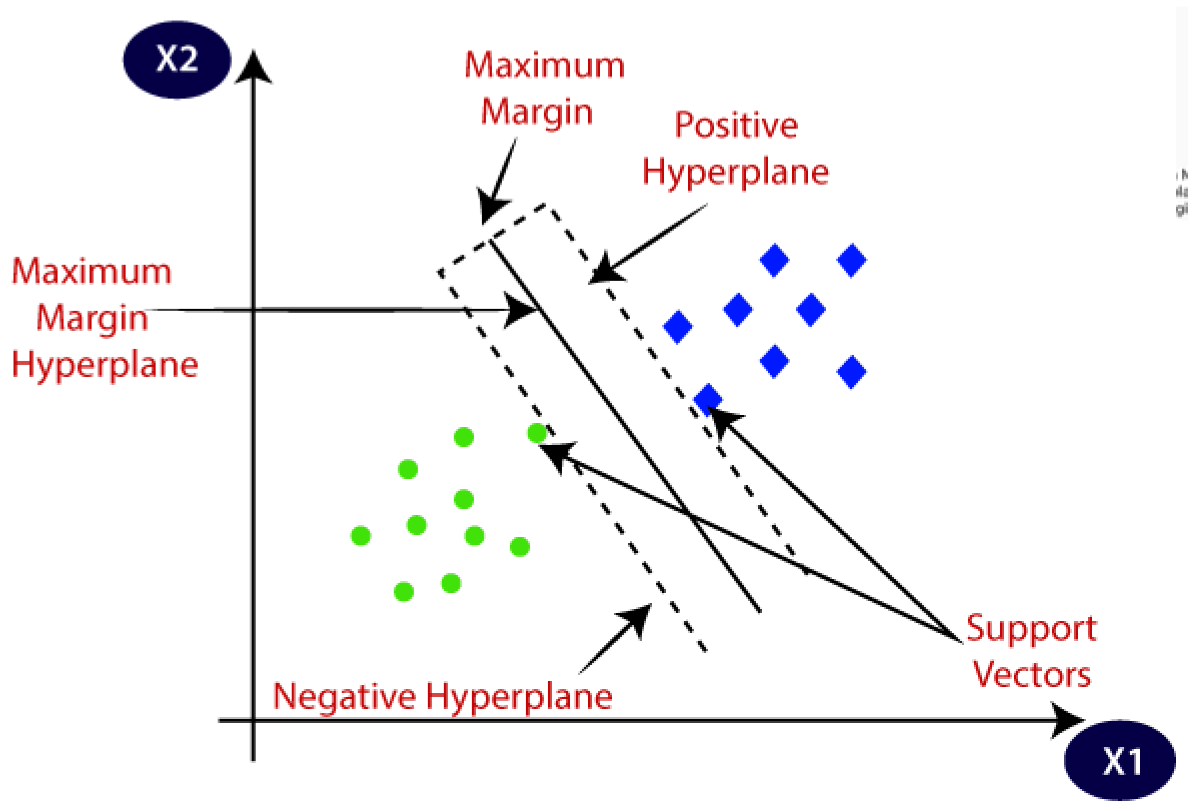
Output:

```
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)
```

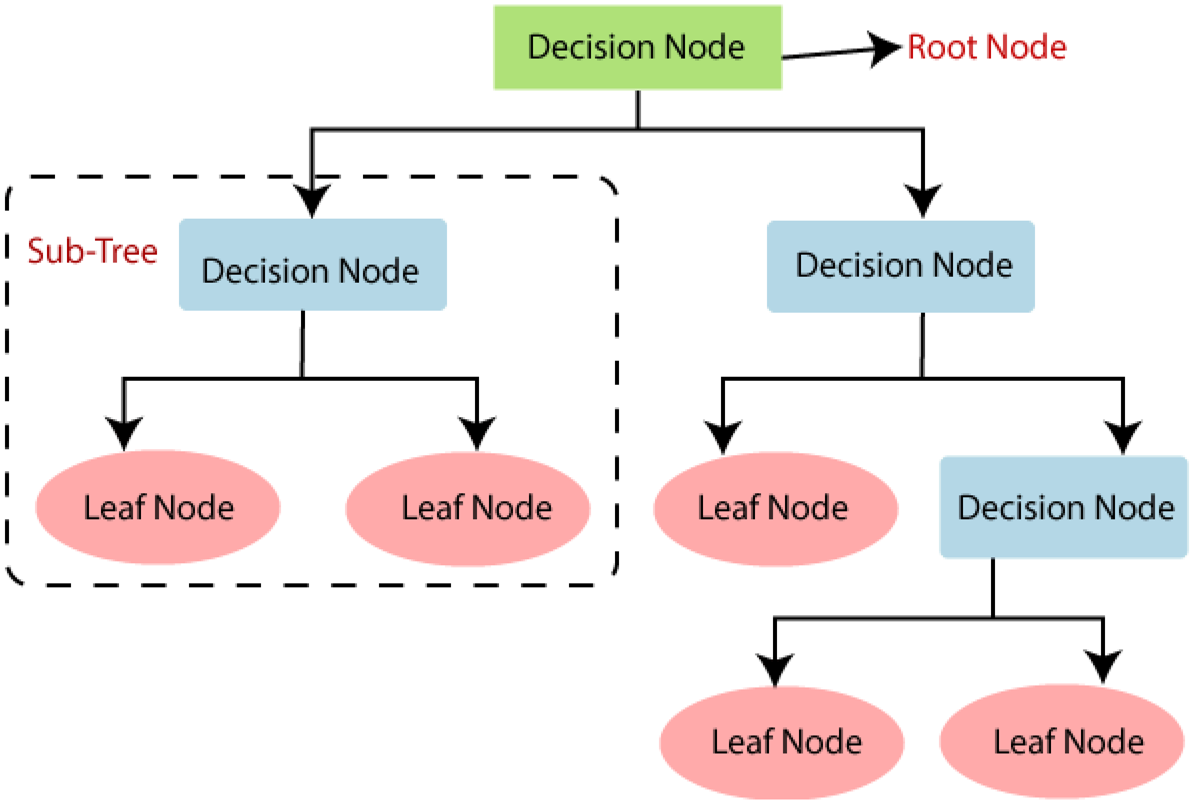
```
prediction_results = svm_clf.predict(X_test[:7,:])
print(prediction_results)
```

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])
```

<u>Python Library Function Used</u>:

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).

2. **predict(X[, check_input]):**
   Predict class or regression value for X.

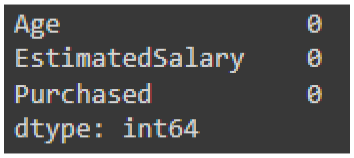
3. **score(X, y[, sample_weight]):**
   Return the mean accuracy on the given test data and labels.

4. **set_params(**params):**
   Set the parameters of this estimator.

Data Modeling and Analysis

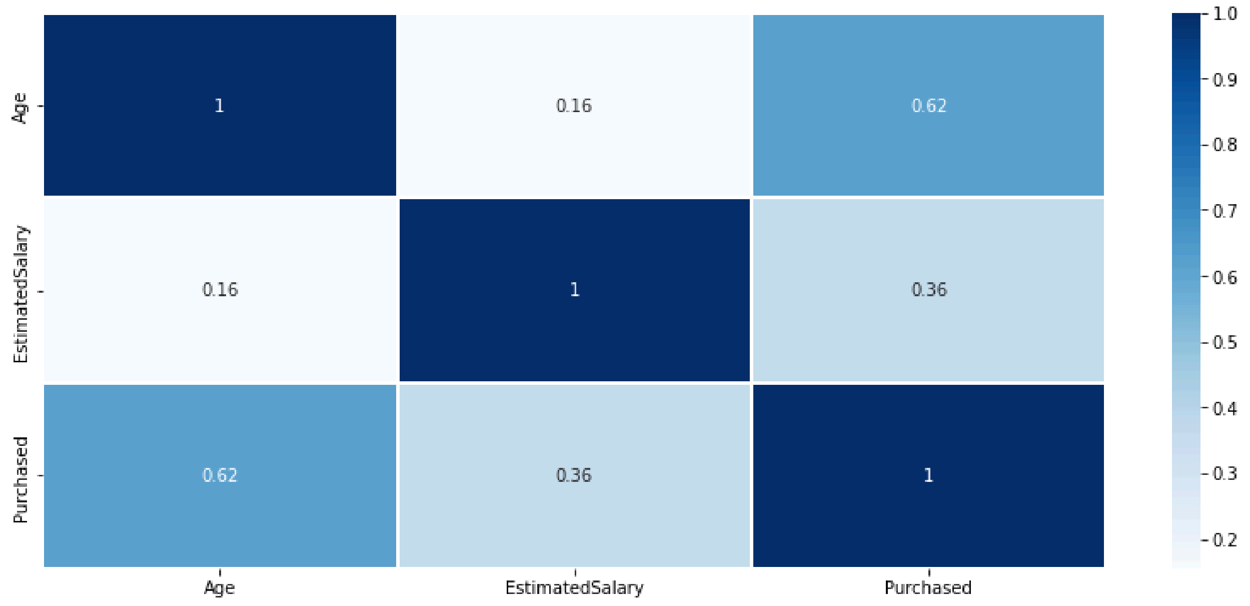
Dataset: Social Network Ads

Preprocessing:
df.isnull().sum()



```
Age                0
EstimatedSalary    0
Purchased          0
dtype: int64
```

Correlation Analysis:
plt.figure(figsize=(14,6))
corr=abs(df.corr())
sns.heatmap(corr,annot=True,linewidth=1,cmap="Blues")
plt.show()



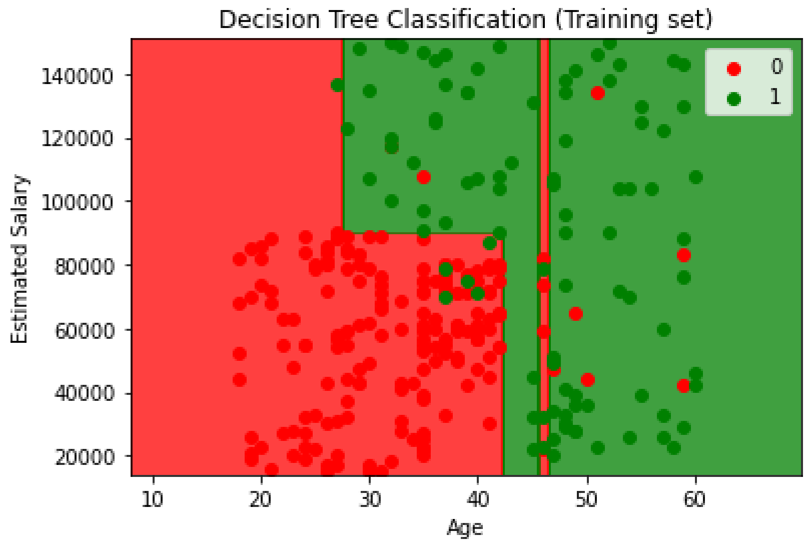
Train Test Split:
X = df.drop(columns=['Purchased'])
y = df['Purchased']
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,random_state=1)

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

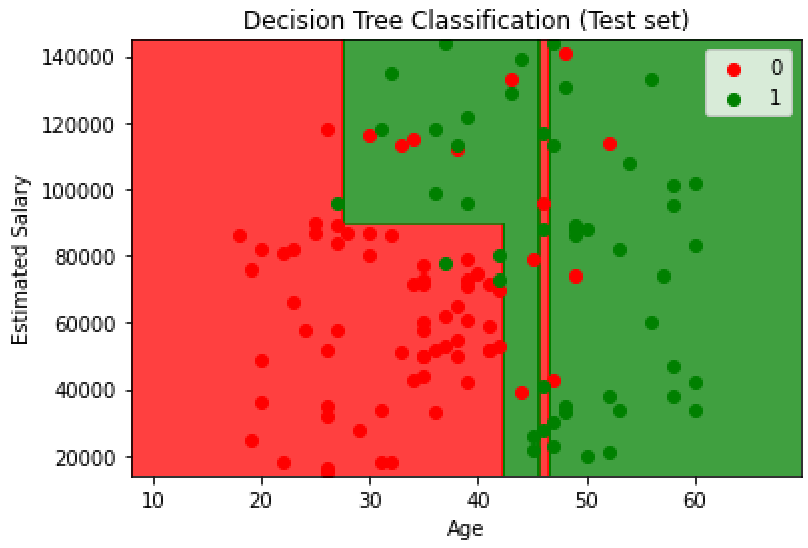
```
from matplotlib.colors import ListedColormap

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()
```

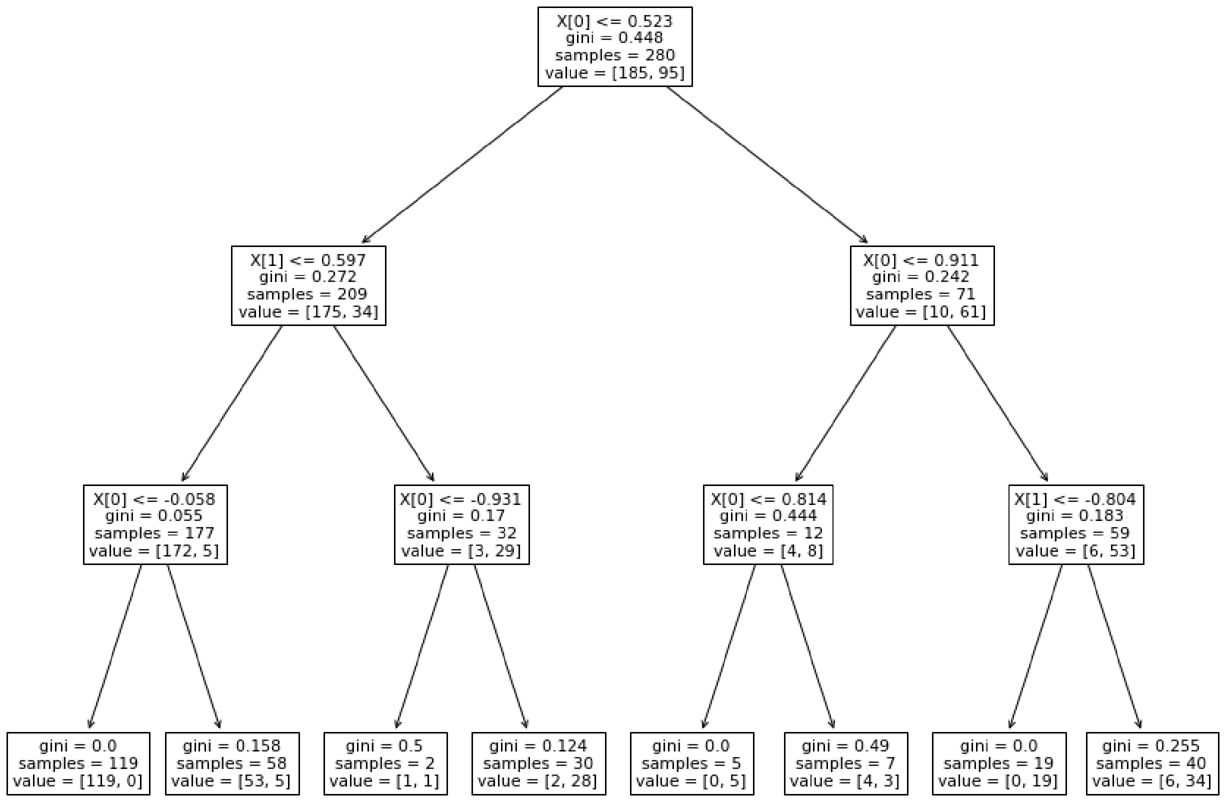


<u>Code and Observation</u>:

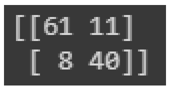
Classification using inbuilt library function

```
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)
```

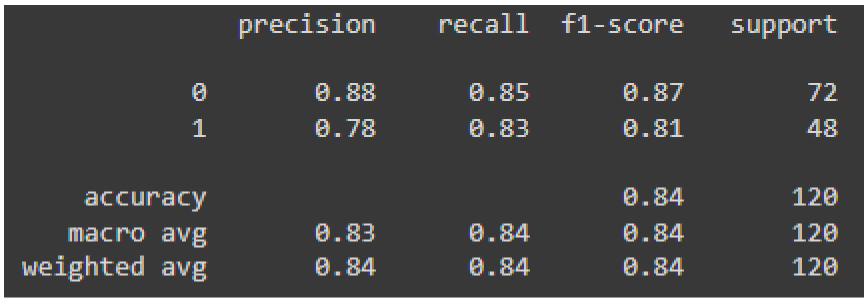


```
y_pred = classifier.predict(X_test)
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report
cm = confusion_matrix(y_test, y_pred)
print(cm)
```



```
[[61 11]
 [ 8 40]]
```

```
print(classification_report(y_test, y_pred))
```

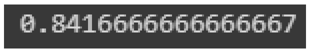


```
              precision    recall  f1-score   support

           0       0.88      0.85      0.87        72
           1       0.78      0.83      0.81        48

    accuracy                           0.84       120
   macro avg       0.83      0.84      0.84       120
weighted avg       0.84      0.84      0.84       120
```

```
accuracy_score(y_test, y_pred)
```



```
0.8416666666666667
```

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)
```

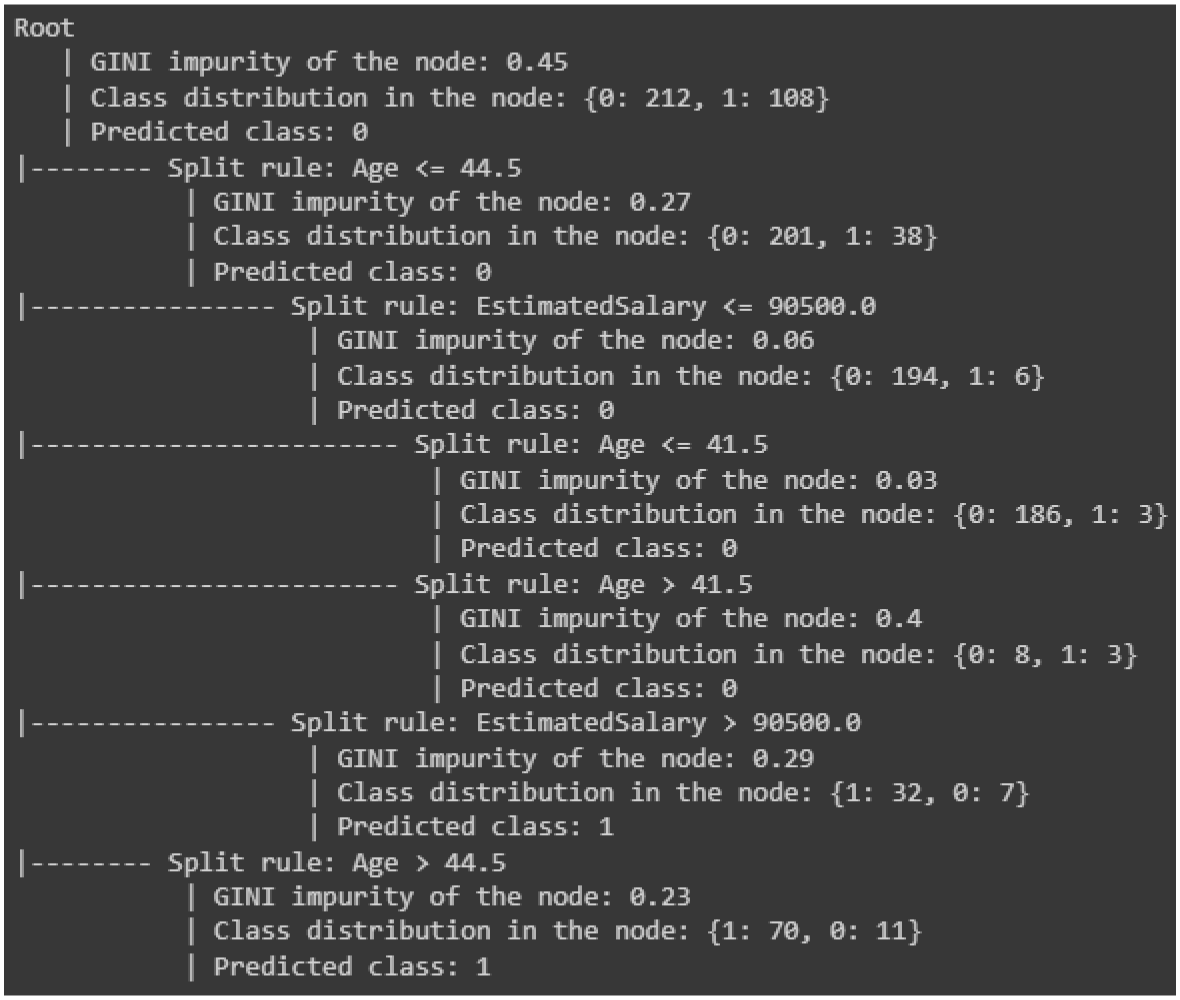
```
from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.2)

# 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()
```



```
Root
   | GINI impurity of the node: 0.45
   | Class distribution in the node: {0: 212, 1: 108}
   | Predicted class: 0
|-------- Split rule: Age <= 44.5
           | GINI impurity of the node: 0.27
           | Class distribution in the node: {0: 201, 1: 38}
           | Predicted class: 0
|---------------- Split rule: EstimatedSalary <= 90500.0
                   | GINI impurity of the node: 0.06
                   | Class distribution in the node: {0: 194, 1: 6}
                   | Predicted class: 0
|------------------------ Split rule: Age <= 41.5
                           | GINI impurity of the node: 0.03
                           | Class distribution in the node: {0: 186, 1: 3}
                           | Predicted class: 0
|------------------------ Split rule: Age > 41.5
                           | GINI impurity of the node: 0.4
                           | Class distribution in the node: {0: 8, 1: 3}
                           | Predicted class: 0
|---------------- Split rule: EstimatedSalary > 90500.0
                   | GINI impurity of the node: 0.29
                   | Class distribution in the node: {1: 32, 0: 7}
                   | Predicted class: 1
|-------- Split rule: Age > 44.5
           | GINI impurity of the node: 0.23
           | Class distribution in the node: {1: 70, 0: 11}
           | Predicted class: 1
```

```
# 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)
```

```
Accuracy: 0.8875
```

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.

<u>**Conclusion**</u>:

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