Loan Approval Status

Introduction

Loan Approval Prediction is one of the problems that Machine Learning has solved in fintech businesses like banks and financial institutions. Loan approval prediction means using credit history data of the loan applicants and algorithms to build an intelligent system that can determine loan approvals.

Loan approval prediction involves the analysis of various factors, such as the applicant's financial history, income, credit rating, employment status, and other relevant attributes. By leveraging historical loan data and applying machine learning algorithms, businesses can build models to determine loan approvals for new applicants.

Import all the liabraries -

Pandas - Used to analyse data. It has function for analysing,cleaning,exploring and manipulating data. Numpy - Mostly work on numerical values for making Arithmatic Operations.

Matplotlib - Comprehensive library for creating static,animated and intractive visualization.

Seaborn - Seaborn is a python data visualization library based on matplotlib. It provides a high-level interface for drawing intractive and informative statastical graphics.

Warnings - warnings are provided to warn the developer of situation that are not necessarily exceptions and ignore them.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
df=pd.read_csv("loan_approval_dataset.csv")
```

In [3]:
```
df
```

Out[3]:

	loan_id	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_te
0	1	2	Graduate	No	9600000	29900000	
1	2	0	Not Graduate	Yes	4100000	12200000	
2	3	3	Graduate	No	9100000	29700000	
3	4	3	Graduate	No	8200000	30700000	
4	5	5	Not Graduate	Yes	9800000	24200000	
...	...	...	...	...	...	...	
4264	4265	5	Graduate	Yes	1000000	2300000	
4265	4266	0	Not Graduate	Yes	3300000	11300000	
4266	4267	2	Not Graduate	No	6500000	23900000	
4267	4268	1	Not Graduate	No	4100000	12800000	
4268	4269	1	Graduate	No	9200000	29700000	

4269 rows × 13 columns

df.info() it will display information like the number of non-null values, data types, and memory usage for each column. This can be helpful for understanding the structure and quality of data..

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4269 entries, 0 to 4268
Data columns (total 13 columns):
 #   Column                     Non-Null Count  Dtype 
---  ------                     --------------  ----- 
 0   loan_id                    4269 non-null   int64 
 1    no_of_dependents          4269 non-null   int64 
 2    education                 4269 non-null   object
 3    self_employed             4269 non-null   object
 4    income_annum              4269 non-null   int64 
 5    loan_amount               4269 non-null   int64 
 6    loan_term                 4269 non-null   int64 
 7    cibil_score               4269 non-null   int64 
 8    residential_assets_value  4269 non-null   int64 
 9    commercial_assets_value   4269 non-null   int64 
 10   luxury_assets_value       4269 non-null   int64 
 11   bank_asset_value          4269 non-null   int64 
 12   loan_status               4269 non-null   object
dtypes: int64(10), object(3)
memory usage: 433.7+ KB
```

In [5]:
```
df.isnull().sum()
```

Out[5]:
```
loan_id                      0
 no_of_dependents            0
 education                   0
 self_employed               0
 income_annum                0
 loan_amount                 0
 loan_term                   0
 cibil_score                 0
 residential_assets_value    0
 commercial_assets_value     0
 luxury_assets_value         0
 bank_asset_value            0
 loan_status                 0
dtype: int64
```

The df.isnull().sum() function is used to check for missing (null) values in a DataFrame and get the count of missing values for each column

In [6]:
```
df.drop("loan_id",axis=1,inplace=True)
```

using the Pandas DataFrame method drop to remove a column named "loan_id" from DataFrame df. The axis=1 argument specifies that dropping a column, and inplace=True means that the change will be applied to the DataFrame df directly, without the need to reassign it.

In [7]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4269 entries, 0 to 4268
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype 
---  ------                     --------------  ----- 
 0    no_of_dependents          4269 non-null   int64 
 1    education                 4269 non-null   object
 2    self_employed             4269 non-null   object
 3    income_annum              4269 non-null   int64 
 4    loan_amount               4269 non-null   int64 
 5    loan_term                 4269 non-null   int64 
 6    cibil_score               4269 non-null   int64 
 7    residential_assets_value  4269 non-null   int64 
 8    commercial_assets_value   4269 non-null   int64 
 9    luxury_assets_value       4269 non-null   int64 
 10   bank_asset_value          4269 non-null   int64 
 11   loan_status               4269 non-null   object
dtypes: int64(9), object(3)
memory usage: 400.3+ KB
```

All the column names contain a space in front of the text, we need to trim them up to avoid future confusions.

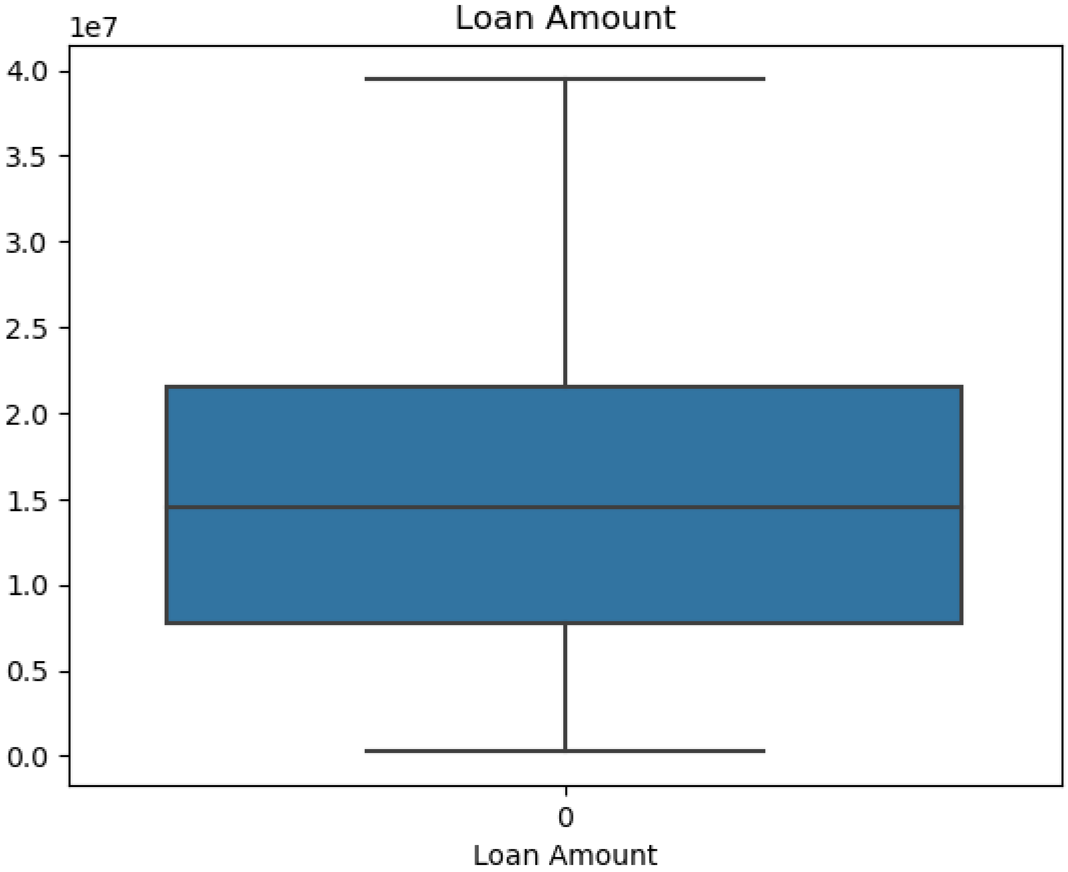
In [8]:
```
df.columns = df.columns.str.replace(' ', '')
```

Let's take a look at the distribution of loan amounts.

In [9]:
```
df["loan_amount"].value_counts()
```

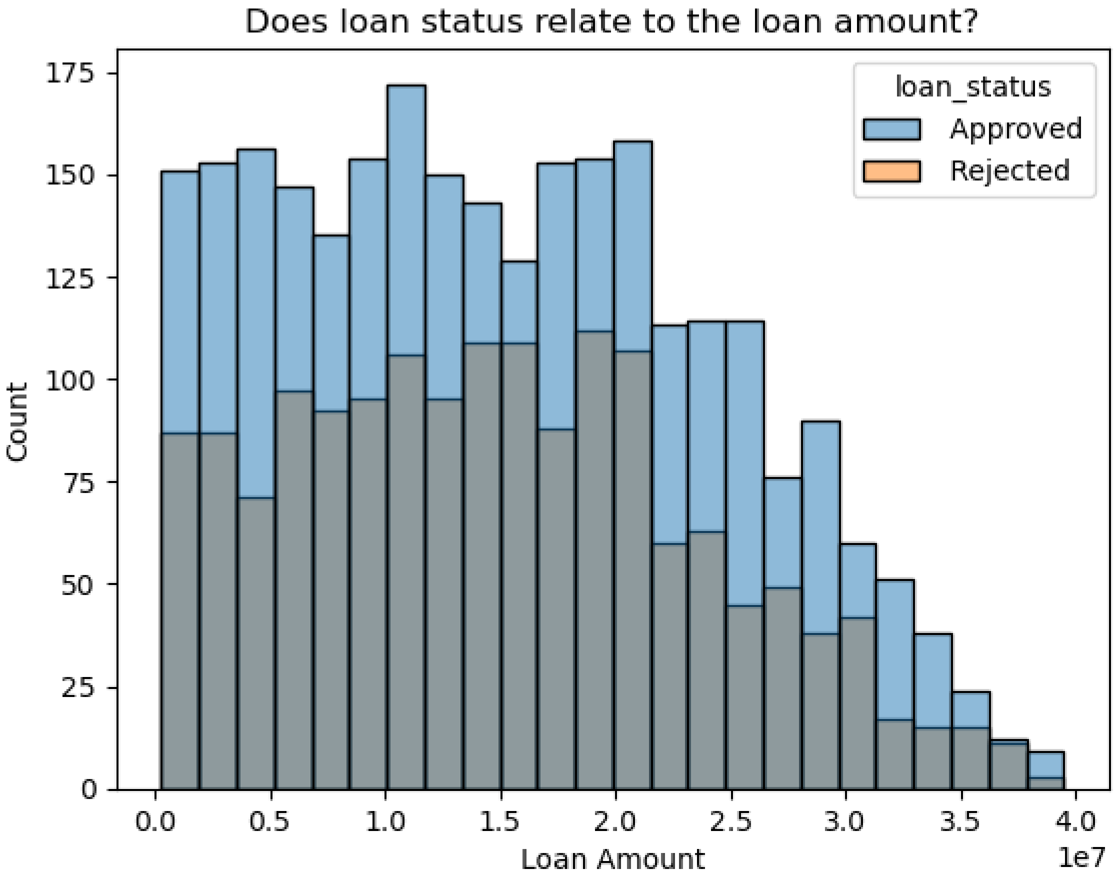
Out[9]:
```
10600000    27
20000000    24
9400000     24
16800000    23
23900000    23
            ..
35800000     1
38500000     1
39500000     1
38800000     1
36100000     1
Name: loan_amount, Length: 378, dtype: int64
```

In [10]:
```
sns.boxplot(df['loan_amount'])
plt.title("Loan Amount")
plt.xlabel("Loan Amount")
plt.show()
```



This code will generate a boxplot of the "loan_amount" column, which is a great way to visualize the distribution and statistical characteristics of the loan amounts in dataset. The boxplot will display the median, quartiles, and any potential outliers.

In [11]:
```
sns.histplot(df, x='loan_amount', hue='loan_status')
plt.title("Does loan status relate to the loan amount?")
plt.xlabel("Loan Amount")
plt.ylabel("Count")
plt.show()
```



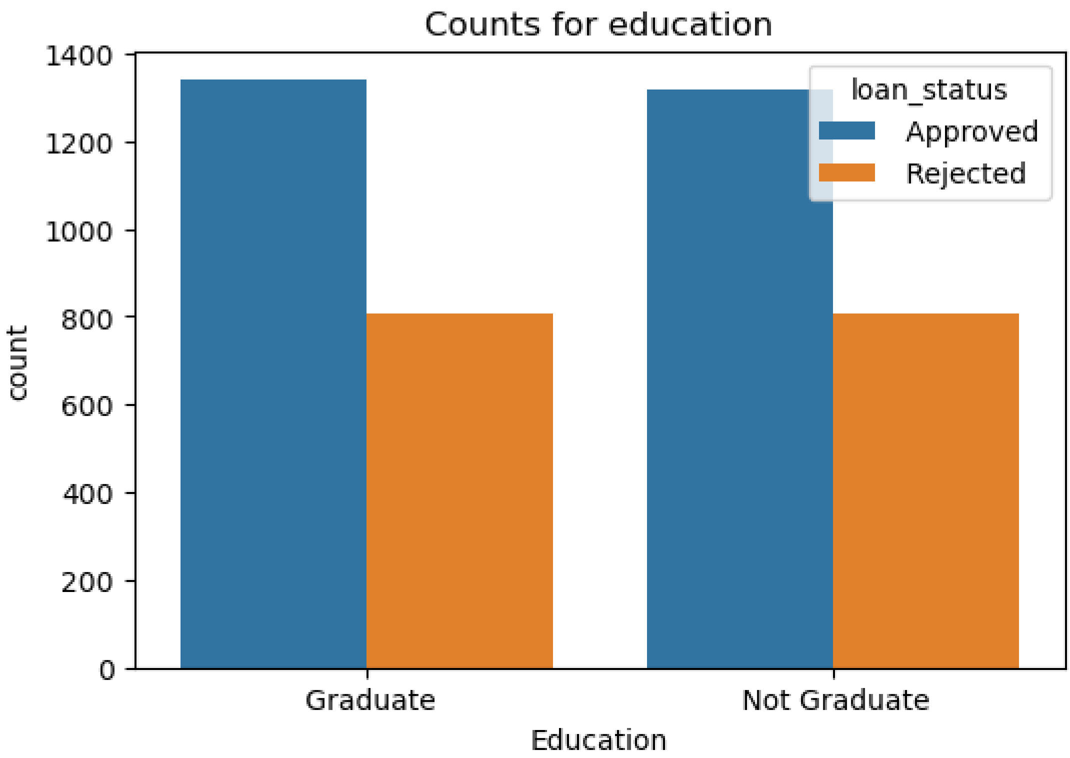
This code will create a histogram where the x-axis represents the loan amount, and different colors (or shades) will represent different loan statuses. It allows visually compare the distribution of loan amounts across various loan statuses and see if there are any noticeable patterns or differences.

In [12]:
```
sns.scatterplot(x=df['income_annum'], y= df['loan_amount'], hue=df['loan_status'])
plt.title("Loan Status, Loan Amount, Annual Income")
plt.xlabel("Annual Income")
plt.ylabel("Loan Amount")
plt.show()
```

This code will generate a scatterplot where the x-axis represents annual income, the y-axis represents loan amount, and different colors (distinguished by hue) represent different loan statuses. This visualization can helpcunderstand the relationship between these variables and see if there are any patterns or trends.

In [13]:
```
plt.figure(figsize = (6,4))
sns.countplot(data=df, x='education', hue='loan_status')
plt.xlabel("Education")
plt.title("Counts for education")

plt.show()
```

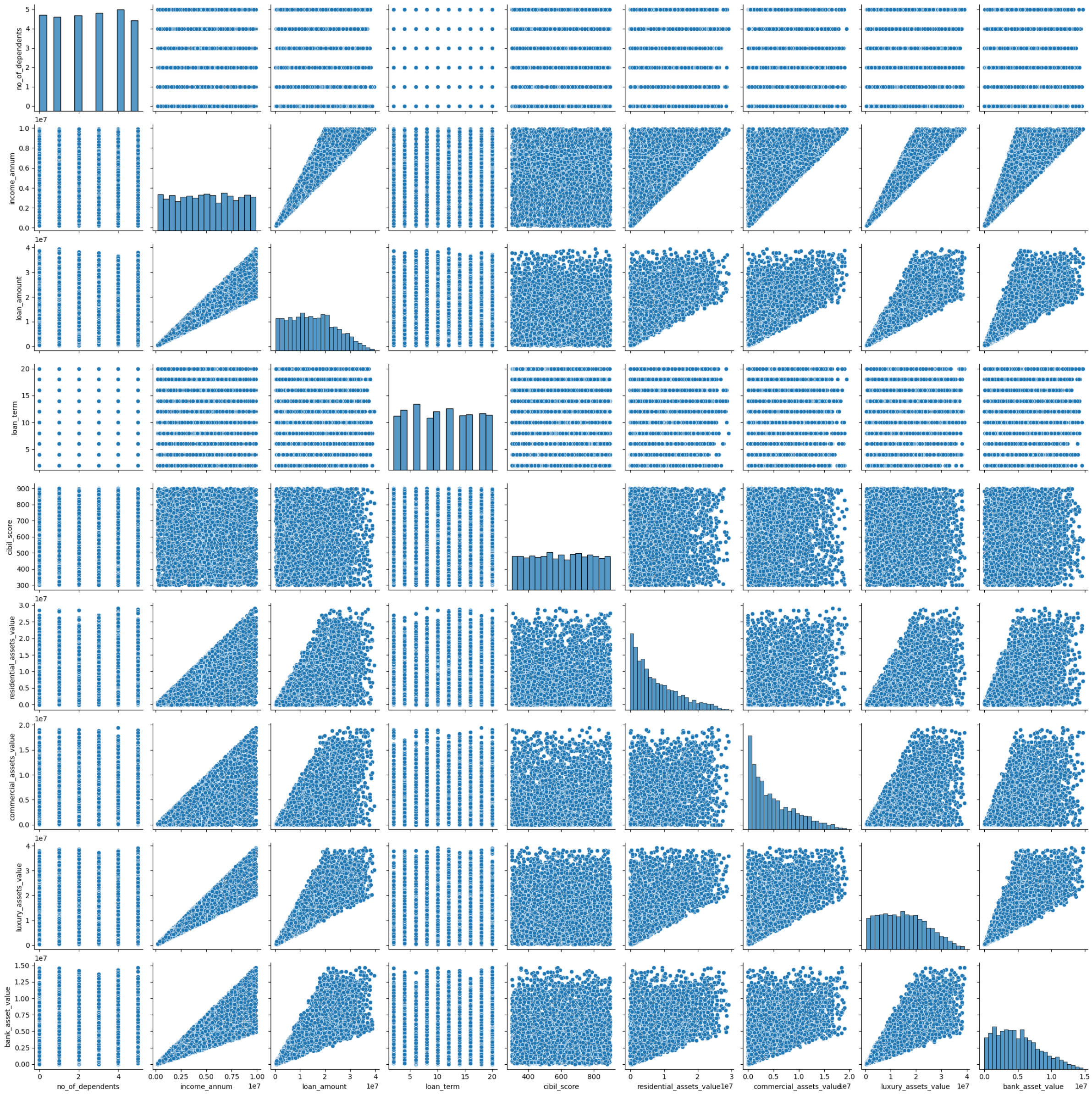


This code will generate a countplot where the x-axis represents education levels, and different colors (distinguished by hue) represent different loan statuses.the distribution of loan statuses within different education categories.

The counts based on different education status are approximately the same.

In [14]:
```
sns.pairplot(df)
```

Out[14]:
```
<seaborn.axisgrid.PairGrid at 0x26c11a2c850>
```



This code will generate a grid of scatterplots, where each plot shows the relationship between two numeric variables in DataFrame. It's a useful way to visually explore correlations and patterns between different variables.

Divide the Data into Numeric and Categorical form

In [15]:

```
#In the code you've provided, you're separating your DataFrame, df, into numerical

num_feature=df.select_dtypes(["int64","float64"])
cat_feature=df.select_dtypes(["object"]).columns
```

In [16]:

```
#num_feature: This DataFrame contains all the columns with data types 'int64' and '

num_feature
```

Out[16]:

	no_of_dependents	income_annum	loan_amount	loan_term	cibil_score	residential_assets_va
0	2	9600000	29900000	12	778	2400(
1	0	4100000	12200000	8	417	2700(
2	3	9100000	29700000	20	506	7100(
3	3	8200000	30700000	8	467	18200(
4	5	9800000	24200000	20	382	12400(
...	...	...	...	...	...	
4264	5	1000000	2300000	12	317	2800(
4265	0	3300000	11300000	20	559	4200(
4266	2	6500000	23900000	18	457	1200(
4267	1	4100000	12800000	8	780	8200(
4268	1	9200000	29700000	10	607	17800(

4269 rows × 9 columns

In [17]:

```
#cat_feature: This variable contains the names of the columns in df with data type

cat_feature
```

Out[17]:

```
Index(['education', 'self_employed', 'loan_status'], dtype='object')
```

In [18]:

```
from sklearn.preprocessing import OrdinalEncoder
oe=OrdinalEncoder()
df[cat_feature]=oe.fit_transform(df[cat_feature])
```

This code will replace the categorical values in the specified cat_feature columns with ordinal integers. It's essential to ensure that this encoding is appropriate for specific dataset and the nature of the categorical variables, as it assumes an ordinal relationship between the categories`

In [19]:

```
df
```

Out[19]:

	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_
0	2	0.0	0.0	9600000	29900000	12	
1	0	1.0	1.0	4100000	12200000	8	
2	3	0.0	0.0	9100000	29700000	20	
3	3	0.0	0.0	8200000	30700000	8	
4	5	1.0	1.0	9800000	24200000	20	
...	...	...	...	...	...	...	
4264	5	0.0	1.0	1000000	2300000	12	
4265	0	1.0	1.0	3300000	11300000	20	
4266	2	1.0	0.0	6500000	23900000	18	
4267	1	1.0	0.0	4100000	12800000	8	
4268	1	0.0	0.0	9200000	29700000	10	

4269 rows × 12 columns

Splitting Data into Features And Target

In [20]:
```
x=df.iloc[:,:-1]
x
```

Out[20]:

	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_
0	2	0.0	0.0	9600000	29900000	12	
1	0	1.0	1.0	4100000	12200000	8	
2	3	0.0	0.0	9100000	29700000	20	
3	3	0.0	0.0	8200000	30700000	8	
4	5	1.0	1.0	9800000	24200000	20	
...	...	...	...	...	...	...	
4264	5	0.0	1.0	1000000	2300000	12	
4265	0	1.0	1.0	3300000	11300000	20	
4266	2	1.0	0.0	6500000	23900000	18	
4267	1	1.0	0.0	4100000	12800000	8	
4268	1	0.0	0.0	9200000	29700000	10	

4269 rows × 11 columns

In this code, df.iloc[:, :-1] means selecting all rows and all columns up to, but not including, the last column into DataFrame. x will contain all the features intend to use for analysis or modeling, excluding the target variable.

In [21]:
```
y=df[["loan_status"]]
y
```

Out[21]:

	loan_status
0	0.0
1	1.0
2	1.0
3	1.0
4	1.0
...	...
4264	1.0
4265	0.0
4266	1.0
4267	0.0
4268	0.0

4269 rows × 1 columns

In this code, df[["loan_status"]] is used to select the "loan_status" column and create a new DataFrame y. This DataFrame y will contain the target variable intend to predict or analyze machine learning task.

Apply Standard Scaler to Scale the data at one level

In [22]:
```
from sklearn.preprocessing import StandardScaler
```

In [23]:
```
from sklearn.model_selection import train_test_split
xtrain,xtest,ytrain,ytest=train_test_split(x,y,test_size=0.2,random_state=42)
```

#

x represents feature set.

y represents target variable.

The train_test_split function randomly splits data into training and testing sets. It takes several arguments:

x and y: The feature set and target variable to be split.

test_size: The proportion of the data to be used for testing (in this case, 20%).

random_state: A seed for the random number generator, which ensures reproducibility.

After running this code, you will have four new datasets:

xtrain: The training set of features.

xtest: The testing set of features.

ytrain: The corresponding training set of target values.

ytest: The corresponding testing set of target values.

These splits are commonly used in machine learning for training and evaluating models.

In [24]:
```
sc=StandardScaler()
xtrain=sc.fit_transform(xtrain)
xtest=sc.fit_transform(xtest)
```

first create a StandardScaler object named sc.

then use the fit_transform method to standardize the training data (xtrain).

This method computes the mean and standard deviation from the training data and scales it accordingly.

For the testing data (xtest), use the transform method to apply the same transformation that was learned from the training data. It's important not to fit the scaler again on the testing data,to apply the same scaling factors used for the training data to ensure consistency.

In [25]:
```
xtrain
```

Out[25]:
```
array([[ 1.51250774,  1.00263891, -1.01504731, ...,  2.04678575,
         0.07808278,  1.16041374],
       [-1.43500078,  1.00263891,  0.98517575, ...,  1.22311091,
         2.49843196,  0.88201987],
       [-0.84549907, -0.99736803, -1.01504731, ..., -0.8818359 ,
        -1.33923881, -1.31419838],
       ...,
       [ 0.92300603,  1.00263891,  0.98517575, ...,  1.29175048,
         1.47359943,  0.13963624],
       [-0.25599737,  1.00263891,  0.98517575, ..., -0.83607619,
         0.50327926,  1.4388076 ],
       [ 0.92300603, -0.99736803, -1.01504731, ..., -0.28695963,
         1.03750048, -0.10782497]])
```

In [26]:
```
xtest
```

Out[26]:
```
array([[ 1.34468623, -0.98835816, -0.97684871, ...,  1.11057559,
         0.09116237,  0.66704919],
       [-0.45337542, -0.98835816, -0.97684871, ...,  0.99842012,
         0.32938266,  0.48602276],
       [ 0.14597846, -0.98835816, -0.97684871, ...,  2.50130348,
         1.54317176,  0.45585169],
       ...,
       [ 1.34468623,  1.01177897,  1.02369997, ...,  0.46007384,
        -0.36259057,  0.81790455],
       [ 0.74533235,  1.01177897, -0.97684871, ..., -0.84092967,
        -1.02053232, -1.11304405],
       [-1.05272931,  1.01177897, -0.97684871, ...,  0.12360742,
        -0.58946704, -0.99235977]])
```

Train_Test_Split for separating data into training and testing phase

In [27]:
```
from sklearn.model_selection import train_test_split
```

It imported the train_test_split function from scikit-learn. This function is commonly used to split a dataset into training and testing subsets, which is a fundamental step in machine learning and model evaluation

In [28]:
```
xtrain,xtest,ytrain,ytest=train_test_split(x,y,test_size=0.2,random_state=1)
```

test_size is set to 0.2, indicating that 20% of the data will be used for testing, and the remaining 80% for training.

random_state is set to 1, which provides a seed for the random number generator. Setting this ensures the same split every time run the code, which is useful for reproducibility.

Build a Model by using LogisticRegression Algoritham

In [29]:
```
from sklearn.linear_model import LogisticRegression
```

Classification Report of model trained on Logistic Regression

In [30]:
```
from sklearn.metrics import classification_report
```

In [31]:
```
lr=LogisticRegression()
lr.fit(xtrain,ytrain)
ypred_train=lr.predict(xtrain)
ypred_test=lr.predict(xtest)
```

lr = LogisticRegression(): creating a logistic regression model and initializing it with the default hyperparameters. Depending on specific problem and dataset,to tune these hyperparameters for better performance.

lr.fit(xtrain, ytrain): fitting (training) the logistic regression model on training data. xtrain should contain the feature data, and ytrain should contain the corresponding target labels. This step is where the model learns the relationship between the features and the target variable.

ypred_train = lr.predict(xtrain):using the trained logistic regression model to make predictions on the training data (xtrain). This will predicted labels for the training data.

ypred_test = lr.predict(xtest):using the trained logistic regression model to make predictions on the test data (xtest).the predicted labels for the test data, use to evaluate the model's performance on unseen data.

After fitting the logistic regression model and obtaining predictions,proceed to evaluate the model's performance, typically using metrics like accuracy, precision, recall, F1-score, and confusion matrices, depending on the nature of classification problem. These metrics will help assess model is performing and whether any fine-tuning or adjustments are needed.

In [32]:
```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.72      0.95      0.82      2135
         1.0       0.80      0.38      0.51      1280

    accuracy                           0.73      3415
   macro avg       0.76      0.66      0.66      3415
weighted avg       0.75      0.73      0.70      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.71      0.93      0.80       521
         1.0       0.78      0.41      0.53       333

    accuracy                           0.72       854
   macro avg       0.74      0.67      0.67       854
weighted avg       0.74      0.72      0.70       854
```

By using Hyperparameter or Hypertunners

In [33]:
```
lr=LogisticRegression(solver="liblinear")
lr.fit(xtrain,ytrain)
ypred_train=lr.predict(xtrain)
ypred_test=lr.predict(xtest)
```

#

lr = LogisticRegression(solver="liblinear"): create a logistic regression model with the "liblinear" solver. This solver is suitable for small and medium-sized datasets, and it is especially effective for binary classification tasks.

lr.fit(xtrain, ytrain): train the logistic regression model using the training data, where xtrain contains the feature data, and ytrain contains the corresponding target labels. During this step, the model's parameters are optimized to fit the training data.

ypred_train = lr.predict(xtrain): use the trained logistic regression model to make predictions on the training data (xtrain). to evaluate how well the model fits the data it was trained on.

ypred_test = lr.predict(xtest): same trained model to make predictions on the test data (xtest). This is important for assessing how well the model generalizes to new, unseen data. Evaluating the model's performance on the test data is crucial for determining its effectiveness in real-world scenarios.

After obtaining predictions for both the training and test datasets,should proceed to evaluate the model's performance using appropriate classification metrics, as mentioned in my previous response. This will help understand how well your model is performing and whether any adjustments or further optimization are needed.

In [34]:
```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.63      1.00      0.77      2135
         1.0       0.40      0.00      0.01      1280

    accuracy                           0.62      3415
   macro avg       0.51      0.50      0.39      3415
weighted avg       0.54      0.62      0.48      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.61      0.99      0.76       521
         1.0       0.33      0.01      0.01       333

    accuracy                           0.61       854
   macro avg       0.47      0.50      0.38       854
weighted avg       0.50      0.61      0.47       854
```

In [35]:

```
lr=LogisticRegression(solver="sag")
lr.fit(xtrain,ytrain)
ypred_train=lr.predict(xtrain)
ypred_test=lr.predict(xtest)
```

lr = LogisticRegression(solver="sag"):create a logistic regression model with the "sag" solver. The "sag" solver is particularly useful for large datasets and can be more efficient than "liblinear" for such cases. It approximates the gradient using a stochastic approach.

lr.fit(xtrain, ytrain):train the logistic regression model using the training data (xtrain for features and ytrain for target labels). During this step, the model optimizes its parameters to fit the training data.

ypred_train = lr.predict(xtrain):trained logistic regression model to make predictions on the training data (xtrain).

ypred_test = lr.predict(xtest):trained model to make predictions on the test data (xtest). This is crucial for evaluating how well the model generalizes to new, unseen data.

After obtaining predictions for both the training and test datasets, it's essential to evaluate the model's performance. i can do this using various classification metrics, such as accuracy, precision, recall, F1-score, and confusion matrices, to determine how well model is performing and whether any further tuning or optimization is required.

In [36]:

```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.63      1.00      0.77      2135
         1.0       0.36      0.00      0.01      1280

    accuracy                           0.62      3415
   macro avg       0.49      0.50      0.39      3415
weighted avg       0.53      0.62      0.48      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.61      0.99      0.76       521
         1.0       0.33      0.01      0.01       333

    accuracy                           0.61       854
   macro avg       0.47      0.50      0.38       854
weighted avg       0.50      0.61      0.47       854
```

In [37]:

```
lr=LogisticRegression(solver="saga")
lr.fit(xtrain,ytrain)
ypred_train=lr.predict(xtrain)
ypred_test=lr.predict(xtest)
```

lr = LogisticRegression(solver="saga"):create a logistic regression model with the "saga" solver. The "saga" solver is a versatile option that can handle various types of datasets and problems.

lr.fit(xtrain, ytrain):train the logistic regression model using the training data (xtrain for features and ytrain for target labels).the model optimizes its parameters to fit the training data.

ypred_train = lr.predict(xtrain):trained logistic regression model to make predictions on the training data (xtrain). to evaluate how well the model fits the data it was trained on.

ypred_test = lr.predict(xtest):use the trained model to make predictions on the test data (xtest). This is essential for evaluating how well the model generalizes to new, unseen data.

After obtaining predictions for both the training and test datasets, it's important to evaluate the model's performance. various classification metrics, such as accuracy, precision, recall, F1-score, and confusion matrices, to determine how well your model is performing and whether any further tuning or optimization is needed.

In [38]:

```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.63      1.00      0.77      2135
         1.0       0.36      0.00      0.01      1280

    accuracy                           0.62      3415
   macro avg       0.49      0.50      0.39      3415
weighted avg       0.53      0.62      0.48      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.61      0.99      0.76       521
         1.0       0.33      0.01      0.01       333

    accuracy                           0.61       854
   macro avg       0.47      0.50      0.38       854
weighted avg       0.50      0.61      0.47       854
```

Build a model by using KNN Algoritham

In [39]:
```
from sklearn.neighbors import KNeighborsClassifier
```

In [40]:
```
knn=KNeighborsClassifier(n_neighbors=5)
knn.fit(xtrain,ytrain)
ypred_train=knn.predict(xtrain)
ypred_test=knn.predict(xtest)
```

knn = KNeighborsClassifier(n_neighbors=5):KNN classifier by initializing it with n_neighbors=5.the KNN algorithm will consider the 5 nearest neighbors when making predictions. adjust the n_neighbors parameter based on problem's requirements.

knn.fit(xtrain, ytrain): KNN classifier using the training data. xtrain should contain the feature data, and ytrain should contain the corresponding target labels. During this step, the KNN model learns to classify data points based on the distances to their nearest neighbors.

ypred_train = knn.predict(xtrain): trained KNN classifier to make predictions on the training data (xtrain).well the model fits the data it was trained on.

ypred_test = knn.predict(xtest): use the trained model to make predictions on the test data (xtest). This is crucial for evaluating how well the model generalizes to new, unseen data.

After obtaining predictions for both the training and test datasets, it's important to evaluate the KNN model's performance. use various classification metrics, such as accuracy, precision, recall, F1-score, and confusion matrices, to assess how well your model is performing.

In [41]:
```
from sklearn.metrics import classification_report
```

In [42]:
```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.74      0.86      0.79      2135
         1.0       0.68      0.50      0.58      1280

    accuracy                           0.72      3415
   macro avg       0.71      0.68      0.69      3415
weighted avg       0.72      0.72      0.71      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.62      0.74      0.67       521
         1.0       0.41      0.28      0.33       333

    accuracy                           0.56       854
   macro avg       0.51      0.51      0.50       854
weighted avg       0.53      0.56      0.54       854
```

Build a model on SVM Algoritham

In [43]:
```
from sklearn.svm import SVC
```

SVC (Support Vector Classification) class from scikit-learn. The SVC is a powerful algorithm commonly used for classification tasks and it's based on the concept of support vectors within a high-dimensional space.

In [44]:
```
svm=SVC()
```

In [45]:
```
def mymodel(model):
    model.fit(xtrain,ytrain)
    ypred=model.predict(xtest)
    print(classification_report(ytest,ypred))
    return model
```

Defined a function mymodel that takes a model as an argument, fits the model on the training data, makes predictions on the testing data, and prints the classification report.

In [46]:
```
from sklearn.pipeline import Pipeline
```

We using pipeline beacause data should be process smoothly.

In [47]:
```
pipe=Pipeline(steps=[('scaler',StandardScaler()),('svm',SVC())])
```

In pipe we write steps so that steps will be executed one by one

In [48]:
```
pipe.fit(xtrain,ytrain)
ypred=pipe.predict(xtest)
```

Fitting the entire pipeline (pipe) on the training data (xtrain, ytrain) and then making predictions on the testing data (xtest).

In [49]:
```
mymodel(svm)
```

```
              precision    recall  f1-score   support

         0.0       0.61      1.00      0.76       521
         1.0       0.00      0.00      0.00       333

    accuracy                           0.61       854
   macro avg       0.31      0.50      0.38       854
weighted avg       0.37      0.61      0.46       854
```

Out[49]:

```
▾ SVC
SVC()
```

In [50]:

```
from sklearn.tree import DecisionTreeClassifier
```

In [51]:

```
dt=DecisionTreeClassifier()
dt.fit(xtrain,ytrain)
ytrain_pred=dt.predict(xtrain)
ytest_pred=dt.predict(xtest)
```


dt = DecisionTreeClassifier(): create a decision tree classifier by initializing it with the default hyperparameters. A decision tree is a simple but powerful machine learning model used for classification and regression tasks.

dt.fit(xtrain, ytrain): train the decision tree classifier using the training data. xtrain should contain the feature data, and ytrain should contain the corresponding target labels. During this step, the decision tree model builds a tree structure that helps it make predictions based on the features.

ytrain_pred = dt.predict(xtrain): use the trained decision tree classifier to make predictions on the training data (xtrain). how well the model fits the data it was trained on.

ytest_pred = dt.predict(xtest):the trained model to make predictions on the test data (xtest). This is important for evaluating how well the model generalizes to new, unseen data.

Evaluate the model by using Classification_report

In [52]:

```
from sklearn.metrics import classification_report,accuracy_score
```

In [53]:

```
print("Train Data")
print(classification_report(ytrain,ytrain_pred))
print("Test Data")
print(classification_report(ytest,ytest_pred))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      2135
         1.0       1.00      1.00      1.00      1280

    accuracy                           1.00      3415
   macro avg       1.00      1.00      1.00      3415
weighted avg       1.00      1.00      1.00      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.99      0.98      0.99       521
         1.0       0.97      0.98      0.98       333

    accuracy                           0.98       854
   macro avg       0.98      0.98      0.98       854
weighted avg       0.98      0.98      0.98       854
```

Performing Hypertunning on model

Hyper parameters with impurity checking gini

In [54]:
```
for i in range(1,31):
    dt1=DecisionTreeClassifier(max_depth=i)
    dt1.fit(xtrain,ytrain)
    ypred=dt1.predict(xtest)
    ac=accuracy_score(ytest,ypred)
    print(f"Max Depth:{i} accuracy): {ac}")
```

```
Max Depth:1 accuracy): 0.9484777517564403
Max Depth:2 accuracy): 0.9601873536299765
Max Depth:3 accuracy): 0.9637002341920374
Max Depth:4 accuracy): 0.9695550351288056
Max Depth:5 accuracy): 0.9648711943793911
Max Depth:6 accuracy): 0.9672131147540983
Max Depth:7 accuracy): 0.9707259953161592
Max Depth:8 accuracy): 0.9695550351288056
Max Depth:9 accuracy): 0.9730679156908665
Max Depth:10 accuracy): 0.9742388758782201
Max Depth:11 accuracy): 0.9800936768149883
Max Depth:12 accuracy): 0.9800936768149883
Max Depth:13 accuracy): 0.9800936768149883
Max Depth:14 accuracy): 0.9824355971896955
Max Depth:15 accuracy): 0.9812646370023419
Max Depth:16 accuracy): 0.9824355971896955
Max Depth:17 accuracy): 0.9800936768149883
Max Depth:18 accuracy): 0.9812646370023419
Max Depth:19 accuracy): 0.9800936768149883
Max Depth:20 accuracy): 0.9824355971896955
Max Depth:21 accuracy): 0.9824355971896955
Max Depth:22 accuracy): 0.9824355971896955
Max Depth:23 accuracy): 0.9836065573770492
Max Depth:24 accuracy): 0.9836065573770492
Max Depth:25 accuracy): 0.9824355971896955
Max Depth:26 accuracy): 0.9836065573770492
Max Depth:27 accuracy): 0.9800936768149883
Max Depth:28 accuracy): 0.9824355971896955
Max Depth:29 accuracy): 0.9836065573770492
Max Depth:30 accuracy): 0.9812646370023419
```

In [55]:
```
dt1=DecisionTreeClassifier(max_depth=5)
mymodel(dt1)
```

```
              precision    recall  f1-score   support

         0.0       1.00      0.95      0.97       521
         1.0       0.92      0.99      0.96       333

    accuracy                           0.97       854
   macro avg       0.96      0.97      0.96       854
weighted avg       0.97      0.97      0.97       854
```

Out[55]:
```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=5)
```

Lets checking overfitting scenario

In [56]:
```
dt1.score(xtrain,ytrain)
```

Out[56]:
```
0.9742313323572475
```

Hyper parameters with impurity checking min_sample_split

In [57]:
```
for i in range(5,50):
    dt3=DecisionTreeClassifier(min_samples_split=i)
    dt3.fit(xtrain,ytrain)
    ypred=dt3.predict(xtest)
    ac=accuracy_score(ytest,ypred)
    print(f"min sample split:{i} accuracy): {ac}")
```

```
min sample split:5 accuracy): 0.9836065573770492
min sample split:6 accuracy): 0.9847775175644028
min sample split:7 accuracy): 0.9847775175644028
min sample split:8 accuracy): 0.9859484777517564
min sample split:9 accuracy): 0.9847775175644028
min sample split:10 accuracy): 0.9859484777517564
min sample split:11 accuracy): 0.9859484777517564
min sample split:12 accuracy): 0.9859484777517564
min sample split:13 accuracy): 0.9859484777517564
min sample split:14 accuracy): 0.9859484777517564
min sample split:15 accuracy): 0.9859484777517564
min sample split:16 accuracy): 0.9859484777517564
min sample split:17 accuracy): 0.9859484777517564
min sample split:18 accuracy): 0.9859484777517564
min sample split:19 accuracy): 0.9859484777517564
min sample split:20 accuracy): 0.9859484777517564
min sample split:21 accuracy): 0.9859484777517564
min sample split:22 accuracy): 0.9859484777517564
min sample split:23 accuracy): 0.9859484777517564
min sample split:24 accuracy): 0.9859484777517564
min sample split:25 accuracy): 0.9859484777517564
min sample split:26 accuracy): 0.9859484777517564
min sample split:27 accuracy): 0.9859484777517564
min sample split:28 accuracy): 0.9859484777517564
min sample split:29 accuracy): 0.9859484777517564
min sample split:30 accuracy): 0.9859484777517564
min sample split:31 accuracy): 0.9859484777517564
min sample split:32 accuracy): 0.9859484777517564
min sample split:33 accuracy): 0.9859484777517564
min sample split:34 accuracy): 0.9859484777517564
min sample split:35 accuracy): 0.9859484777517564
min sample split:36 accuracy): 0.9859484777517564
min sample split:37 accuracy): 0.9859484777517564
min sample split:38 accuracy): 0.9859484777517564
min sample split:39 accuracy): 0.9859484777517564
min sample split:40 accuracy): 0.9859484777517564
min sample split:41 accuracy): 0.9859484777517564
min sample split:42 accuracy): 0.9859484777517564
min sample split:43 accuracy): 0.9859484777517564
min sample split:44 accuracy): 0.9859484777517564
min sample split:45 accuracy): 0.9859484777517564
min sample split:46 accuracy): 0.9859484777517564
min sample split:47 accuracy): 0.9859484777517564
min sample split:48 accuracy): 0.9859484777517564
min sample split:49 accuracy): 0.9859484777517564
```

In [ ]:

In [58]:

```
dt3=DecisionTreeClassifier(min_samples_split=9)
mymodel(dt3)
```

```
              precision    recall  f1-score   support

         0.0       0.99      0.99      0.99       521
         1.0       0.98      0.98      0.98       333

    accuracy                           0.99       854
   macro avg       0.99      0.98      0.99       854
weighted avg       0.99      0.99      0.99       854
```

Out[58]:

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(min_samples_split=9)
```

Lets checking overfitting scenario

In [59]:

```
dt3.score(xtrain,ytrain)
```

Out[59]:

```
0.9944363103953148
```

Hyper parameters with impurity checking min_sample_leaf

In [60]:

```
for i in range(1,51):
    dt4=DecisionTreeClassifier(min_samples_leaf=i)
    dt4.fit(xtrain,ytrain)
    ypred=dt4.predict(xtest)
    ac=accuracy_score(ytest,ypred)
    print(f"min sample leaf:{i} accuracy): {ac}")
```

```
min sample leaf:1 accuracy): 0.9824355971896955
min sample leaf:2 accuracy): 0.9812646370023419
min sample leaf:3 accuracy): 0.9836065573770492
min sample leaf:4 accuracy): 0.9871194379391101
min sample leaf:5 accuracy): 0.9882903981264637
min sample leaf:6 accuracy): 0.9859484777517564
min sample leaf:7 accuracy): 0.9847775175644028
min sample leaf:8 accuracy): 0.9847775175644028
min sample leaf:9 accuracy): 0.9812646370023419
min sample leaf:10 accuracy): 0.9824355971896955
min sample leaf:11 accuracy): 0.9824355971896955
min sample leaf:12 accuracy): 0.977751756440281
min sample leaf:13 accuracy): 0.977751756440281
min sample leaf:14 accuracy): 0.977751756440281
min sample leaf:15 accuracy): 0.9800936768149883
min sample leaf:16 accuracy): 0.9800936768149883
min sample leaf:17 accuracy): 0.9800936768149883
min sample leaf:18 accuracy): 0.9812646370023419
min sample leaf:19 accuracy): 0.9800936768149883
min sample leaf:20 accuracy): 0.977751756440281
min sample leaf:21 accuracy): 0.977751756440281
min sample leaf:22 accuracy): 0.977751756440281
min sample leaf:23 accuracy): 0.977751756440281
min sample leaf:24 accuracy): 0.977751756440281
min sample leaf:25 accuracy): 0.9742388758782201
min sample leaf:26 accuracy): 0.9718969555035128
min sample leaf:27 accuracy): 0.9718969555035128
min sample leaf:28 accuracy): 0.9718969555035128
min sample leaf:29 accuracy): 0.9730679156908665
min sample leaf:30 accuracy): 0.9730679156908665
min sample leaf:31 accuracy): 0.9754098360655737
min sample leaf:32 accuracy): 0.9754098360655737
min sample leaf:33 accuracy): 0.9754098360655737
min sample leaf:34 accuracy): 0.9742388758782201
min sample leaf:35 accuracy): 0.9742388758782201
min sample leaf:36 accuracy): 0.9742388758782201
min sample leaf:37 accuracy): 0.9742388758782201
min sample leaf:38 accuracy): 0.9742388758782201
min sample leaf:39 accuracy): 0.9742388758782201
min sample leaf:40 accuracy): 0.9765807962529274
min sample leaf:41 accuracy): 0.9765807962529274
min sample leaf:42 accuracy): 0.9765807962529274
min sample leaf:43 accuracy): 0.9765807962529274
min sample leaf:44 accuracy): 0.9765807962529274
min sample leaf:45 accuracy): 0.9754098360655737
min sample leaf:46 accuracy): 0.9730679156908665
min sample leaf:47 accuracy): 0.9730679156908665
min sample leaf:48 accuracy): 0.9730679156908665
min sample leaf:49 accuracy): 0.9718969555035128
min sample leaf:50 accuracy): 0.9718969555035128
```

Building model with final value of max depth as 2

In [61]:
```
dt4=DecisionTreeClassifier(min_samples_split=2)
mymodel(dt4)
```

```
              precision    recall  f1-score   support

         0.0       0.99      0.98      0.99       521
         1.0       0.98      0.98      0.98       333

    accuracy                           0.98       854
   macro avg       0.98      0.98      0.98       854
weighted avg       0.98      0.98      0.98       854
```

Out[61]:

```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

Lets checking overfitting scenario

In [62]:

```
dt4.score(xtrain,ytrain)
```

Out[62]:

```
1.0
```

Voting

In [63]:

```
from sklearn.ensemble import VotingClassifier
```

In [64]:

```
model=[]
accuracy=[]
model.append(("Logistic Regression",LogisticRegression()))
model.append(("Decision Tree",DecisionTreeClassifier()))
```

In [65]:

```
model
```

Out[65]:

```
[('Logistic Regression', LogisticRegression()),
 ('Decision Tree', DecisionTreeClassifier())]
```

In [66]:

```
vc=VotingClassifier(estimators=model)
vc.fit(xtrain,ytrain)
ypred_train=vc.predict(xtrain)
ypred_test=vc.predict(xtest)
```

# #

vc = VotingClassifier(estimators=model):create a VotingClassifier by initializing it with the list of (name, estimator) pairs.

vc.fit(xtrain, ytrain):train the VotingClassifier using the training data (xtrain for features and ytrain for target labels). The VotingClassifier combines the predictions of the individual models to make an ensemble prediction.

ypred_train = vc.predict(xtrain): use the trained VotingClassifier to make predictions on the training data (xtrain). to assess how well the ensemble model fits the data it was trained on.

ypred_test = vc.predict(xtest): also use the trained model to make predictions on the test data (xtest). This is important for evaluating how well the ensemble model generalizes to new, unseen data.

After obtaining predictions for both the training and test datasets, it's important to evaluate the VotingClassifier's performance. use various classification metrics, such as accuracy, precision, recall, F1-score, and confusion matrices, to assess how well your ensemble model is performing.

The VotingClassifier is a useful tool for combining the predictions of multiple models and can be particularly effective when the individual models have different strengths or weaknesses.

Classification Report

In [67]:
```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.73      1.00      0.84      2135
         1.0       1.00      0.38      0.55      1280

    accuracy                           0.77      3415
   macro avg       0.86      0.69      0.69      3415
weighted avg       0.83      0.77      0.73      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.72      0.99      0.84       521
         1.0       0.98      0.41      0.57       333

    accuracy                           0.76       854
   macro avg       0.85      0.70      0.71       854
weighted avg       0.82      0.76      0.73       854
```

Bagging

In [68]:
```
from sklearn.ensemble import BaggingClassifier
```

In [69]:
```
bg=BaggingClassifier(LogisticRegression())
bg.fit(xtrain,ytrain)
ypred_train=bg.predict(xtrain)
ypred_test=bg.predict(xtest)
```

bg = BaggingClassifier(LogisticRegression()): create a BaggingClassifier by initializing it with a base estimator, which is the LogisticRegression in this case. The BaggingClassifier builds an ensemble by training multiple instances of the base estimator on subsets of the training data.

bg.fit(xtrain, ytrain): train the BaggingClassifier using the training data. xtrain should contain the feature data, and ytrain should contain the corresponding target labels. The BaggingClassifier will train multiple instances of the LogisticRegression classifier on different subsets of the training data.

ypred_train = bg.predict(xtrain):use the trained BaggingClassifier to make predictions on the training data (xtrain). assess how well the ensemble model fits the data it was trained on.

ypred_test = bg.predict(xtest):use the trained BaggingClassifier to make predictions on the test data (xtest). This is important for evaluating how well the ensemble model generalizes to new, unseen data.

Classification Report

In [70]:
```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.69      0.97      0.81      2135
         1.0       0.83      0.28      0.42      1280

    accuracy                           0.71      3415
   macro avg       0.76      0.62      0.61      3415
weighted avg       0.74      0.71      0.66      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.69      0.96      0.81       521
         1.0       0.85      0.33      0.48       333

    accuracy                           0.72       854
   macro avg       0.77      0.65      0.64       854
weighted avg       0.75      0.72      0.68       854
```

In [71]:
```
bg=BaggingClassifier(KNeighborsClassifier())
bg.fit(xtrain,ytrain)
ypred_train=bg.predict(xtrain)
ypred_test=bg.predict(xtest)
```

bg = BaggingClassifier(KNeighborsClassifier()):create a BaggingClassifier by initializing it with a base estimator, which is the KNeighborsClassifier in this case. The BaggingClassifier builds an ensemble by training multiple instances of the base estimator on subsets of the training data.

bg.fit(xtrain, ytrain): train the BaggingClassifier using the training data. xtrain should contain the feature data, and ytrain should contain the corresponding target labels. The BaggingClassifier will train multiple instances of the KNeighborsClassifier on different subsets of the training data.

ypred_train = bg.predict(xtrain):use the trained BaggingClassifier to make predictions on the training data (xtrain). assess how well the ensemble model fits the data it was trained on.

ypred_test = bg.predict(xtest): use the trained BaggingClassifier to make predictions on the test data (xtest). This is important for evaluating how well the ensemble model generalizes to new, unseen data

Classification Report

In [72]:
```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       0.74      0.87      0.80      2135
         1.0       0.69      0.48      0.57      1280

    accuracy                           0.72      3415
   macro avg       0.71      0.67      0.68      3415
weighted avg       0.72      0.72      0.71      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.61      0.76      0.68       521
         1.0       0.40      0.25      0.31       333

    accuracy                           0.56       854
   macro avg       0.51      0.50      0.49       854
weighted avg       0.53      0.56      0.53       854
```

Random Forest

In [73]:
```
from sklearn.ensemble import RandomForestClassifier

rf=RandomForestClassifier()
rf.fit(xtrain,ytrain)
ypred_train=rf.predict(xtrain)
ypred_test=rf.predict(xtest)
```

#

rf = RandomForestClassifier(): create a random forest classifier by initializing it with the default hyperparameters. A random forest is an ensemble learning method that combines multiple decision trees to make predictions. We can later adjust the hyperparameters to fine-tune the model's performance.

rf.fit(xtrain, ytrain): train the random forest classifier using the training data. xtrain should contain the feature data, and ytrain should contain the corresponding target labels. During this step, the random forest model builds a collection of decision trees based on subsets of the training data and features.

ypred_train = rf.predict(xtrain): use the trained random forest classifier to make predictions on the training data (xtrain). assess how well the model fits the data it was trained on.

ypred_test = rf.predict(xtest):use the trained model to make predictions on the test data (xtest). This is important for evaluating how well the model generalizes to new, unseen data.

Classification Report

In [74]:
```
print("Train Data")
print(classification_report(ytrain,ypred_train))
print("Test Data")
print(classification_report(ytest,ypred_test))
```

```
Train Data
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      2135
         1.0       1.00      1.00      1.00      1280

    accuracy                           1.00      3415
   macro avg       1.00      1.00      1.00      3415
weighted avg       1.00      1.00      1.00      3415

Test Data
              precision    recall  f1-score   support

         0.0       0.98      0.99      0.99       521
         1.0       0.98      0.97      0.98       333

    accuracy                           0.98       854
   macro avg       0.98      0.98      0.98       854
weighted avg       0.98      0.98      0.98       854
```

The classification_report provides metrics like precision, recall, f1-score, and support for each class. It's a useful way to evaluate the performance of your classifier on different classes.

According to classification Report accuracy of training data is 100% and testing data is 98% .

In [ ]:
```

```