TREELEAF TECHNOLOGIES PVT. LTD.

MACHINE LEARNING INTERNSHIP

QUALIFICATION TASK

Submitted to: Treeleaf Technologies

Submitted by: Baibhab Bhattarai

Date:16th August 2023

Contents

1. Analysis Report: Preprocessing and Predictive Modeling for Bank Loan Approval

1.1. Approach Overview

The analysis concentrates on preprocessing and predictive modeling for loan approval. The dataset is loaded from and Excel file and processed to handle missing values, categorical features, and outliers. A Logistic Regression model is trained to predict Personal Loan approval based on various features. The model's performance is evaluated using accuracy, confusion matrix, classification report, and ROC curve. Finally, the trained model is serialized using Pickle for future use.

2. Key Findings

2.1. Data Preprocessing

2.1.1. The original dataset is loaded and examined for its shape, data types, and missing values.

In [2]:
```python
import numpy as np
import pandas as pd
```

In [3]:
```python
file_path = "New Bank_loan_data.xlsx"
df_original = pd.read_excel(file_path)
df_original.head()
```

Out[3]:

	ID	Age	Gender	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Home Ownership	Personal Loan	Securities Account	CD Account	Online	CreditCard
0	1	25	M	1	49.0	91107	4	1.6	1	0	Home Owner	0	1	0	0.0	0
1	2	45	M	19	34.0	90089	3	1.5	1	0	Rent	0	1	0	0.0	0
2	3	39	M	15	11.0	94720	1	1.0	1	0	Rent	0	0	0	0.0	0
3	4	35	M	9	100.0	94112	1	2.7	2	0	Rent	0	0	0	0.0	0
4	5	35	M	8	45.0	91330	4	1.0	2	0	Rent	0	0	0	0.0	1

In [3]:
```python
df_original.shape
```

Out[3]:
```
(5000, 16)
```

In [4]:
```python
df_original.dtypes
```

Out[4]:
```
ID                      int64
Age                     int64
Gender                 object
Experience              int64
Income                float64
ZIP Code                int64
Family                  int64
CCAvg                 float64
Education               int64
Mortgage                int64
Home Ownership         object
Personal Loan          object
Securities Account      int64
CD Account              int64
Online                float64
CreditCard              int64
dtype: object
```

In [5]:
```python
df_original.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 16 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   ID                  5000 non-null   int64  
 1   Age                 5000 non-null   int64  
 2   Gender              3404 non-null   object 
 3   Experience          5000 non-null   int64  
 4   Income              4933 non-null   float64
 5   ZIP Code            5000 non-null   int64  
 6   Family              5000 non-null   int64  
 7   CCAvg               5000 non-null   float64
 8   Education           5000 non-null   int64  
 9   Mortgage            5000 non-null   int64  
 10  Home Ownership      3811 non-null   object 
 11  Personal Loan       5000 non-null   object 
 12  Securities Account  5000 non-null   int64  
 13  CD Account          5000 non-null   int64  
 14  Online              4960 non-null   float64
 15  CreditCard          5000 non-null   int64  
dtypes: float64(3), int64(10), object(3)
memory usage: 625.1+ KB
```

In [6]:
```
df_original.describe(include = "all")
```

Out[6]:

	ID	Age	Gender	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Home Ownership	Personal Loan
count	5000.000000	5000.000000	3404	5000.000000	4933.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	3811	5000.0
unique	NaN	NaN	5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	3	3.0
top	NaN	NaN	M	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Home Mortage	0.0
freq	NaN	NaN	1677	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1705	4520.0
mean	2500.500000	46.008200	NaN	20.104600	72.758159	93152.503000	2.396400	1.937913	1.880600	56.498800	NaN	NaN
std	1443.520003	25.444898	NaN	11.467954	45.425519	2121.852197	1.147663	1.747666	0.839812	101.713802	NaN	NaN
min	1.000000	0.000000	NaN	-3.000000	8.000000	9307.000000	1.000000	0.000000	1.000000	0.000000	NaN	NaN
25%	1250.750000	35.000000	NaN	10.000000	39.000000	91911.000000	1.000000	0.700000	1.000000	0.000000	NaN	NaN
50%	2500.500000	45.000000	NaN	20.000000	63.000000	93437.000000	2.000000	1.500000	2.000000	0.000000	NaN	NaN
75%	3750.250000	55.000000	NaN	30.000000	95.000000	94608.000000	3.000000	2.500000	3.000000	101.000000	NaN	NaN
max	5000.000000	978.000000	NaN	43.000000	224.000000	96651.000000	4.000000	10.000000	3.000000	635.000000	NaN	NaN

In [7]:
```
df_original.isnull().sum()
```

Out[7]:
```
ID                       0
Age                      0
Gender                1596
Experience               0
Income                  67
ZIP Code                 0
Family                   0
CCAvg                    0
Education                0
Mortgage                 0
Home Ownership        1189
Personal Loan            0
Securities Account       0
CD Account               0
Online                  40
CreditCard               0
dtype: int64
```

The code above shows total number of missing values in each column.

2.1.2. Categorical columns ‘Gender’, ‘Home Ownership’, are preprocessed using one-hot encoding.

There were 6 different unique values in ‘Gender’ column including null values. The column also includes values like “#” and “-“. Those values were replaced by np.nan.

The pd.get_dummies() function is used to convert categorical variables into a set of binary columns (dummy variables) in a DataFrame. In this case, we're applying it to the "Gender" column of the df_preprocessed DataFrame.

pd.get_dummies(df_preprocessed['Gender']) takes the "Gender" column of the DataFrame and converts it into a set of dummy variables. Each distinct value in the "Gender" column (e.g., Male, Female, Others) will become a new binary column.

If the original "Gender" column had three unique values (Male, Female, Others), the resulting gender_dummies DataFrame would have three columns: one for Male, one for Female, and one for Others. The values in these columns will be 1 if the corresponding row's gender matches the column value and 0 otherwise.

The two DataFrame are concatenated horizontally (along columns) in the further process. The ‘Gender’ column is dropped from the DataFrame at the final step of this process.

```
In [8]: # different unique values in Gender column
        df_preprocessed = df_original.copy()
        df_preprocessed['Gender'].unique()
```

```
Out[8]: array(['M', 'F', 'O', nan, '#', '-'], dtype=object)
```

```
In [9]: # Since, "#" and "-" are random values, we are replacing "#" and "-" data in Gender column with NaN
        df_preprocessed['Gender'] = df_preprocessed['Gender'].replace(['#', '-'], np.nan)
```

```
In [10]: # handling categorical value "Missing" in gender column witn pandas.get_dummies method
         # it converts categorical values into dummy variables
         gender_dummies = pd.get_dummies(df_preprocessed['Gender'])
         df_preprocessed = pd.concat([df_preprocessed, gender_dummies], axis = 1)
         df_preprocessed
```

Out[10]:

	ID	Age	Gender	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Home Ownership	Personal Loan	Securities Account	CD Account	Online	CreditCard	F
0	1	25	M	1	49.0	91107	4	1.6	1	0	Home Owner	0	1	0	0.0	0	0
1	2	45	M	19	34.0	90089	3	1.5	1	0	Rent	0	1	0	0.0	0	0
2	3	39	M	15	11.0	94720	1	1.0	1	0	Rent	0	0	0	0.0	0	0
3	4	35	M	9	100.0	94112	1	2.7	2	0	Rent	0	0	0	0.0	0	0
4	5	35	M	8	45.0	91330	4	1.0	2	0	Rent	0	0	0	0.0	1	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4995	4996	29	NaN	3	40.0	92697	1	1.9	3	0	NaN	0	0	0	1.0	0	0
4996	4997	30	NaN	4	15.0	92037	4	0.4	1	85	NaN	0	0	0	1.0	0	0
4997	4998	63	NaN	39	24.0	93023	2	0.3	3	0	NaN	0	0	0	0.0	0	0
4998	4999	65	NaN	40	49.0	90034	3	0.5	2	0	NaN	0	0	0	1.0	0	0
4999	5000	28	NaN	4	83.0	92612	3	0.8	1	0	NaN	0	0	0	1.0	1	0

5000 rows × 19 columns

```
In [13]: df_preprocessed.drop('Gender', axis=1, inplace=True)
```

The same procedure is repeated for the 'Home Ownership' column. The steps done is presented in the images below:

```
In [14]: df_preprocessed['Home Ownership'].unique()
```

```
Out[14]: array(['Home Owner', 'Rent', 'Home Mortage', nan], dtype=object)
```

```
In [12]: home_ownership_dummies = pd.get_dummies(df_preprocessed['Home Ownership'])
         df_preprocessed = pd.concat([df_preprocessed, home_ownership_dummies], axis=1)
```

```
In [16]: df_preprocessed.drop('Home Ownership', axis=1, inplace=True)
```

```
In [17]: df_preprocessed
```

Out[17]:

	ID	Age	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Personal Loan	Securities Account	CD Account	Online	CreditCard	F	M	O	Home Mortage	Ho Ow
0	1	25	1	49.0	91107	4	1.6	1	0	0	1	0	0.0	0	0	1	0	0	
1	2	45	19	34.0	90089	3	1.5	1	0	0	1	0	0.0	0	0	1	0	0	
2	3	39	15	11.0	94720	1	1.0	1	0	0	0	0	0.0	0	0	1	0	0	
3	4	35	9	100.0	94112	1	2.7	2	0	0	0	0	0.0	0	0	1	0	0	
4	5	35	8	45.0	91330	4	1.0	2	0	0	0	0	0.0	1	0	1	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
4995	4996	29	3	40.0	92697	1	1.9	3	0	0	0	0	1.0	0	0	0	0	0	
4996	4997	30	4	15.0	92037	4	0.4	1	85	0	0	0	1.0	0	0	0	0	0	
4997	4998	63	39	24.0	93023	2	0.3	3	0	0	0	0	0.0	0	0	0	0	0	
4998	4999	65	40	49.0	90034	3	0.5	2	0	0	0	0	1.0	0	0	0	0	0	
4999	5000	28	4	83.0	92612	3	0.8	1	0	0	0	0	1.0	1	0	0	0	0	

5000 rows × 20 columns

2.1.3. Rows with missing values in ‘Personal Loan’ column are dropped.

The rows with a value of ‘ ‘ in ‘Personal Loan’ column are dropped because it is irrelevant to the target variable in our dataset.

```
In [18]: df_preprocessed['Personal Loan'].unique()

Out[18]: array([0, 1, ' '], dtype=object)

In [19]: df_preprocessed['Personal Loan'].value_counts()

Out[19]: 0    4520
         1     479
                 1
         Name: Personal Loan, dtype: int64

In [20]: # we can drop the row where "Personal Loan" == " " since there is only 1 such row
         df_preprocessed = df_preprocessed[df_preprocessed['Personal Loan'] != ' ']
         df_preprocessed.shape

Out[20]: (4999, 20)
```

2.1.4. Outliers in the ‘Age’ column are detected using the IQR method and capped

Age outliers are handled in the 'Age' column by capping values above a specific range, which is based on the pragmatic insight that "age cannot be more than 100; it's unrealistic."

Age is a parameter that naturally has upper and lower bounds in every real-world dataset. Humans cannot live longer than a certain age, which is typically seen as being approximately 80 years. As a result, numbers in the 'Age' column that are higher than this limit are probably inaccurate or the result of data entry mistakes.

The study effectively solves this implausible scenario by utilizing the Interquartile Range (IQR) proximity approach to identify and cap outliers in the 'Age' column.

In [29]:
```
# Since there are outlies in the "Age" column we are going to remove those outliers by using
# IQR Proximity Technique
percentile25 = df_preprocessed['Age'].quantile(0.25)
percentile75 = df_preprocessed['Age'].quantile(0.75)
```

In [30]:
```
# Calculating Inter Quartile Range (IQR)
iqr = percentile75 - percentile25
iqr
```

Out[30]:
```
20.0
```

In [31]:
```
upper_limit = percentile75 + 1.5 * iqr
lower_limit = percentile25 -1.5 *iqr
print("Upper Limit", upper_limit)
print("Lower Limit", lower_limit)
```

```
Upper Limit 85.0
Lower Limit 5.0
```

In [32]:
```
df_preprocessed[df_preprocessed['Age'] > upper_limit]
```

Out[32]:

	ID	Age	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Personal Loan	Securities Account	CD Account	Online	CreditCard	F	M	O	Home Mortage	Ho Ow
47	48	97	12	194.0	91380	4	0.2	3	211	1	1	1	1.0	1	0	1	0	0	
53	54	567	26	190.0	90245	3	2.1	3	240	1	0	0	1.0	0	0	1	0	0	
131	132	122	34	149.0	93720	4	7.2	2	0	1	0	1	1.0	1	0	1	0	0	
765	766	978	21	109.0	95822	4	1.8	1	0	1	0	0	0.0	0	0	1	0	0	
2005	2006	786	23	170.0	90254	2	6.5	2	0	1	0	1	1.0	1	1	0	0	1	
2101	2102	600	5	203.0	95032	1	10.0	3	0	1	0	0	0.0	0	0	1	0	1	
2541	2542	797	8	171.0	90212	2	2.2	2	569	1	0	0	1.0	0	0	1	0	0	

In [33]:
```
df_preprocessed[df_preprocessed['Age'] < lower_limit]
```

Out[33]:

	ID	Age	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Personal Loan	Securities Account	CD Account	Online	CreditCard	F	M	O	Home Mortage	Ho Ow
78	79	0	30	63.0	93305	2	2.6	3	0	1	0	0	0.0	0	0	1	0	0	
1583	1584	0	36	184.0	92028	4	2.3	2	342	1	0	1	1.0	1	1	0	0	0	
1666	1667	2	25	190.0	95138	2	4.2	2	0	1	0	0	1.0	0	1	0	0	0	
1798	1799	4	20	185.0	94086	3	2.7	1	0	1	0	0	1.0	0	0	1	0	1	

In [34]:
```
df_no_outliers = df_preprocessed.copy()

# Capping outliers in Age column where upper range outliers are capped with upper_limit value and
# lower range outliers are capped with lower_limit value
df_no_outliers['Age'] = np.where(df_no_outliers['Age'] > upper_limit, upper_limit, np.where(df_no_outliers['Age'] < lower_limit,
```

2.1.5. Irrelevant columns (‘ID’, ‘ZIP Code’) are dropped to create the final preprocessed dataset.

Since ‘ID’, ‘ZIP Code’ are irrelevant for the prediction, they’re dropped from the final dataset.

In [36]:
```
# Final dataset for prediction, dropping unneccesary columns that are irrelevant for the prediction
df_final = df_no_outliers.drop(['ID', 'ZIP Code'], axis=1)
df_final
```

Out[36]:

	Age	Experience	Income	Family	CCAvg	Education	Mortgage	Personal Loan	Securities Account	CD Account	Online	CreditCard	F	M	O	Home Mortage	Home Owner	Rent
0	25.0	1	49.0	4	1.6	1	0	0	1	0	0.0	0	0	1	0	0	1	0
1	45.0	19	34.0	3	1.5	1	0	0	1	0	0.0	0	0	1	0	0	0	1
2	39.0	15	11.0	1	1.0	1	0	0	0	0	0.0	0	0	1	0	0	0	1
3	35.0	9	100.0	1	2.7	2	0	0	0	0	0.0	0	0	1	0	0	0	1
4	35.0	8	45.0	4	1.0	2	0	0	0	0	0.0	1	0	1	0	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4995	29.0	3	40.0	1	1.9	3	0	0	0	0	1.0	0	0	0	0	0	0	0
4996	30.0	4	15.0	4	0.4	1	85	0	0	0	1.0	0	0	0	0	0	0	0
4997	63.0	39	24.0	2	0.3	3	0	0	0	0	0.0	0	0	0	0	0	0	0
4998	65.0	40	49.0	3	0.5	2	0	0	0	0	1.0	0	0	0	0	0	0	0
4999	28.0	4	83.0	3	0.8	1	0	0	0	0	1.0	1	0	0	0	0	0	0

4999 rows × 18 columns

2.1.6. Missing values in the ‘Income’ column are filled with the median value

It is a sensible decision to handle missing values in the "Income" column by replacing them with the median even when the column contains outliers since "outliers aren't handled because anyone can earn any money."

This choice was made with the knowledge that there is a wide range of potential income levels for everyone represented in the "Income" column. Income outliers may emerge because of a variety of variables, including highly compensated employees, successful business owners, or other extraordinary income sources. These anomalies may indicate people with far higher salaries than average, thus they are not always errors.

The approach avoids potentially altering the income distribution by selecting to replace missing values with the median, especially if these missing values pertain to cases with distinctive income profiles. The central tendency of the income distribution will remain intact and won't be significantly impacted by extreme outlier values if missing data are replaced with the median.

It's significant to remember that several approaches, such as capping or converting the outliers, could be taken into consideration for handling outliers. However, in this instance, given the recognition that people can in fact have very high incomes, the choice to keep the outliers while impute missing values using the median upholds the integrity of the data's income distribution and considers the potential for a wide range of income profiles.

```python
In [23]: # calculate the median of 'Income' column
median_income = df_preprocessed['Income'].median()

# Fill missing values in the 'Income column with the median
df_preprocessed['Income'].fillna(median_income, inplace=True)
df_preprocessed
```

```
C:\Users\LEVEL51PC\AppData\Local\Temp\ipykernel_12876\1956685948.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
  df_preprocessed['Income'].fillna(median_income, inplace=True)
```

Out[23]:

	ID	Age	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Personal Loan	Securities Account	CD Account	Online	CreditCard	F	M	O	Home Mortage	Ho Ow
0	1	25	1	49.0	91107	4	1.6	1	0	0	1	0	0.0	0	0	1	0	0	
1	2	45	19	34.0	90089	3	1.5	1	0	0	1	0	0.0	0	0	1	0	0	
2	3	39	15	11.0	94720	1	1.0	1	0	0	0	0	0.0	0	0	1	0	0	
3	4	35	9	100.0	94112	1	2.7	2	0	0	0	0	0.0	0	0	1	0	0	
4	5	35	8	45.0	91330	4	1.0	2	0	0	0	0	0.0	1	0	1	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
4995	4996	29	3	40.0	92697	1	1.9	3	0	0	0	0	1.0	0	0	0	0	0	
4996	4997	30	4	15.0	92037	4	0.4	1	85	0	0	0	1.0	0	0	0	0	0	
4997	4998	63	39	24.0	93023	2	0.3	3	0	0	0	0	0.0	0	0	0	0	0	
4998	4999	65	40	49.0	90034	3	0.5	2	0	0	0	0	1.0	0	0	0	0	0	
4999	5000	28	4	83.0	92612	3	0.8	1	0	0	0	0	1.0	1	0	0	0	0	

4999 rows × 20 columns

2.1.7. Missing values in the 'Online' column is filled with the mode value

Based on the justification that "missing values in the 'Online' column are less and using the mode wouldn't make any significant difference," it is reasonable to fill missing values in the 'Online' column with the mode value.

The 'Online' column probably refers to a categorical variable that indicates if a person engages in online transactions or other activities. It is assumed that the data for this attribute is either unavailable or was only recorded for a limited portion of the entire dataset when missing values are present in this column.

Since there aren't many missing values, it makes reasonable to fill them up with the mode (value that appears the most frequently).

In essence, filling missing values with the mode value in the 'Online' column is a pragmatic choice that maintains the integrity of the data while ensuring that the impact on the analysis and results is minimal due to the small number of missing values.

```
In [24]: df_preprocessed['Online'].value_counts(dropna = False)

Out[24]: 1.0    2960
         0.0    1999
         NaN      40
         Name: Online, dtype: int64

In [25]: # Since there are only few null values we can replace the null values using mode imputation
         df_preprocessed['Online'].mode()

Out[25]: 0    1.0
         Name: Online, dtype: float64
```

In [26]:
```python
# Since there are only few null values we can replace the null values using mode imputation

# Calculate the mode of the 'Online' column
mode_online = df_preprocessed['Online'].mode()[0]

# Fill missing categorical values in the 'Online' column with the mode
df_preprocessed['Online'].fillna(mode_online, inplace=True)
df_preprocessed
```

```
C:\Users\LEVEL51PC\AppData\Local\Temp\ipykernel_12876\1113759204.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
  df_preprocessed['Online'].fillna(mode_online, inplace=True)
```

Out[26]:

	ID	Age	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Personal Loan	Securities Account	CD Account	Online	CreditCard	F	M	O	Home Mortage	Hc Ow
0	1	25	1	49.0	91107	4	1.6	1	0	0	1	0	0.0	0	0	1	0	0	
1	2	45	19	34.0	90089	3	1.5	1	0	0	1	0	0.0	0	0	1	0	0	
2	3	39	15	11.0	94720	1	1.0	1	0	0	0	0	0.0	0	0	1	0	0	
3	4	35	9	100.0	94112	1	2.7	2	0	0	0	0	0.0	0	0	1	0	0	
4	5	35	8	45.0	91330	4	1.0	2	0	0	0	0	0.0	1	0	1	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
4995	4996	29	3	40.0	92697	1	1.9	3	0	0	0	0	1.0	0	0	0	0	0	
4996	4997	30	4	15.0	92037	4	0.4	1	85	0	0	0	1.0	0	0	0	0	0	
4997	4998	63	39	24.0	93023	2	0.3	3	0	0	0	0	0.0	0	0	0	0	0	
4998	4999	65	40	49.0	90034	3	0.5	2	0	0	0	0	1.0	0	0	0	0	0	
4999	5000	28	4	83.0	92612	3	0.8	1	0	0	0	0	1.0	1	0	0	0	0	

4999 rows × 20 columns

2.2. Exploratory Data Analysis

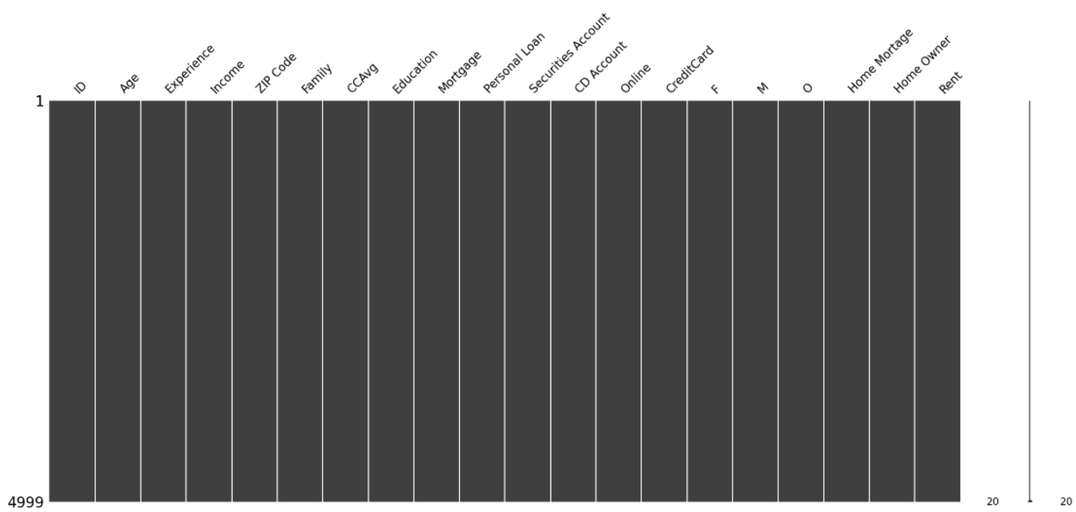
2.2.1. Visualizations are used to inspect the distributions and characteristics of the processed features.

This code utilizes the missingno library to visualize and identify missing values in a pandas DataFrame named df_preprocessed. We can see that there are no missing values after the preprocessing is done.

In [27]:
```
# missingno helps us to see if there are any missing values in dataframe, the white horizontal lines
# indicate missing values, we can see none. Therefore, we can conclude that there are no missing values in our dataframe
import missingno as msno

msno.matrix(df_preprocessed)
```

Out[27]: <Axes: >



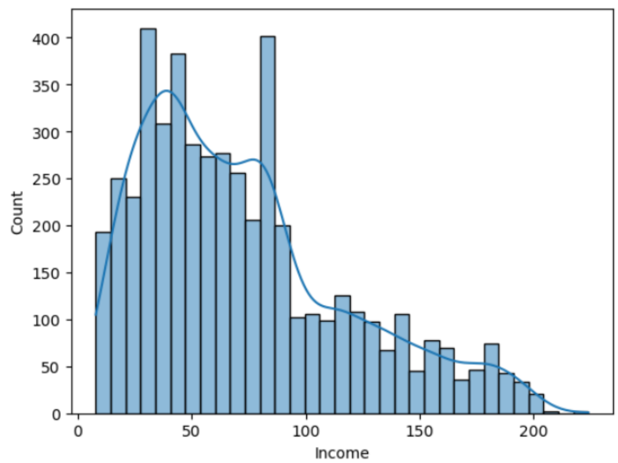
2.2.2. Histograms, box plots are employed to analyze features like ‘Income’, ‘Age’.

We can see that the Histogram for ‘Income’ column is skewed to the right. It denotes that the data is highly favored to the right. The outliers in the ‘Income’ column weren’t handles as mentioned above.

The box plot for ‘Income’ column also shows that there are outliers in the column.

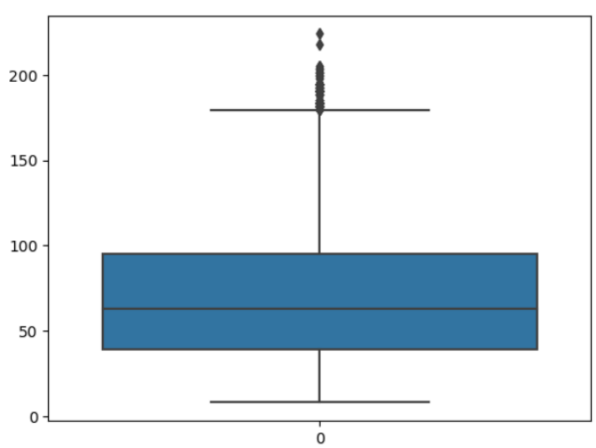
In [21]:
```
import seaborn as sns
sns.histplot(df_preprocessed['Income'], kde=True)
```

Out[21]: <Axes: xlabel='Income', ylabel='Count'>



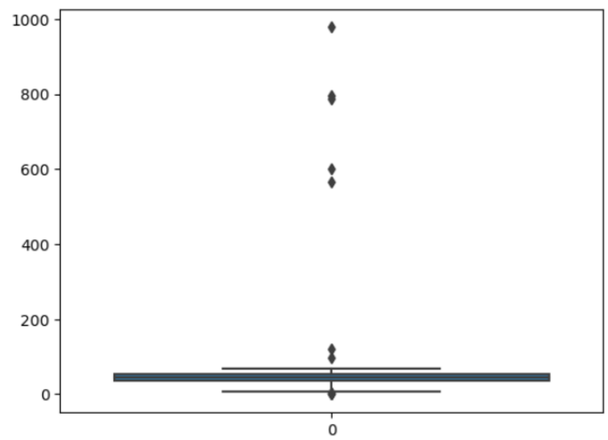
In [22]:
```
sns.boxplot(df_preprocessed['Income'])
```

Out[22]: <Axes: >



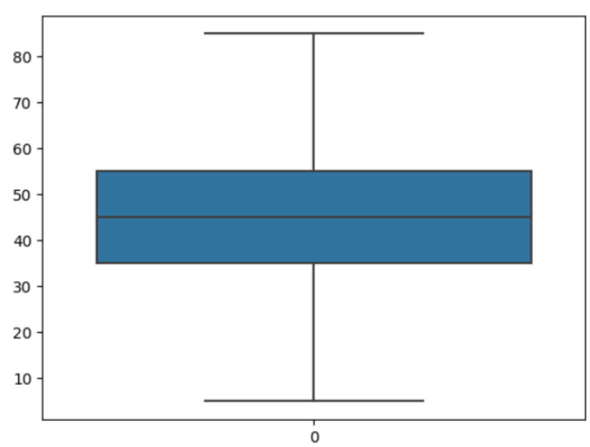
In the figure below, we can see that the ‘Age’ column contains outliers. The figure below shows the boxplot for ‘Age’ column before dealing with outliers.

```
In [28]: sns.boxplot(df_preprocessed['Age'])
Out[28]: <Axes: >
```



The outliers in the ‘Age' column were handles using the IQR proximity method and the outliers were capped with a value of 85. The figure below shows the boxplot of ‘Age’ column after dealing with outliers. We can see that there are no remaining outliers.

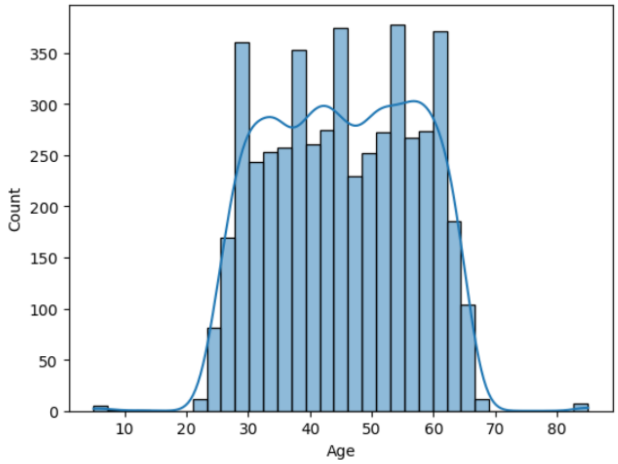
```
In [38]: # After the outliers detection and dealing with them, we can see that the box-plot for Age column has no outliers
         sns.boxplot(df_final['Age'])
Out[38]: <Axes: >
```



After carrying out outlier detection and handling in the ‘Age’ column, the histogram for the ‘Age’ column looked normally distributed.

In [37]:
```python
# After the outliers detection and dealing with them, we can see that the plot for Age column is normally distributed
sns.histplot(df_final['Age'], kde=True)
```

Out[37]: <Axes: xlabel='Age', ylabel='Count'>



2.3. Modeling and Evaluation

2.3.1. The dataset is split into training and testing sets.

To assess a machine learning model's effectiveness, the dataset is divided into training and testing sets. The model is trained using the training set, which enables it to discover patterns and connections in the data. The model's ability to generalize to fresh, untested data is next evaluated using the testing set. By modeling the model's performance on data, it has never seen before during training, this separation aids in estimating how well it will perform on real-world data. It aids in avoiding overfitting, which occurs when a model performs well on training data but badly on fresh data, and it offers a more accurate assessment of the model's performance.

Here the test size is 0.2 whereas the train size is 0.8.

```
In [63]: # Split the data into features (X) and target (Y)
         X = df_final.drop('Personal Loan', axis=1)
         y = df_final['Personal Loan']

         # Split into train and test sets
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2.3.2. Class weights are adjusted due to the imbalanced nature of the ‘Personal Loan’ target variable.

In the presence of unbalanced data, class weights are modified to account for the uneven distribution of classes in a classification task. If one class, such as "No Personal Loan," greatly outweighs the other class, such as "Personal Loan," a model may be biased toward forecasting the majority class, which would result in subpar performance for the minority class in the context of the "Personal Loan" objective.

The 'class_weights' argument in Logistic Regression is used to give certain classes in the training process distinct weights. By giving the minority class more power and significance, the model can learn to predict outcomes more accurately for both classes—even when there is inequality. In essence, adjusting class weights aids in lessening the effects of class imbalance and encourages a more balanced educational process.

```
# Since the target column "Personal Loan" has an imbalanced data, we need to use class weights to balance it
class_0_count = len(y_train) - y_train.sum()
class_1_count = y_train.sum()
print("class_0_count:", class_0_count)
print("class_1_count:", class_1_count)

# Assigning class_weight
class_weights = {0: class_1_count / class_0_count, 1: 1}
print(class_weights)
```

2.3.3. A Logistic Regression model is trained on the training data

A Logistic Regression model is being instantiated in the figure below. The instantiated model is fitted or trained on the training data (‘X_train’ and ‘y_train’). The model learns the relationship between the features (‘X_train’) and the target variable (‘y_train') during this step.

After training, the model is used to make predictions on the test data(‘X_test’). The predicted values for the target variable are stored in the ‘y_pred’ variable.

```
# Create a Logistic Regression Model with class weights
model = LogisticRegression(class_weight=class_weights)

# Fit the model on the training data
model.fit(X_train, y_train)

#Predict on the test data
y_pred = model.predict(X_test)
```

2.3.4. The model’s performance is evaluated using accuracy, confusion matrix, and classification report.

- These steps are crucial for evaluating the model's performance and understanding its strengths and weaknesses.
- The confusion matrix provides a comprehensive view of the model's predictive behavior across different classes.
- The classification report offers detailed insights into metrics that account for precision, recall, and their harmonic mean (F1-score).
- Accuracy provides a general measure of correctness, but it's important to consider it alongside other metrics, especially in imbalanced datasets.

In summary, these steps quantify the performance of the model in terms of true and false predictions, precision, recall, and accuracy. They help assess the model's suitability for the specific problem and guide potential adjustments for better outcomes.

In the figure below:

Confusion Matrix

The confusion matrix shows a 2x2 matrix with four values:

- Top-left: True negatives (TN) - 799 instances were correctly predicted as class 0 (not approved).
- Top-right: False positives (FP) - 103 instances were wrongly predicted as class 1 (approved) when they were class 0.
- Bottom-left: False negatives (FN) - 17 instances were wrongly predicted as class 0 when they were class 1.

- Bottom-right: True positives (TP) - 81 instances were correctly predicted as class 1.

Classification Report

- **Precision:** Precision is the ratio of correctly predicted positive instances (TP) to the total predicted positives (TP + FP). For class 0, it's 0.98, and for class 1, it's 0.44.
- **Recall:** Recall (also known as sensitivity) is the ratio of correctly predicted positive instances (TP) to the total actual positives (TP + FN). For class 0, it's 0.89, and for class 1, it's 0.83.
- **F1-score:** F1-score is the harmonic mean of precision and recall, which provides a balanced metric for binary classification. For class 0, it's 0.93, and for class 1, it's 0.57.
- **Support:** The number of occurrences of each class in the test set. For class 0, it's 902, and for class 1, it's 98.
- **Accuracy:** The overall accuracy of the model's predictions, calculated as the ratio of correct predictions to the total number of instances. It's 0.88 or 88%.
- **Macro Average (macro avg):** The average of precision, recall, and F1-score across classes, without considering class imbalance. It's 0.71 for precision, 0.86 for recall, and 0.75 for F1-score.
- **Weighted Average (weighted avg):** The weighted average of precision, recall, and F1-score, taking into account the class imbalance. It's 0.93 for precision, 0.88 for recall, and 0.90 for F1-score.

```python
#Calculate and print confusion matreix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

# Print classification report
print(classification_report(y_test, y_pred))

#Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy: .2f}")
```

```
class_0_count: 3618
class_1_count: 381
{0: 0.10530679933665009, 1: 1}
Confusion Matrix:
[[799 103]
 [ 17  81]]
              precision    recall  f1-score   support

           0       0.98      0.89      0.93       902
           1       0.44      0.83      0.57        98

    accuracy                           0.88      1000
   macro avg       0.71      0.86      0.75      1000
weighted avg       0.93      0.88      0.90      1000

Accuracy:  0.88
```

2.3.5. An ROC curve is plotted, and the AUC value is calculated to assess the model’s discriminative power.

- The AUC is a measure of the model's ability to distinguish between the positive and negative classes. It quantifies the overall performance of the model across different threshold values for classification.
- An AUC value of 0.92 indicates that the model is performing well in differentiating between the two classes. The higher the AUC, the better the model's ability to correctly rank positive instances higher than negative instances, on average.
- An AUC value of 0.92 suggests that the model's predictions are reasonably accurate and the ROC curve demonstrates good separation between true positive rate and false positive rate, indicating effective discrimination between the two classes.

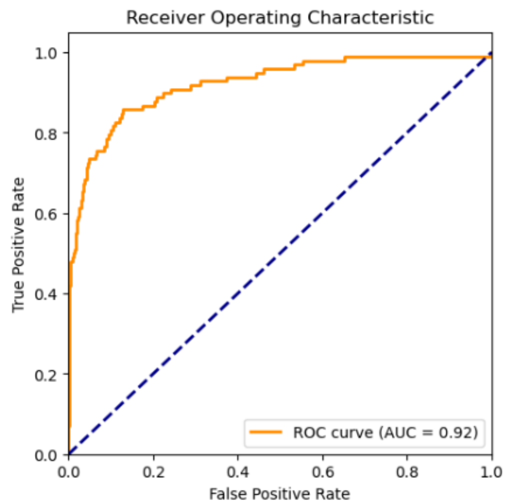
In summary, the code calculates the ROC curve and AUC for the model's predictions, and the AUC value of 0.92 indicates strong predictive performance, with the model effectively distinguishing between loan approvals and non-approvals.

In [45]:
```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Get predicted probabilities for positive class
y_pred_proba = model.predict_proba(X_test)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure(figsize=(5, 5))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



The trained machine learning model is saved in a format that can be easily loaded and used for prediction. The code for doing so is shown below:

1. Save Model Using Pickle:

- The code begins by importing the `pickle` module, which allows serializing (saving) and deserializing (loading) Python objects.

- The trained Logistic Regression model (`model`) is saved to a file named 'model_pickle' using the `pickle.dump()` function.

- The file is opened in 'write binary' mode (‘wb’) to ensure compatibility across different platforms.

2. Load Model Using Pickle:

- The code then proceeds to load the previously saved model back into memory.

- The 'model_pickle' file is opened in 'read binary' mode (‘rb’).

- The `pickle.load()` function is used to load the model object from the file and store it in the `loaded_model` variable.

In summary, this code snippet demonstrates how to serialize and deserialize a machine learning model using the `pickle` module. The model is saved to a file after training and can later be loaded back into memory for future use or deployment. This approach allows for efficient storage and retrieval of trained models, making them available for making predictions without the need to retrain them.

```
In [91]: # Import the pickle module for serialization
         import pickle

In [92]: # Save the trained model to a file
         # 'wb' stands for write binary mode, ensuring compatibility across different platforms
         with open('model_pickle', 'wb') as file:
             # Use the pickle.dump() function to save the model into the file
             pickle.dump(model, file)

In [93]: # Load the model back
         # 'rb' stands for read binary mode, matching the mode used during saving
         with open('model_pickle', 'rb') as file:
             # Use the pickle.load() function to load the model from the file
             loaded_model = pickle.load(file)

In [94]: # Use the loaded model for prediction
         # Here, X_new represents the new data you want to predict on
         # predictions = loaded_model.predict(X_new)
```

3. Insights and Observations

3.1. Data Preprocessing:

- Missing values and outliers are handled appropriately to ensure the quality of the dataset.
- One-hot encoding is used to transform categorical variables into numerical features.
- The final preprocessed dataset is ready for modeling, with unnecessary columns removed.

3.2. Model Performance:

- The Logistic Regression model is able to achieve a certain level of accuracy in predicting loan approvals.
- Class weights are adjusted to address the class imbalance issue, resulting in a more balanced performance evaluation.
- The ROC curve indicates the model's trade-off between true positive rate and false positive rate.

4. Recommendations

4.1. Feature Engineering:

- Further feature engineering could be explored to potentially improve the model's

4.2. Model Selection:

- Consider exploring other classification algorithms such as Random Forest, Gradient Boosting, or Support Vector Machines to compare their performance against the Logistic Regression model.

5. Conclusion

The analysis provides a thorough method for preprocessing loan data and using Logistic Regression to create a predictive model for loan approval. The effectiveness of the model is assessed using a number of measures, and suggestions are made for future development. Subject to continuing upgrades and optimizations, the serialized model can be used to anticipate loan acceptance in real time. The suggestions made can direct subsequent iterations of the study to improve model precision and generalization.