In [1]:
```
#importing data
import pandas as pd
data = pd.read_csv('credit_risk.csv')
data.head(10)
```

Out[1]:

	Id	Age	Income	Home	Emp_length	Intent	Amount	Rate	Status	Percent_income
0	0	22	59000	RENT	123.0	PERSONAL	35000	16.02	1	0.59
1	1	21	9600	OWN	5.0	EDUCATION	1000	11.14	0	0.10
2	2	25	9600	MORTGAGE	1.0	MEDICAL	5500	12.87	1	0.57
3	3	23	65500	RENT	4.0	MEDICAL	35000	15.23	1	0.53
4	4	24	54400	RENT	8.0	MEDICAL	35000	14.27	1	0.55
5	5	21	9900	OWN	2.0	VENTURE	2500	7.14	1	0.25
6	6	26	77100	RENT	8.0	EDUCATION	35000	12.42	1	0.45
7	7	24	78956	RENT	5.0	MEDICAL	35000	11.11	1	0.44
8	8	24	83000	RENT	8.0	PERSONAL	35000	8.90	1	0.42
9	9	21	10000	OWN	6.0	VENTURE	1600	14.74	1	0.16

In [2]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32581 entries, 0 to 32580
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Id              32581 non-null  int64
 1   Age             32581 non-null  int64
 2   Income          32581 non-null  int64
 3   Home            32581 non-null  object
 4   Emp_length      31686 non-null  float64
 5   Intent          32581 non-null  object
 6   Amount          32581 non-null  int64
 7   Rate            29465 non-null  float64
 8   Status          32581 non-null  int64
 9   Percent_income  32581 non-null  float64
 10  Default         32581 non-null  object
 11  Cred_length     32581 non-null  int64
dtypes: float64(3), int64(6), object(3)
memory usage: 3.0+ MB
```

Column Descriptions:

- **ID:** Unique identifier for each loan applicant.
- **Age:** Age of the loan applicant.
- **Income:** Income of the loan applicant.
- **Home:** Home ownership status (Own, Mortgage, Rent).
- **Emp_Length:** Employment length in years.
- **Intent:** Purpose of the loan (e.g., education, home improvement).

- **Amount:** Loan amount applied for.
- **Rate:** Interest rate on the loan.
- **Status:** Loan approval status (Fully Paid, Charged Off, Current).
- **Percent_Income:** Loan amount as a percentage of income.
- **Default:** Whether the applicant has defaulted on a loan previously (Yes, No).
- **Cred_Length:** Length of the applicant's credit history.

In [3]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

In [4]:
```
# Check missing values
print("Missing Values:\n", data.isnull().sum())
```

```
Missing Values:
 Id                   0
Age                   0
Income                0
Home                  0
Emp_length          895
Intent                0
Amount                0
Rate               3116
Status                0
Percent_income        0
Default               0
Cred_length           0
dtype: int64
```

HANDLING MISSING VALUES

There are 895 missing values in `Emp_length` which represents length of employment in years. since the length of employment in years is susceptible to skweiness, we will use the median to fill in the missing values of `emp_length`

there are 3116 missing values in `Rate` which represents the applicable interest rate. To deal with the missing value, we will group the data by `intent` then fill in the missing data with the rate of its marching intent. this is because, interest rate is a financial tool that can be and is aftfected by the intent of the loan in this case.

see the cels below

In [5]:
```
 #Filling Emp_Length missing values with the median
data['Emp_length'].fillna(data['Emp_length'].median(), inplace=True)
```

In [6]:
```
# Filling Rate missing values with mean per Intent category
data['Rate'] = data.groupby('Intent')['Rate'].transform(lambda x: x.fillna(x.mean
```

In [7]:
```
# Confirm missing values are handled
print(data.isnull().sum())
```

```
Id                0
Age               0
Income            0
Home              0
Emp_length        0
Intent            0
Amount            0
Rate              0
Status            0
Percent_income    0
Default           0
Cred_length       0
dtype: int64
```

In [8]:
```
# Check for duplicates
print("Duplicates: ", data.duplicated().sum())
```

```
Duplicates:  0
```

In [9]:
```
data['Intent'].unique()
```

Out[9]:
```
array(['PERSONAL', 'EDUCATION', 'MEDICAL', 'VENTURE', 'HOMEIMPROVEMENT',
       'DEBTCONSOLIDATION'], dtype=object)
```

In [10]:
```
data['Home'].unique()
```

Out[10]:
```
array(['RENT', 'OWN', 'MORTGAGE', 'OTHER'], dtype=object)
```

There are no Duplicates in the data.

DEALING WITH NON NUMERICAL COLUMNS

The data set has 3 non numerical colums namely `Home` , `Intent` and `Default` . `Home` have 4 unique variables namely RENT, OWN, MORTGAGE and OTHER while `Intent` has 6 unque variables namely PERSONAL, EDUCATION, MEDICAL, VENTURE, HOMEIMPROVEMENT AND DEBTCONSOLIDATION. The variables in these two colums have no inherent order thus we cant use label encoding to convert the columns to numerical values but will instead use one hot encoding

`Default` column on the other hand have two variables yes or no which has an inherent order making it easy to convert to numerical using label encoding.

we will make a copy of the data in order to retain our original data then transform the columns using the copy of the data. see the cells below.

In [11]:
```
# Make a copy of the original dataset
df = data.copy()

# One-hot encode 'Home' and 'Intent'
df = pd.get_dummies(df, columns=['Home', 'Intent'], drop_first=False)

# Label encode 'Default' (Y/N to 1/0)
df['Default'] = df['Default'].map({'Y': 1, 'N': 0})
```

In [12]:
```
df.head(10)
```

Out[12]:

	Id	Age	Income	Emp_length	Amount	Rate	Status	Percent_income	Default	Cred_length	Ho
0	0	22	59000	123.0	35000	16.02	1	0.59	1	3	
1	1	21	9600	5.0	1000	11.14	0	0.10	0	2	
2	2	25	9600	1.0	5500	12.87	1	0.57	0	3	
3	3	23	65500	4.0	35000	15.23	1	0.53	0	2	
4	4	24	54400	8.0	35000	14.27	1	0.55	1	4	
5	5	21	9900	2.0	2500	7.14	1	0.25	0	2	
6	6	26	77100	8.0	35000	12.42	1	0.45	0	3	
7	7	24	78956	5.0	35000	11.11	1	0.44	0	4	
8	8	24	83000	8.0	35000	8.90	1	0.42	0	2	
9	9	21	10000	6.0	1600	14.74	1	0.16	0	3	

EXPLORATORY DATA ANALYSIS (EDA)

In this section, we will perform EDA using the following methods

- Descriptive statistics
- univariate Analysis
- Bivariate Analysis
- Multivariate Analyis

Descriptive Statistics

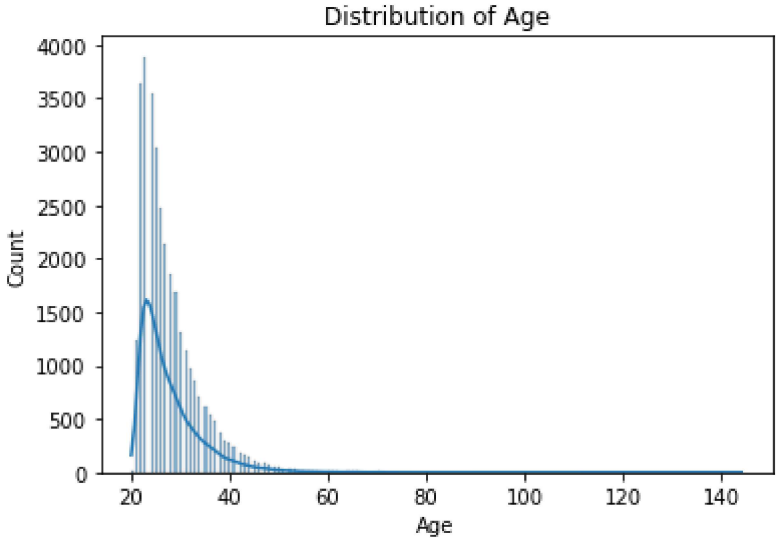
In [13]:
```
print("\nDescriptive Statistics:\n", df.describe())
```

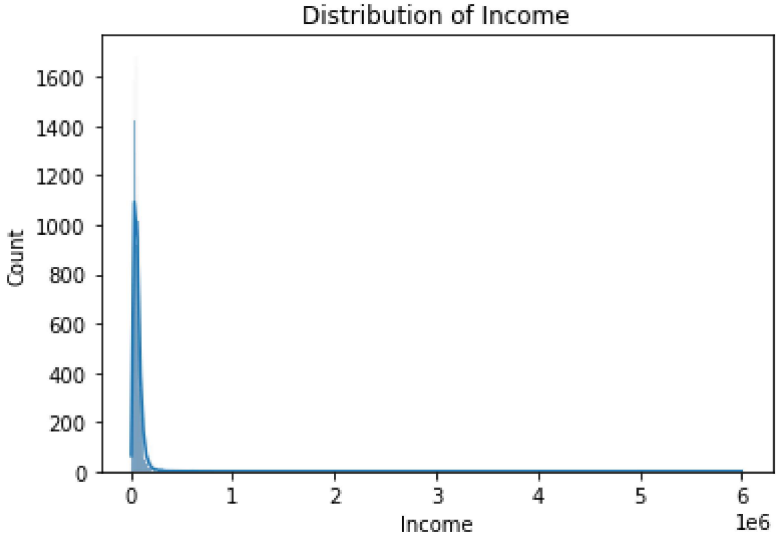
```
Descriptive Statistics:
                  Id           Age        Income    Emp_length        Amount  \
count  32581.000000  32581.000000  3.258100e+04  32581.000000  32581.000000
mean   16290.006139     27.734600  6.607485e+04      4.767994   9589.371106
std     9405.479594      6.348078  6.198312e+04      4.087372   6322.086646
min        0.000000     20.000000  4.000000e+03      0.000000    500.000000
25%     8145.000000     23.000000  3.850000e+04      2.000000   5000.000000
50%    16290.000000     26.000000  5.500000e+04      4.000000   8000.000000
75%    24435.000000     30.000000  7.920000e+04      7.000000  12200.000000
max    32780.000000    144.000000  6.000000e+06    123.000000  35000.000000

               Rate        Status  Percent_income       Default   Cred_length
\
count  32581.000000  32581.000000    32581.000000  32581.000000  32581.000000
mean      11.011565      0.218164        0.170203      0.176330      5.804211
std        3.081693      0.413006        0.106782      0.381106      4.055001
min        5.420000      0.000000        0.000000      0.000000      2.000000
25%        8.490000      0.000000        0.090000      0.000000      3.000000
50%       10.990000      0.000000        0.150000      0.000000      4.000000
75%       13.110000      0.000000        0.230000      0.000000      8.000000
max       23.220000      1.000000        0.830000      1.000000     30.000000

       Home_MORTGAGE    Home_OTHER      Home_OWN     Home_RENT  \
count   32581.000000  32581.000000  32581.000000  32581.000000
mean        0.412633      0.003284      0.079310      0.504773
std         0.492315      0.057214      0.270226      0.499985
min         0.000000      0.000000      0.000000      0.000000
25%         0.000000      0.000000      0.000000      0.000000
50%         0.000000      0.000000      0.000000      1.000000
75%         1.000000      0.000000      0.000000      1.000000
max         1.000000      1.000000      1.000000      1.000000

       Intent_DEBTCONSOLIDATION  Intent_EDUCATION  Intent_HOMEIMPROVEMENT  \
count              32581.000000      32581.000000            32581.000000
mean                   0.159971          0.198060                0.110647
std                    0.366584          0.398544                0.313700
min                    0.000000          0.000000                0.000000
25%                    0.000000          0.000000                0.000000
50%                    0.000000          0.000000                0.000000
75%                    0.000000          0.000000                0.000000
max                    1.000000          1.000000                1.000000

       Intent_MEDICAL  Intent_PERSONAL  Intent_VENTURE
count    32581.000000     32581.000000    32581.000000
mean         0.186336         0.169455        0.175532
std          0.389383         0.375159        0.380427
min          0.000000         0.000000        0.000000
25%          0.000000         0.000000        0.000000
50%          0.000000         0.000000        0.000000
75%          0.000000         0.000000        0.000000
max          1.000000         1.000000        1.000000
```

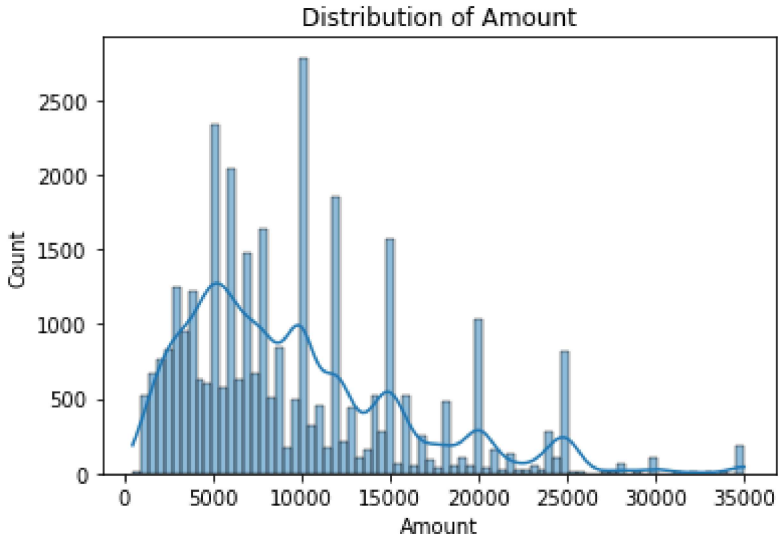
Univariate Analyis

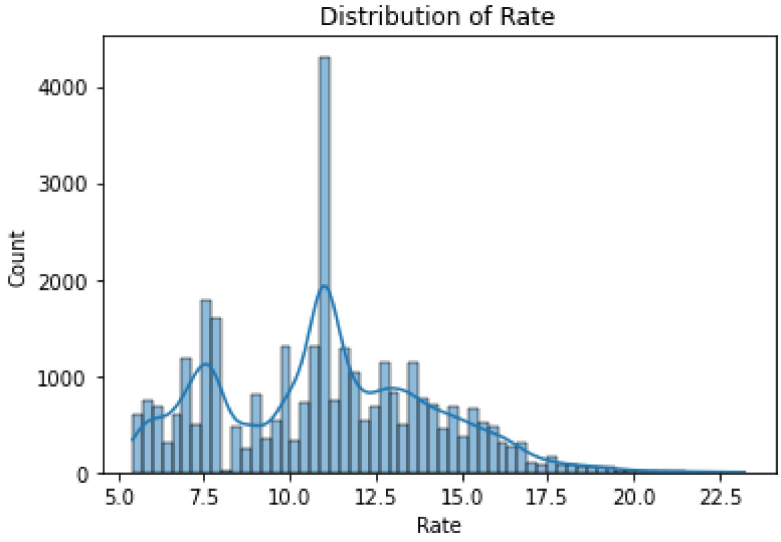
In this section we will perform univariate analysis on the different variable with an aim of understanding their ditribution.this analysis is important ans it helps determine skeweness in the data and also helps in uncovering and idenfication of any outliers.

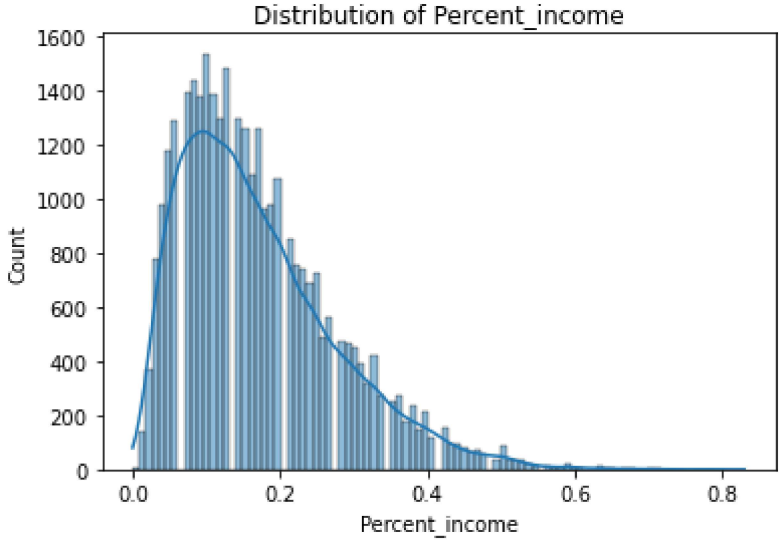
In [14]:
```
# Continuous features
num_cols = ['Age', 'Income', 'Amount', 'Rate', 'Percent_income', 'Cred_length']
for col in num_cols:
    plt.figure(figsize=(6, 4))
    sns.histplot(df[col], kde=True)
    plt.title(f'Distribution of {col}')
    plt.show()
```

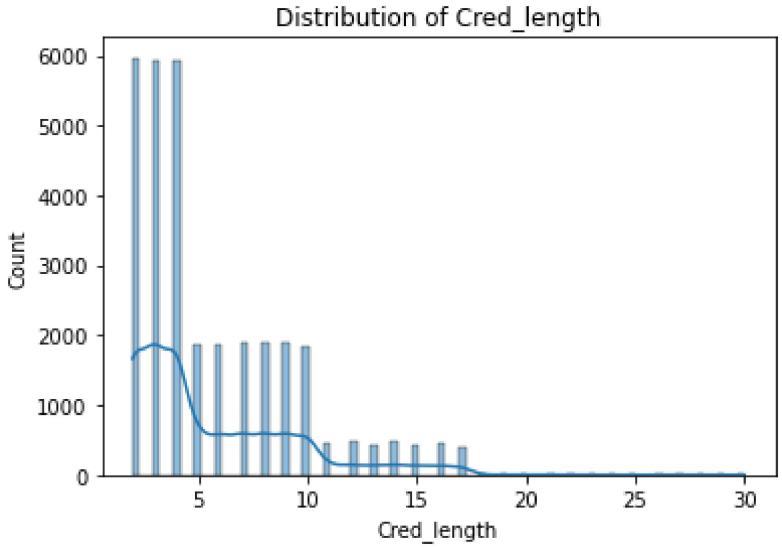




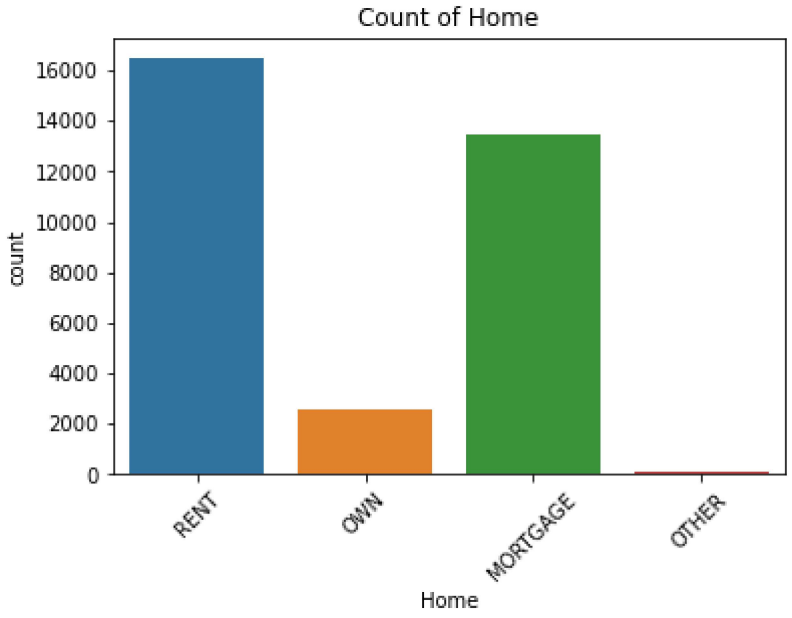

Distribution of Amount
Count
Amount
0
500
1000
1500
2000
2500
0
5000
10000
15000
20000
25000
30000
35000

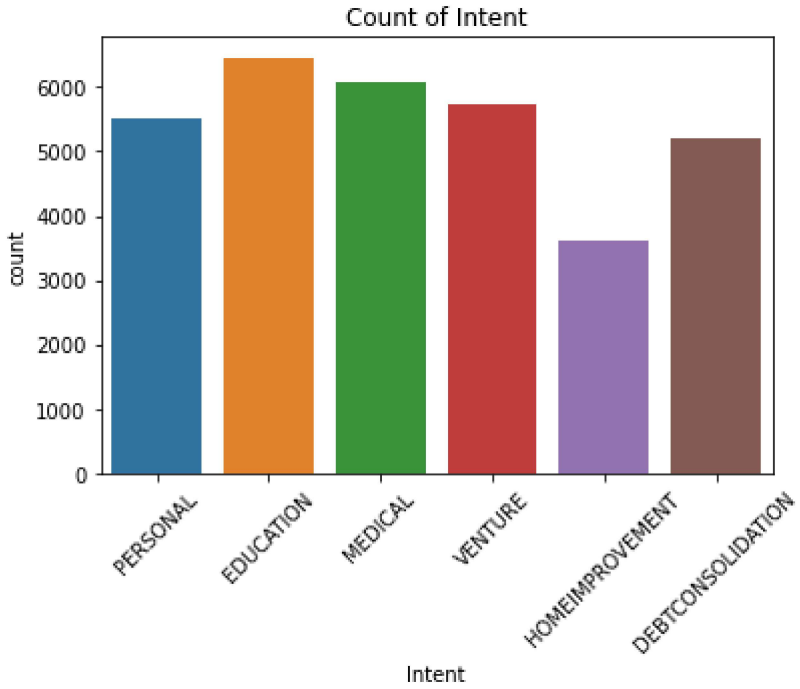

Distribution of Rate
Count
Rate
0
1000
2000
3000
4000
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5

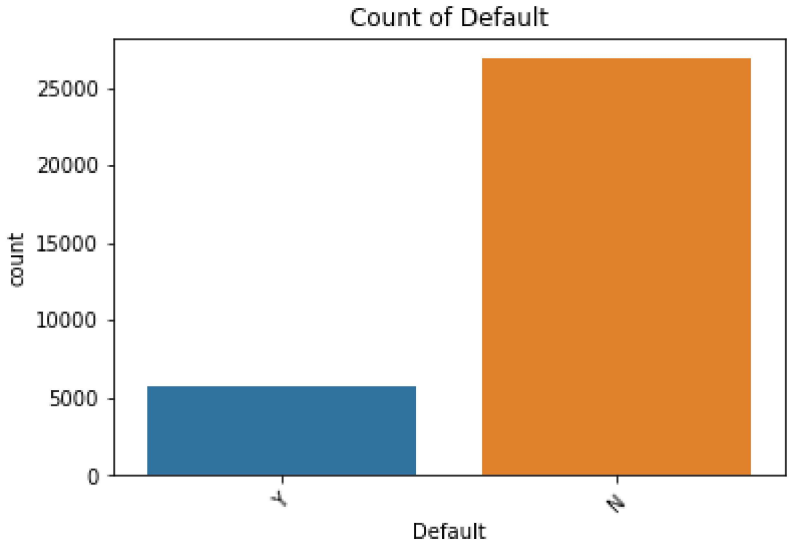

Distribution of Percent_income
Count
Percent_income
0
200
400
600
800
1000
1200
1400
1600
0.0
0.2
0.4
0.6
0.8


Distribution of Cred_length
Count
Cred_length
0
1000
2000
3000
4000
5000
6000
5
10
15
20
25
30

In [15]:
```
# Categorical features
cat_cols = ['Home', 'Intent', 'Default']
for col in cat_cols:
    plt.figure(figsize=(6, 4))
    sns.countplot(data=data, x=col)
    plt.title(f'Count of {col}')
    plt.xticks(rotation=45)
    plt.show()
```







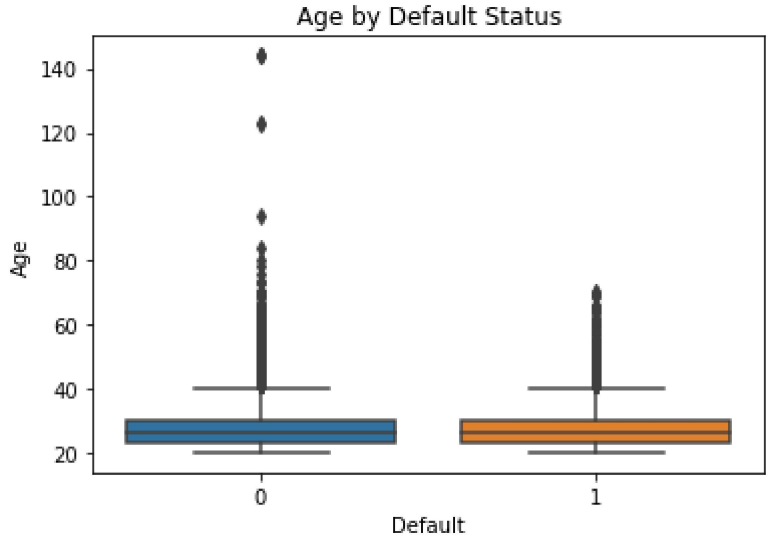
In [ ]:

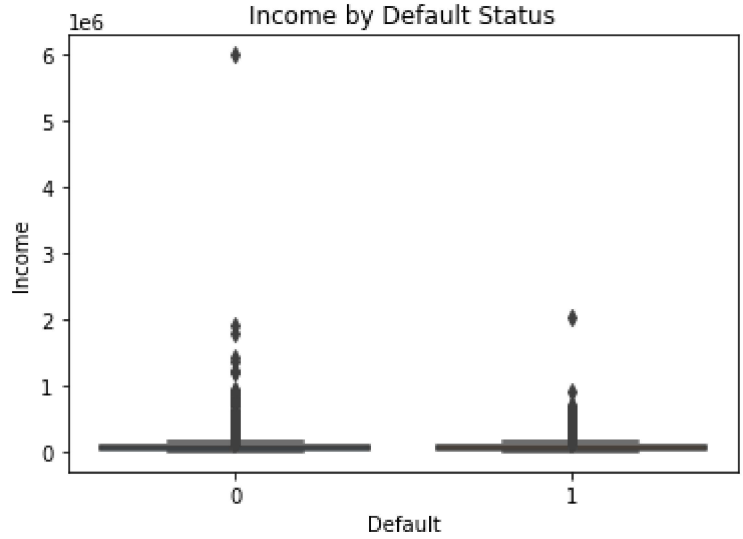
In [ ]:

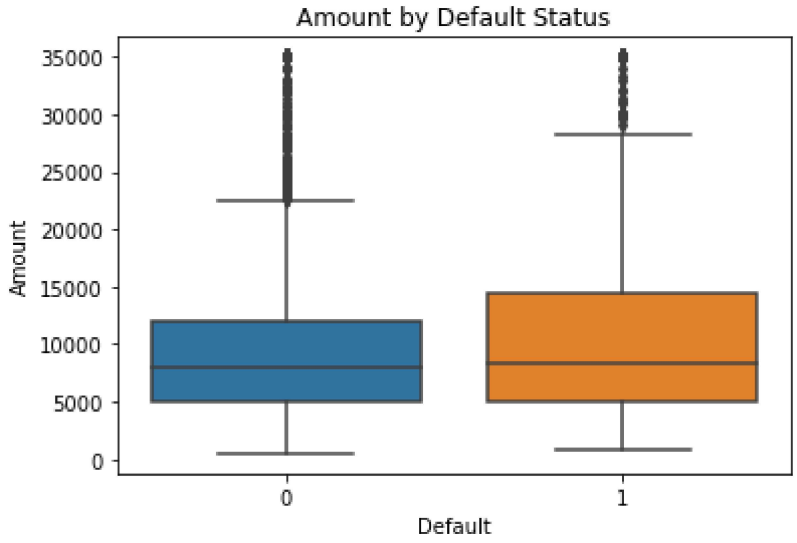
Bivariate Analysis

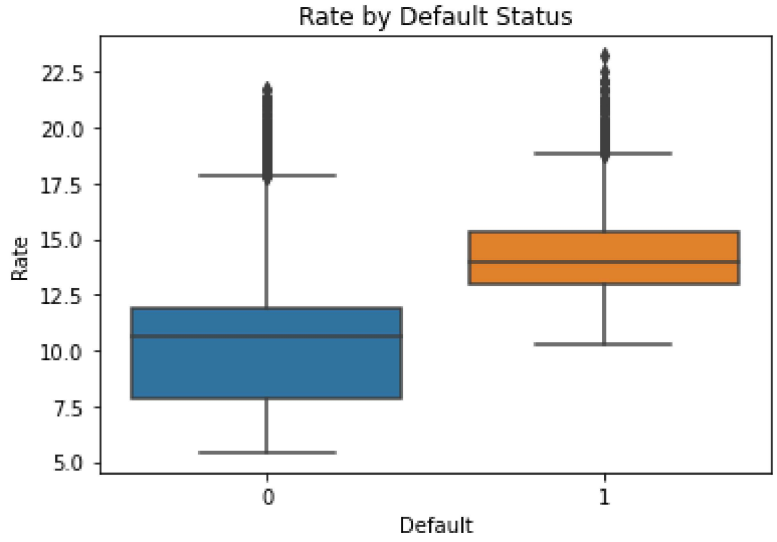
The purpose of Bivariate analysis is to determine the relationship between two variables which in turn helps to determine any correlation or predictive relationships

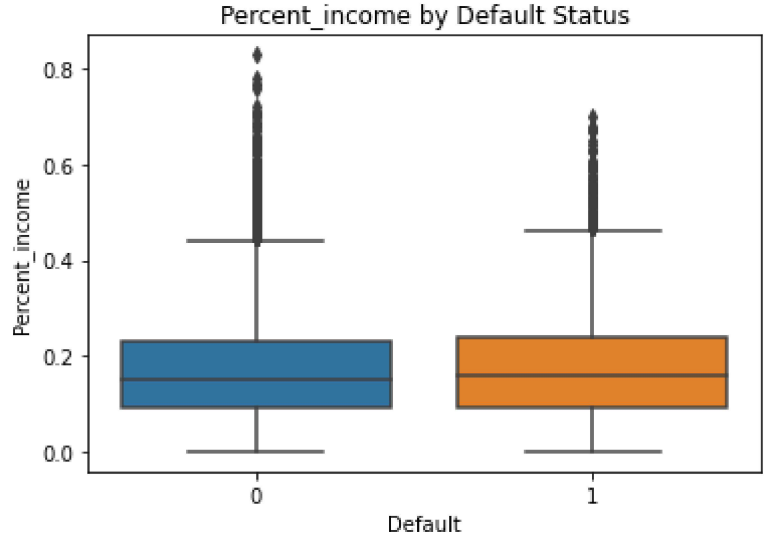
In [16]:
```
# Numerical vs Default
for col in num_cols:
    plt.figure(figsize=(6, 4))
    sns.boxplot(data=df, x='Default', y=col)
    plt.title(f'{col} by Default Status')
    plt.show()
```

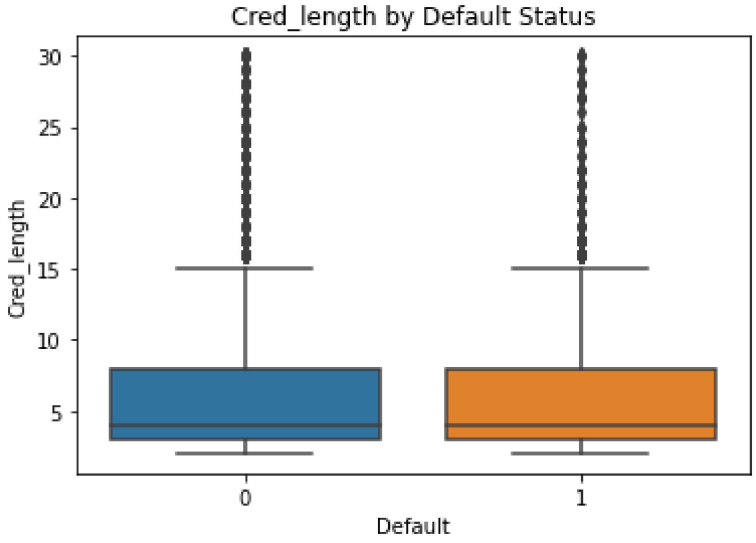




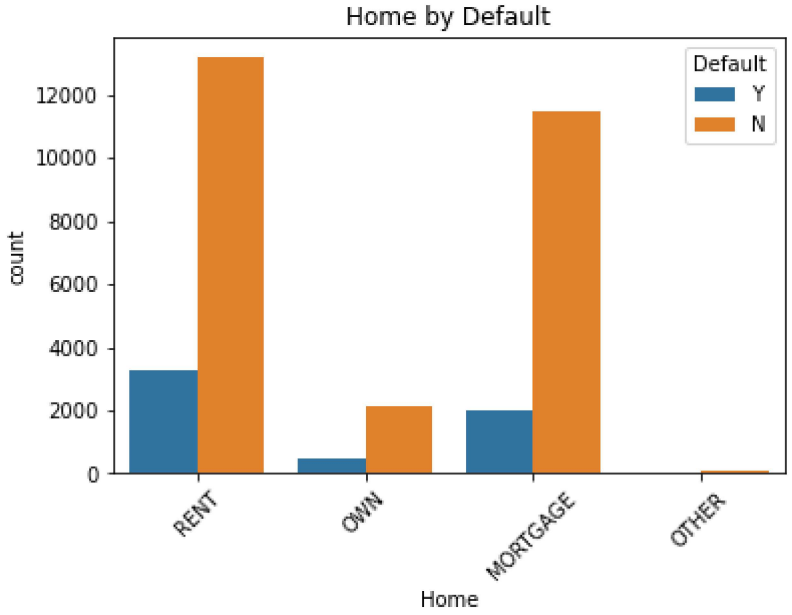

Amount by Default Status
Amount
35000
30000
25000
20000
15000
10000
5000
0
0
1
Default

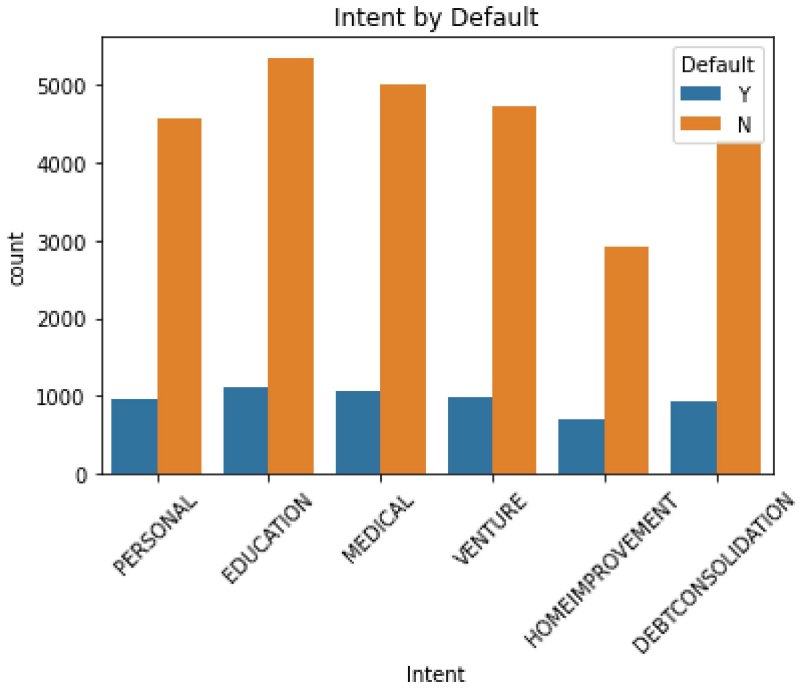

Rate by Default Status
Rate
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
0
1
Default

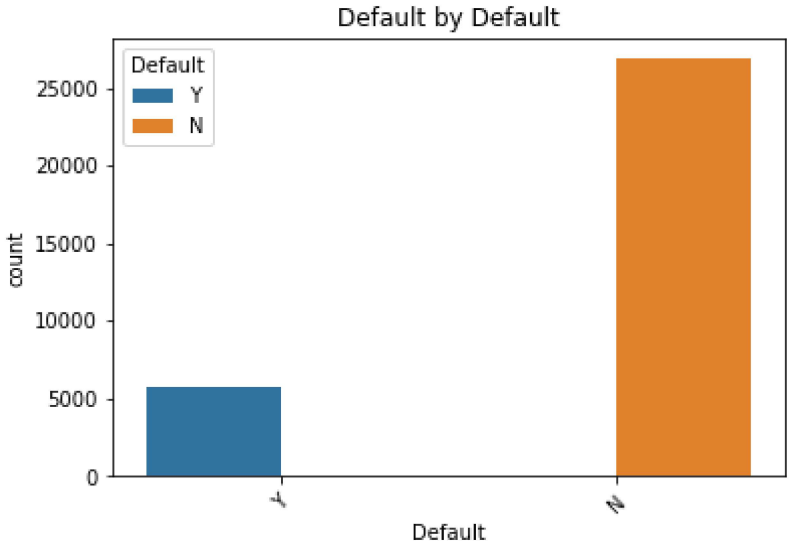

Percent_income by Default Status
Percent_income
0.8
0.6
0.4
0.2
0.0
0
1
Default


Cred_length by Default Status
30
25
20
15
10
5
Cred_length
0
1
Default

In [17]:
```
# Categorical vs Default
for col in cat_cols:
    plt.figure(figsize=(6, 4))
    sns.countplot(data=data, x=col, hue='Default')
    plt.title(f'{col} by Default')
    plt.xticks(rotation=45)
    plt.show()
```

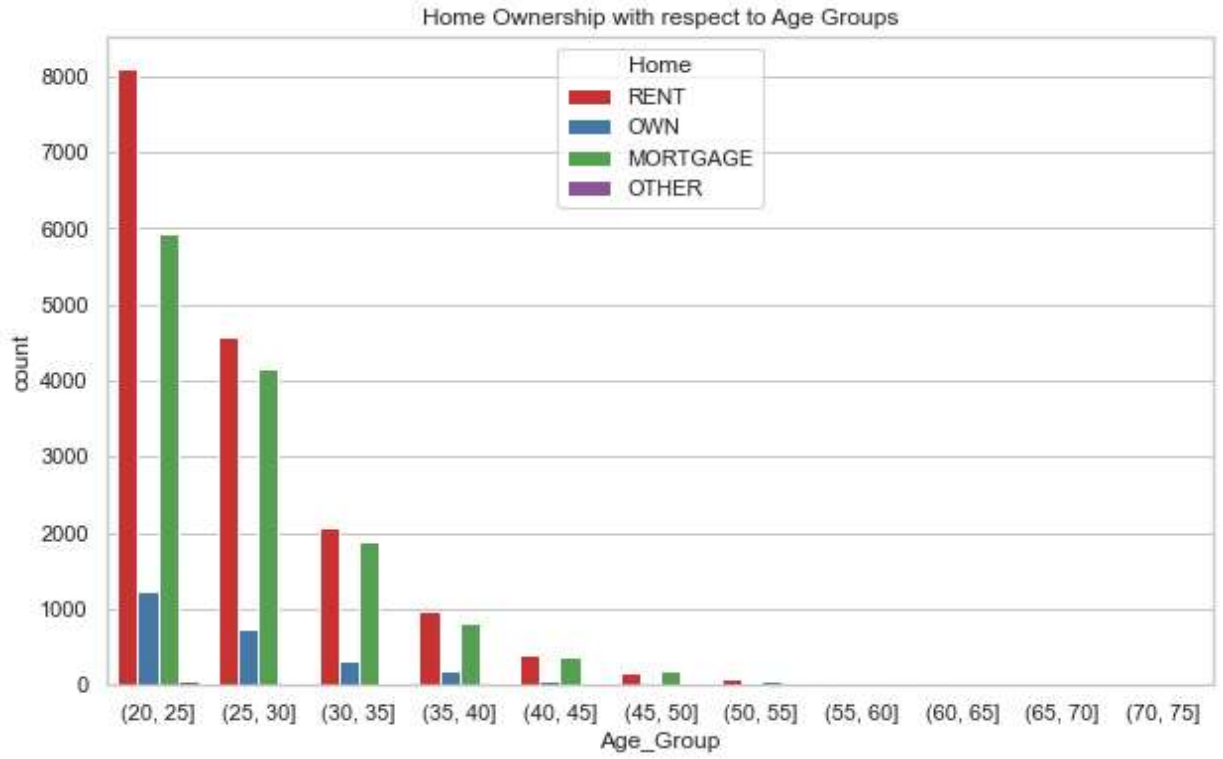


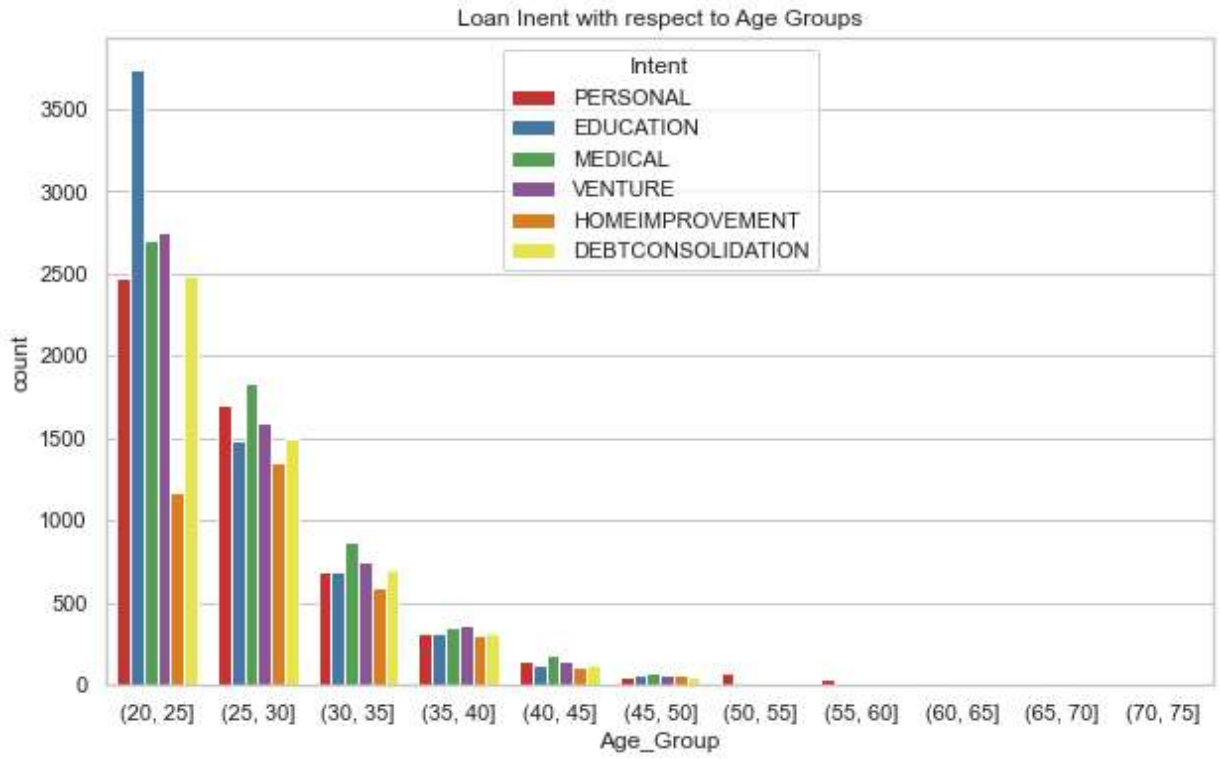


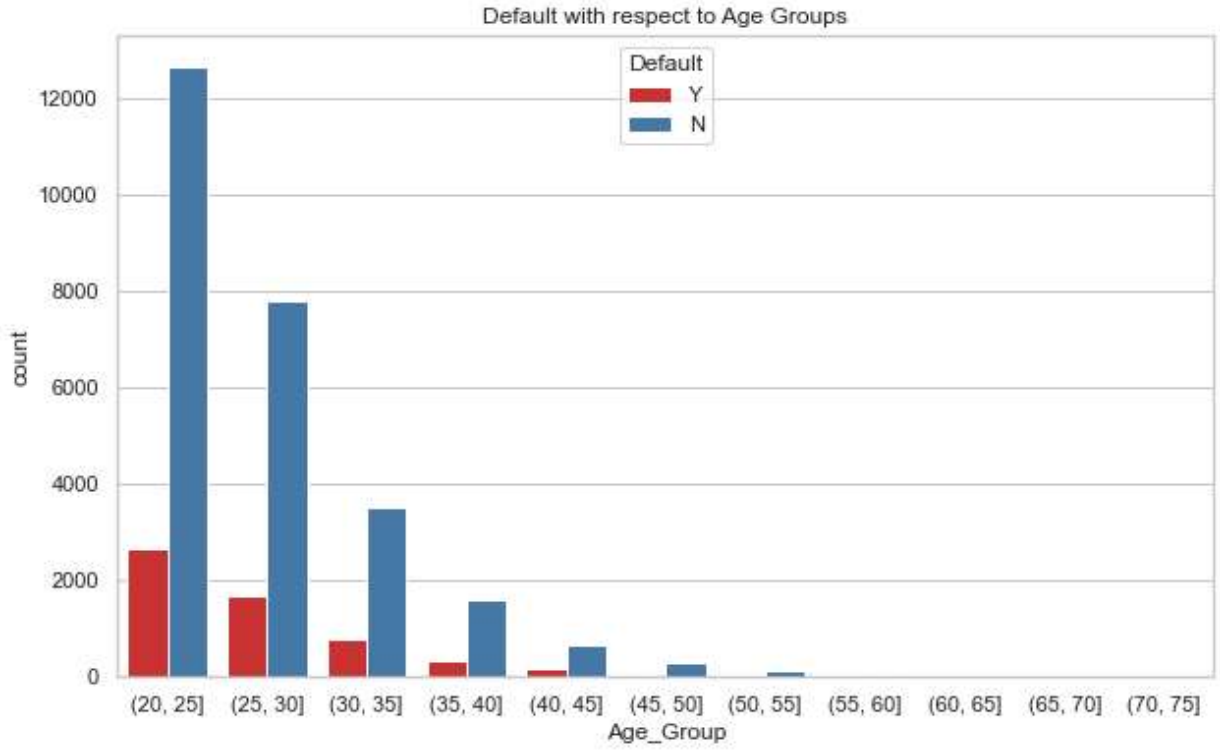


Age group vs Home ownership, Intent and default using the origina data before encoding

In [18]:
```
df2 = data.copy()
age_bins = [20,25,30, 35, 40, 45, 50, 55, 60,65,70,75]
titles = ["Home Ownership", "Loan Inent", "Default"]
df2['Age_Group'] = pd.cut(df2['Age'], bins=age_bins)
t = 0

for col in cat_cols:
    sns.set(style="whitegrid")
    plt.figure(figsize=(10, 6))
    sns.countplot(data=df2, x='Age_Group', hue=col, palette='Set1')
    plt.title(f"{titles[t]} with respect to Age Groups")
    t+=1
```



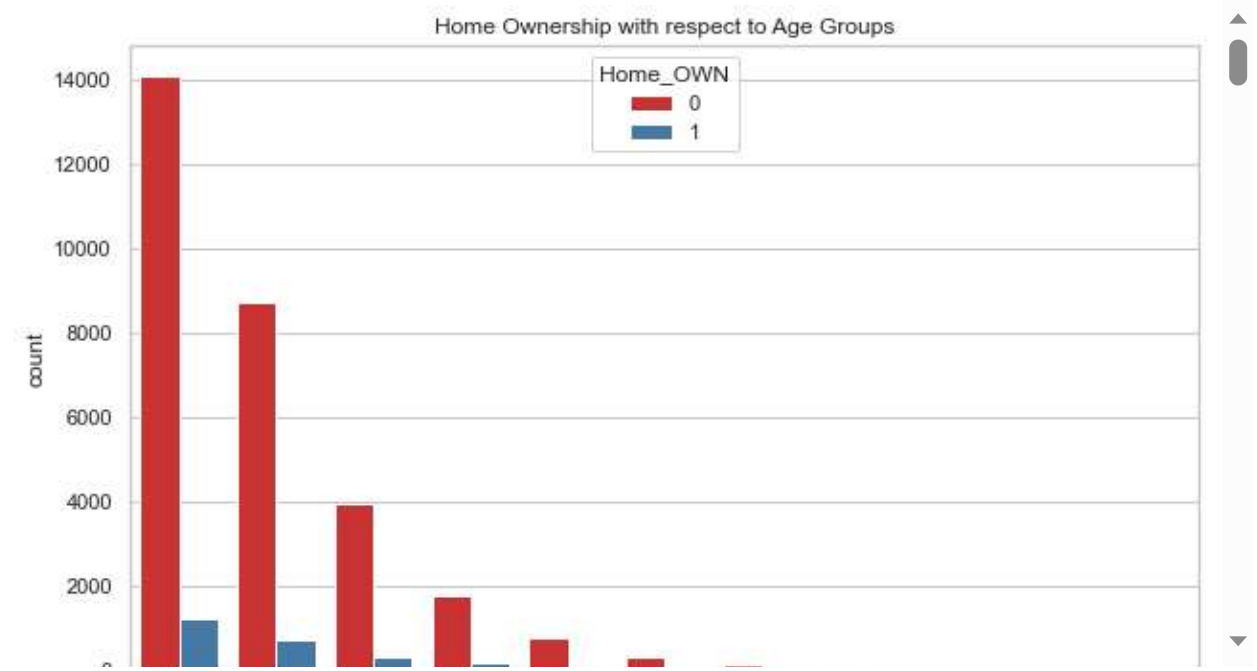




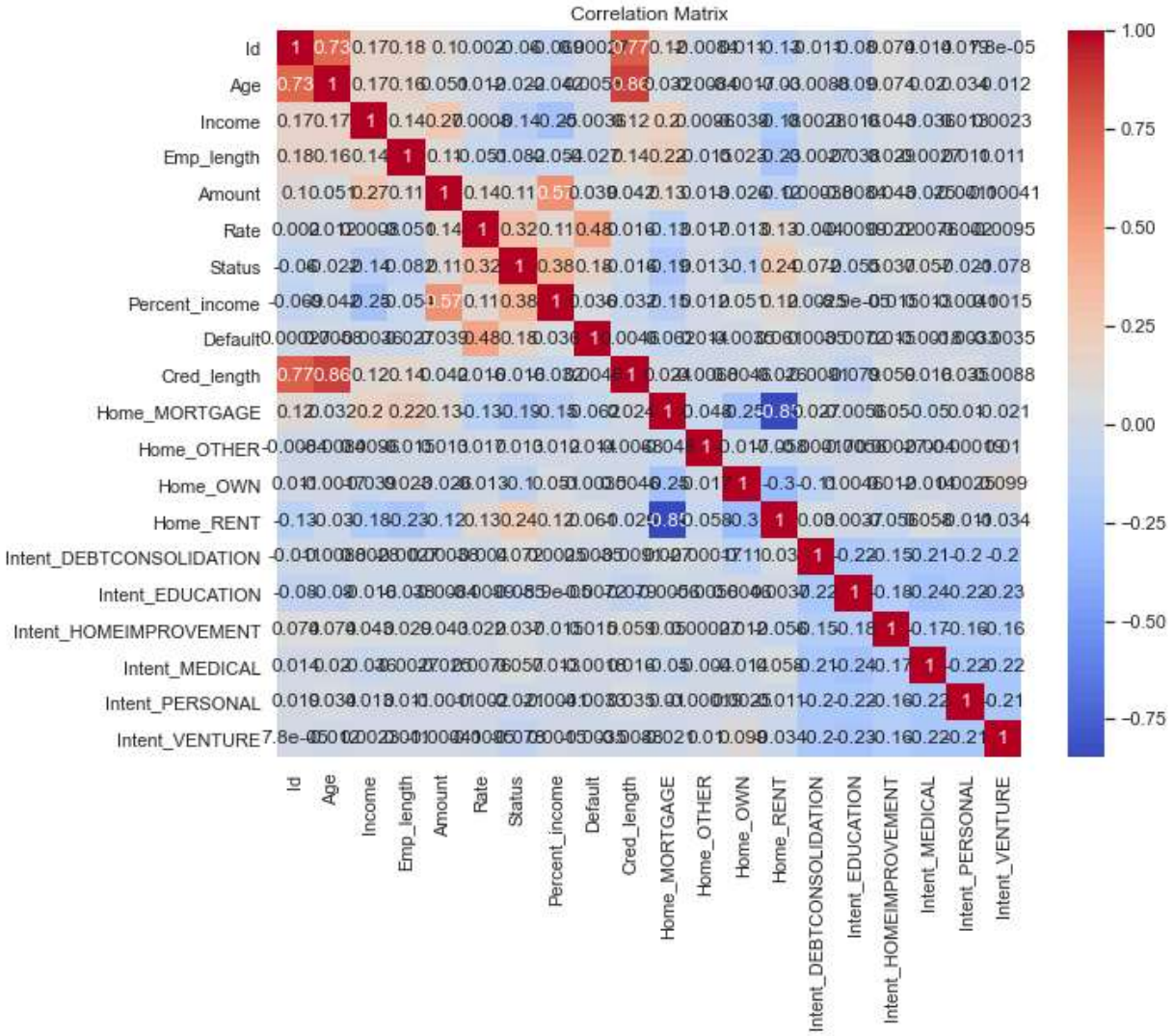
Age group vs Home ownership, Intent and default using the encoded data (df)

In [19]:

```
# Create a copy of the df DataFrame
df2 = df.copy()

# Define the age bins and titles for Home Ownership, Loan Intent, and Previous Lo
age_bins = [20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75]
titles = ["Home Ownership", "Loan Intent", "Default"]

# Create Age Group based on the bins
df2['Age_Group'] = pd.cut(df2['Age'], bins=age_bins)

# List of new columns created by One-Hot Encoding
home_columns = ['Home_OWN', 'Home_RENT', 'Home_OTHER', 'Home_MORTGAGE']
intent_columns = ['Intent_EDUCATION', 'Intent_HOMEIMPROVEMENT', 'Intent_MEDICAL',
default_column = ['Default']

# Combine all categorical columns
new_catcols = home_columns + intent_columns + default_column

# Loop through the categorical columns and generate count plots
t = 0
for col in new_catcols:
    sns.set(style="whitegrid")
    plt.figure(figsize=(10, 6))
    sns.countplot(data=df2, x='Age_Group', hue=col, palette='Set1')
    plt.title(f"{titles[t % len(titles)]} with respect to Age Groups")
    t += 1
```



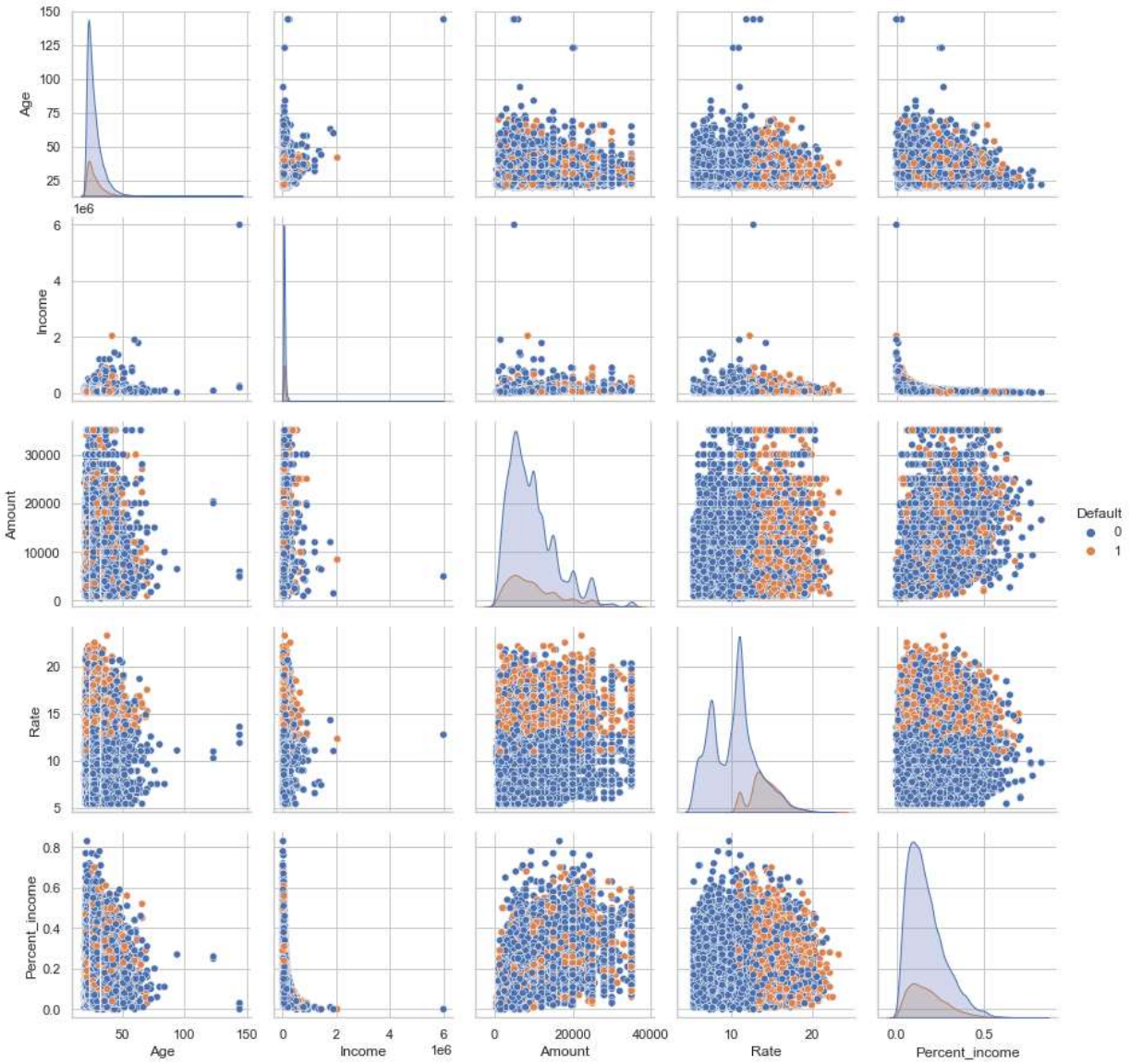
In [20]:
```
# Correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```



Multivariate Analysis

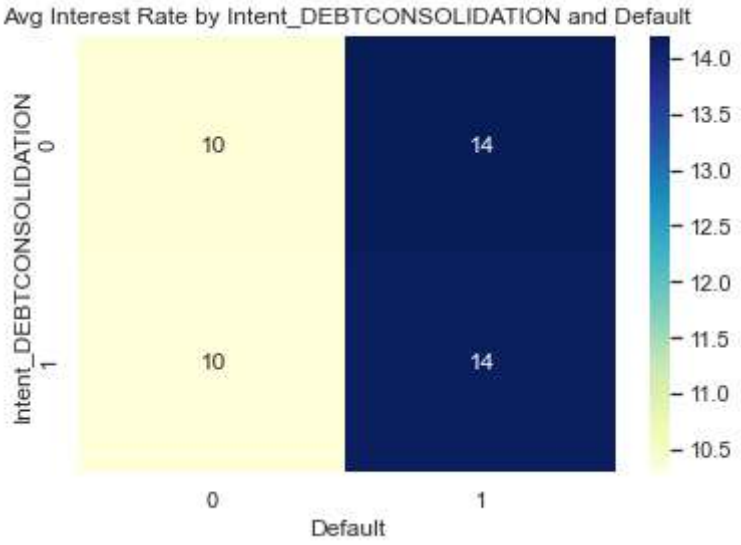
Multivariate analysis entails more than 2 variables interact with each other.

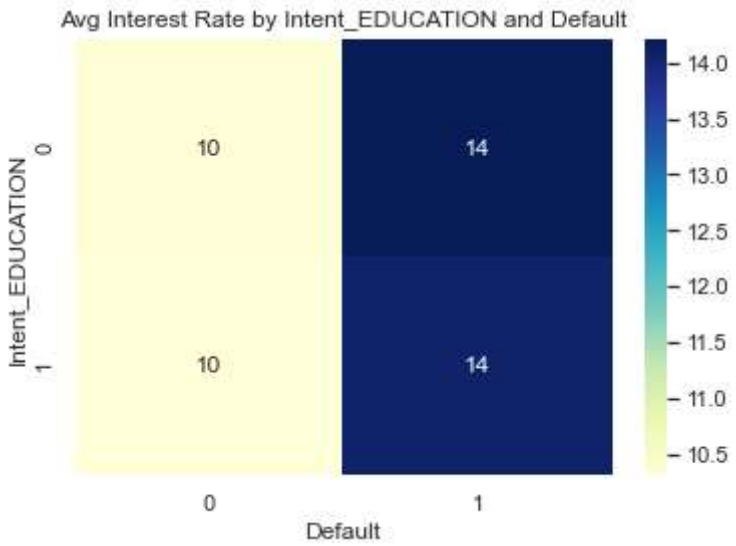
In [21]:
```
sns.pairplot(df[['Age', 'Income', 'Amount', 'Rate', 'Percent_income', 'Default']]
plt.show()
```

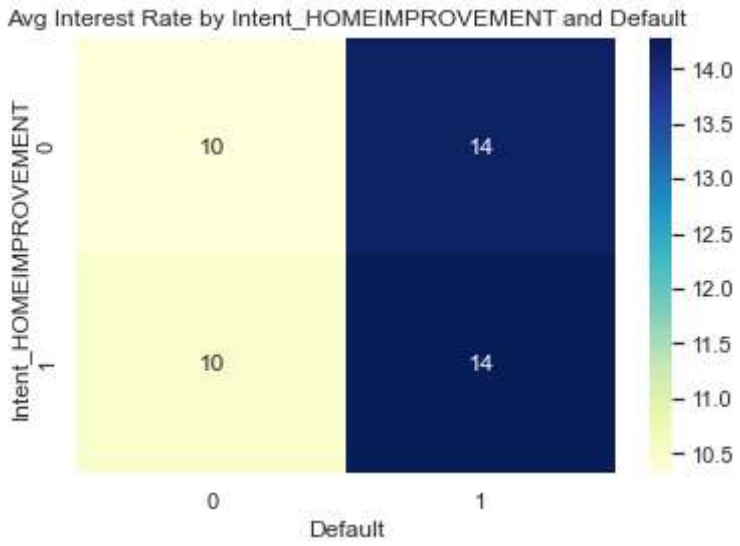


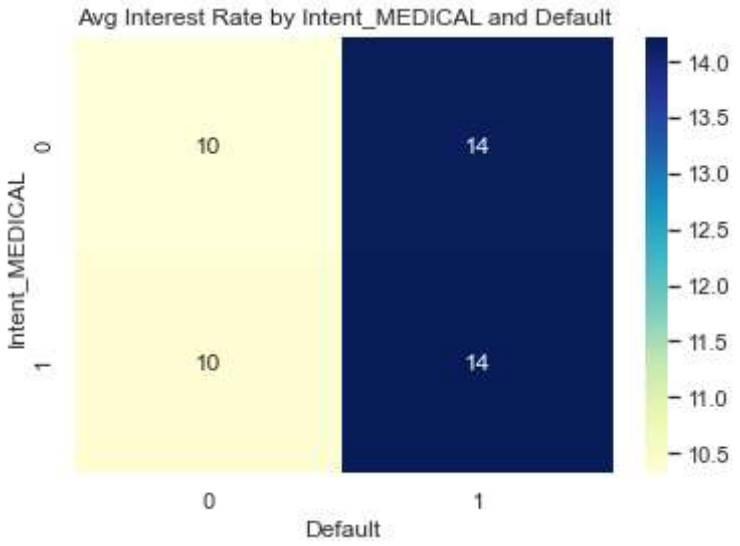
In [22]:
```
home_columns = ['Home_OWN', 'Home_RENT', 'Home_OTHER', 'Home_MORTGAGE']  # Adjust
intent_columns = ['Intent_EDUCATION', 'Intent_HOMEIMPROVEMENT', 'Intent_MEDICAL',
```

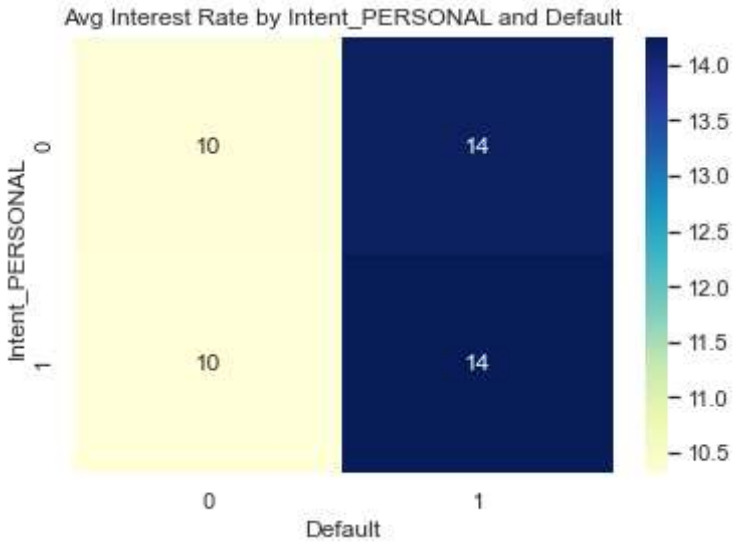
In [23]:
```
intent_columns = [col for col in df.columns if col.startswith('Intent_')]
# Loop through each One-Hot encoded column for Intent and create a heatmap
for intent_col in intent_columns:
    # Create a pivot table with 'Rate' values grouped by the intent column and 'D
    pivot_table = df.pivot_table(values='Rate', index=intent_col, columns='Defaul

    # Create the heatmap
    sns.heatmap(pivot_table, annot=True, cmap='YlGnBu')
    plt.title(f'Avg Interest Rate by {intent_col} and Default')
    plt.show()
```

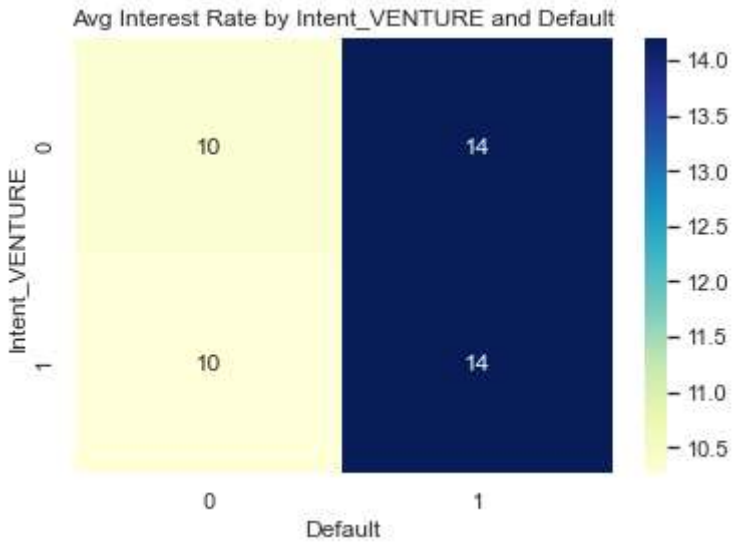





Avg Interest Rate by Intent_HOMEIMPROVEMENT and Default
Intent_HOMEIMPROVEMENT
0
1
10
14
10
14
0
1
Default
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5


Avg Interest Rate by Intent_MEDICAL and Default
Intent_MEDICAL
0
1
10
14
10
14
0
1
Default
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5


Avg Interest Rate by Intent_PERSONAL and Default
Intent_PERSONAL
0
1
10
14
10
14
0
1
Default
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5



PREDICTIVE ANALYSIS AND MODELLING

In this section we will use diffrent machine learning models to build predictive models that will help predict the possibility of default. we will use the following models

- Logistic Regression (baseline)
- Decision Tree
- Random Forest
- Naive Bay

In [24]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_scor
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
```

SPLITTING AND TRANSFORMING OF DATA

In [25]:
```
model_df = df.copy()# copying the data to retain the original df dataframe data
```

In [26]:
```
# Define features and target
X = df.drop("Default", axis=1)
y = df["Default"]
```

In [27]:
```
# Split the data into train and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s
```

In [28]:
```
# Make explicit copies to avoid SettingWithCopyWarning
X_train = X_train.copy()
X_test = X_test.copy()
```

In [29]:
```
# Select numeric columns for scaling
numeric_cols = X.select_dtypes(include=['int64', 'float64']).columns.tolist()
```

In [30]:
```
# Initialize the scaler
scaler = StandardScaler()
```

In [31]:
```
# Fit and transform the numeric columns safely
X_train.loc[:, numeric_cols] = scaler.fit_transform(X_train[numeric_cols])
X_test.loc[:, numeric_cols] = scaler.transform(X_test[numeric_cols])
```

MODELS

1. LOGISTIC REGRETION-BASELINE

In [32]:
```
#initializing the model
lr_model = LogisticRegression()
```

In [33]:
```
#fitting the model usig the training data sets
lr_model.fit(X_train, y_train)
```

Out[33]:
```
LogisticRegression()
```

In [34]:
```
#makingthe y  predictions using the x-test data
y_pred_lr = lr_model.predict(X_test)
print("Logistic Regression Results:\n")
print(confusion_matrix(y_test, y_pred_lr))
print(classification_report(y_test, y_pred_lr))
```

```
Logistic Regression Results:

[[5011  311]
 [ 856  339]]
              precision    recall  f1-score   support

           0       0.85      0.94      0.90      5322
           1       0.52      0.28      0.37      1195

    accuracy                           0.82      6517
   macro avg       0.69      0.61      0.63      6517
weighted avg       0.79      0.82      0.80      6517
```

RESULTS

- The matrix indicates 5011 true negatives which indicates the number of correctly idenifed non defaults and 311 false positives which indicates wrongly predicted non defaults. there are 856 false positives and 339 true positives indicating defaults that was wrongly predicted as non default and correctly predicted defauluts respectfully.

- PRECISION: The model scored 85% and 52 % precision on non-default and default respectively indicating a bias in the model when it comes to predicting non-default
- RECALL: Model has a recall score of 94% and 28 % on non-default and default respectfully.
- FI-SCORE: Model has a score of 90% and 37% on non defaultsand defaults respectively.
- Accuracy: The model has a high accuracy of 82 %

A high precision indicates that when the model forecasts a favorable result, it is likely to be accurate.

A high recall indicates that the model effectively identifies all positive cases, although at the expense of some false positives. The F1-score is the harmonic mean of precision and recall, addressing the question: "How effectively is the model performing, considering both false positives and false negatives?" A high F1-score signifies an effective equilibrium between precision and recall.

Interpretation

Although the model has a high accuracy score of 82% the same is misleading as it has high levels of class imbalance indicated by the recall for defaulters whch is 28% which is dangerously low as it is very important to catch more defaulter even if it is at the cost of falsely predicted non-defaulter(false positive). so we will try another model.

2. DECISION TREE MODEL

In [35]:
```
#initializing the model
dt_model = DecisionTreeClassifier(random_state=42)
```

In [36]:
```
#fitting the model usig the training data sets
dt_model.fit(X_train, y_train)
```

Out[36]:
```
DecisionTreeClassifier(random_state=42)
```

In [37]:
```
#making predictions using the test data
y_pred_dt = dt_model.predict(X_test)
print("Decision Tree Results:\n")
print(confusion_matrix(y_test, y_pred_dt))
print(classification_report(y_test, y_pred_dt))
```

```
Decision Tree Results:

[[4723  599]
 [ 593  602]]
              precision    recall  f1-score   support

           0       0.89      0.89      0.89      5322
           1       0.50      0.50      0.50      1195

    accuracy                           0.82      6517
   macro avg       0.69      0.70      0.70      6517
weighted avg       0.82      0.82      0.82      6517
```

Results

The matrix reveals 4723 true negatives, signifying the accurately recognized non-defaulters, and 599 false positives, representing instances when the model erroneously predicted non-defaults. There are 593 false negatives (defaulters inaccurately classified as non-defaulters) and 602 true positives, indicating accurately identified defaulters.

- PRECISION: The model achieved 89% precision for non-defaulters and 50% for defaulters, signifying more confidence and accuracy in predicting non-defaults compared to defaults.
- The recall rate was 89% for non-defaulters and 50% for defaulters, indicating that the model accurately identified the majority of non-defaulters, while only half of the actual defaulters were correctly recognized.
- The F1-scores were 89% for non-defaulters and 50% for defaulters, indicating a strong performance for non-defaulters but a comparatively poor ability to identify defaulters.
- The model attained an overall accuracy of 82%.

Despite the model's impressive accuracy of 82%, this figure may be deceptive because to class imbalance. The recall for defaulters is 50%, indicating that 50% of true defaulters are overlooked.

3. RANDOM FOREST MODEL

In [38]:
```
#initializing the model
rf_model = RandomForestClassifier(random_state=42, n_estimators=100)
```

In [39]:
```
#fitting the model
rf_model.fit(X_train, y_train)
```

Out[39]:
```
RandomForestClassifier(random_state=42)
```

In [40]:
```
#making predictions
y_pred_rf = rf_model.predict(X_test)
print("Random Forest Results:\n")
print(confusion_matrix(y_test, y_pred_rf))
print(classification_report(y_test, y_pred_rf))
```

```
Random Forest Results:

[[4921  401]
 [ 703  492]]
              precision    recall  f1-score   support

           0       0.88      0.92      0.90      5322
           1       0.55      0.41      0.47      1195

    accuracy                           0.83      6517
   macro avg       0.71      0.67      0.69      6517
weighted avg       0.82      0.83      0.82      6517
```

Results

The Random Forest model attained an impressive accuracy of 83%; nevertheless, this conceals a problem with class imbalance. It accurately identified 4921 non-defaulters (True Negatives) and 492 defaulters (True Positives), but failed to detect 703 true defaulters (False Negatives) and incorrectly classified 401 non-defaulters as defaulters (False Positives). The precision for non-

defaulters was 88%, however for defaulters it decreased to 55%, indicating a bias towards the majority class. The recall for non-defaulters was 92%, however it was just 41% for defaulters, suggesting the model had difficulty identifying defaulters. The model has an F1-score of 90% for non-defaulters and 47% for defaulters, indicating potential underfitting of the minority class. To enhance performance, the exploration of SMOTE for class balance, feature engineering, and hyperparameter optimization is recommended.

4. NAIVE BAYES MODEL

In [41]:
```
#initiliazing the model
nb_model = GaussianNB()
```

In [42]:
```
# fittong the model with training data
nb_model.fit(X_train, y_train)
```

Out[42]:
```
GaussianNB()
```

In [43]:
```
# Making predictions
y_pred_nb = nb_model.predict(X_test)
print("Naive Bayes Results:\n")
print(confusion_matrix(y_test, y_pred_nb))
print(classification_report(y_test, y_pred_nb))
```

```
Naive Bayes Results:

[[4854  468]
 [ 761  434]]
              precision    recall  f1-score   support

           0       0.86      0.91      0.89      5322
           1       0.48      0.36      0.41      1195

    accuracy                           0.81      6517
   macro avg       0.67      0.64      0.65      6517
weighted avg       0.79      0.81      0.80      6517
```

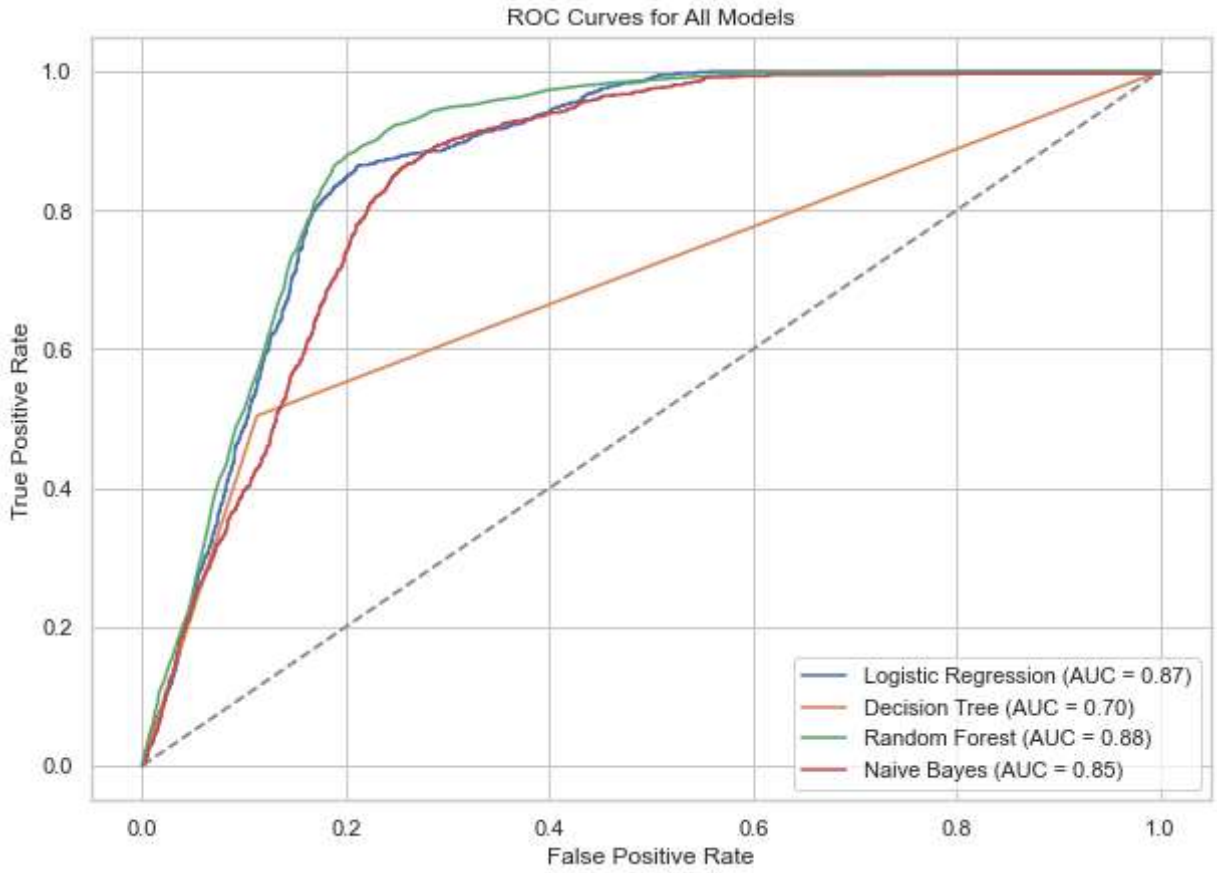
Results

The Naive Bayes model had an accuracy of 81%, although its efficacy regarding defaulters was subpar, achieving just 36% recall for defaulters and 91% recall for non-defaulters. It correctly identified 4,854 non-defaulters (True Negatives) and 434 defaulters (True Positives), while erroneously identifying 761 genuine defaulters as non-defaulters (False Negatives) and 468 non-defaulters as defaulters (False Positives). The precision for non-defaulters was 86%, whereas for defaulters, it was 48%. This model has an F1-score of 89% for non-defaulters and 41% for defaulters, indicating underfitting in the defaulter class. To enhance performance, it is advisable to employ SMOTE to rectify class imbalance, engage in feature engineering, and investigate more adaptable models to more effectively capture the intricate relationships within the data.

MODELS EVALUATION USING ROC-AUC SCORE AND ROC CURVE

In [44]:
```
from sklearn.metrics import roc_auc_score, roc_curve
import matplotlib.pyplot as plt

# Get predicted probabilities for the positive class (class = 1)
lr_probs = lr_model.predict_proba(X_test)[:, 1]
dt_probs = dt_model.predict_proba(X_test)[:, 1]
rf_probs = rf_model.predict_proba(X_test)[:, 1]
nb_probs = nb_model.predict_proba(X_test)[:, 1]

# Calculate ROC-AUC scores
lr_auc = roc_auc_score(y_test, lr_probs)
dt_auc = roc_auc_score(y_test, dt_probs)
rf_auc = roc_auc_score(y_test, rf_probs)
nb_auc = roc_auc_score(y_test, nb_probs)

# Print AUC Scores
print(f"Logistic Regression AUC: {lr_auc:.3f}")
print(f"Decision Tree AUC: {dt_auc:.3f}")
print(f"Random Forest AUC: {rf_auc:.3f}")
print(f"Naive Bayes AUC: {nb_auc:.3f}")

# Plot ROC Curves
fpr1, tpr1, _ = roc_curve(y_test, lr_probs)
fpr2, tpr2, _ = roc_curve(y_test, dt_probs)
fpr3, tpr3, _ = roc_curve(y_test, rf_probs)
fpr4, tpr4, _ = roc_curve(y_test, nb_probs)

plt.figure(figsize=(10, 7))
plt.plot(fpr1, tpr1, label=f'Logistic Regression (AUC = {lr_auc:.2f})')
plt.plot(fpr2, tpr2, label=f'Decision Tree (AUC = {dt_auc:.2f})')
plt.plot(fpr3, tpr3, label=f'Random Forest (AUC = {rf_auc:.2f})')
plt.plot(fpr4, tpr4, label=f'Naive Bayes (AUC = {nb_auc:.2f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')  # Diagonal line
plt.title('ROC Curves for All Models')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

```
Logistic Regression AUC: 0.870
Decision Tree AUC: 0.696
Random Forest AUC: 0.884
Naive Bayes AUC: 0.847
```



Results

1. Logistic Regression AUC: 0.870: Very good score. The model can discriminate defaulters and non-defaulters well with an AUC of 0.87. Most positive cases (defaulters) rank higher than negative ones. This implies that despite low default recall, the model can distinguish the two groups well.
   B. Decision Tree AUC: 0.696: A moderate score. AUC < 0.7 suggests the model is underfitting or oversensitive to data and struggles to differentiate classes. This matches past findings that it only finds 50% of defaulters. Pruning, feature selection, and boosting may help.
   C. Random Forest AUC: 0.884: This model has the highest AUC. It discriminates well and captures more complicated patterns than logistic regression or a decision tree. This makes it rank true positives higher than false positives.
   D. Naive Bayes AUC: 0.847: The model distinguishes classes well with an AUC of 0.85. In this metric, Random Forest and Logistic Regression outperform it.

Based on the false positive rate (x-axis) and the true positive rate (y-axis), the ROC graphic contrasts four classifiers. Optimal discrimination is indicated by the Random Forest curve's closeness to the top-left corner. There are notable trade-offs with logistic regression and naive bayes. The Decision Tree performs poorly, more closely matching the diagonal line denoting random guessing. A higher AUC shows better general categorization performance. The curve shows the false positive rate for every model in relation to the genuine positive rate. With the largest area under the curve (AUC = 0.88), Random Forest shows strong categorization

performance. With AUCs of about 0.87 and 0.85, respectively, Logistic Regression and Naive

COMPARISON OF THE 4 MODELS

Random Forest emerged as the superior model among the four, achieving the highest AUC score (0.884), demonstrating commendable precision and recall for both classes, and exhibiting balanced F1-scores—signifying robust classification proficiency and an enhanced ability to address class imbalance. Despite its subpar recall for defaulters at 28%, signifying a failure to identify several genuine defaults, Logistic Regression achieved a commendable AUC of 0.870 and a high accuracy of 82%. Despite achieving a 36% recall and a 41% F1-score for defaulters, Naive Bayes yielded satisfactory results (AUC: 0.847). The Decision Tree had an AUC of 0.696 and a recall of only 50% for defaulters, indicating it was the least effective model, either due to underfitting or susceptibility to noise. All models demonstrated satisfactory overall accuracy; however, Random Forest and Logistic Regression had superior discrimination power, with Random Forest being the most reliable for default prediction. SMOTE, feature engineering, and model ensemble techniques may yield additional advantages.

In [45]:
```
model_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32581 entries, 0 to 32580
Data columns (total 20 columns):
 #   Column                     Non-Null Count  Dtype  
---  ------                     --------------  -----  
 0   Id                         32581 non-null  int64  
 1   Age                        32581 non-null  int64  
 2   Income                     32581 non-null  int64  
 3   Emp_length                 32581 non-null  float64
 4   Amount                     32581 non-null  int64  
 5   Rate                       32581 non-null  float64
 6   Status                     32581 non-null  int64  
 7   Percent_income             32581 non-null  float64
 8   Default                    32581 non-null  int64  
 9   Cred_length                32581 non-null  int64  
 10  Home_MORTGAGE              32581 non-null  uint8  
 11  Home_OTHER                 32581 non-null  uint8  
 12  Home_OWN                   32581 non-null  uint8  
 13  Home_RENT                  32581 non-null  uint8  
 14  Intent_DEBTCONSOLIDATION   32581 non-null  uint8  
 15  Intent_EDUCATION           32581 non-null  uint8  
 16  Intent_HOMEIMPROVEMENT     32581 non-null  uint8  
 17  Intent_MEDICAL             32581 non-null  uint8  
 18  Intent_PERSONAL            32581 non-null  uint8  
 19  Intent_VENTURE             32581 non-null  uint8  
dtypes: float64(3), int64(7), uint8(10)
memory usage: 2.8 MB
```

In [46]:
```
print("Cred_length range:", df['Cred_length'].min(), "-", df['Cred_length'].max()
```

```
Cred_length range: 2 - 30
```

In [47]:
```
print("Income range:", df['Income'].min(), "-", df['Income'].max())
```

```
Income range: 4000 - 6000000
```

FEATURE ENGINEERING AND SMOTE APPLICATION

In [ ]:

In [73]:
```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import KBinsDiscretizer
```

Feature Engineering

In [74]:
```
# Dropping 'Id' as it's not useful for modeling
df1 = model_df.drop(columns=['Id'])
```

In [75]:
```
df1.head()
```

Out[75]:

	Age	Income	Emp_length	Amount	Rate	Status	Percent_income	Default	Cred_length	Home_
0	22	59000	123.0	35000	16.02	1	0.59	1	3	
1	21	9600	5.0	1000	11.14	0	0.10	0	2	
2	25	9600	1.0	5500	12.87	1	0.57	0	3	
3	23	65500	4.0	35000	15.23	1	0.53	0	2	
4	24	54400	8.0	35000	14.27	1	0.55	1	4	

In [78]:
```
# Creating Age bins and labels
age_bins = [20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75]
age_labels = ['20-25', '25-30', '30-35', '35-40', '40-45', '45-50', '50-55', '55-

# Creating Income bins and labels
income_bins = [4000, 50000, 200000, 1000000, 6000000]
income_labels = ['Low', 'Medium', 'High', 'Very High']

# Creating Credit Length bins and labels
cred_bins = [1, 6, 13, 20, 30]
cred_labels = ['Short', 'Moderate', 'Long', 'Very Long']

# Applying binning to the respective columns
df1['Age_bin'] = pd.cut(df['Age'], bins=age_bins, labels=age_labels, right=True)
df1['Income_bin'] = pd.cut(df['Income'], bins=income_bins, labels=income_labels,
df1['Cred_len_bin'] = pd.cut(df['Cred_length'], bins=cred_bins, labels=cred_label

# Checking the result
df1[['Age', 'Age_bin', 'Income', 'Income_bin', 'Cred_length', 'Cred_len_bin']].he
```

Out[78]:

	Age	Age_bin	Income	Income_bin	Cred_length	Cred_len_bin
0	22	20-25	59000	Medium	3	Short
1	21	20-25	9600	Low	2	Short
2	25	20-25	9600	Low	3	Short
3	23	20-25	65500	Medium	2	Short
4	24	20-25	54400	Medium	4	Short

In [79]:
```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32581 entries, 0 to 32580
Data columns (total 22 columns):
 #   Column                      Non-Null Count  Dtype   
---  ------                      --------------  -----   
 0   Age                         32581 non-null  int64   
 1   Income                      32581 non-null  int64   
 2   Emp_length                  32581 non-null  float64 
 3   Amount                      32581 non-null  int64   
 4   Rate                        32581 non-null  float64 
 5   Status                      32581 non-null  int64   
 6   Percent_income              32581 non-null  float64 
 7   Default                     32581 non-null  int64   
 8   Cred_length                 32581 non-null  int64   
 9   Home_MORTGAGE               32581 non-null  uint8   
 10  Home_OTHER                  32581 non-null  uint8   
 11  Home_OWN                    32581 non-null  uint8   
 12  Home_RENT                   32581 non-null  uint8   
 13  Intent_DEBTCONSOLIDATION    32581 non-null  uint8   
 14  Intent_EDUCATION            32581 non-null  uint8   
 15  Intent_HOMEIMPROVEMENT      32581 non-null  uint8   
 16  Intent_MEDICAL              32581 non-null  uint8   
 17  Intent_PERSONAL             32581 non-null  uint8   
 18  Intent_VENTURE              32581 non-null  uint8   
 19  Age_bin                     32556 non-null  category
 20  Income_bin                  32580 non-null  category
 21  Cred_len_bin                32581 non-null  category
dtypes: category(3), float64(3), int64(6), uint8(10)
memory usage: 2.6 MB
```

In [80]:
```
# Step 4: One-Hot Encode the bin columns
df1 = pd.get_dummies(df1, columns=['Age_bin', 'Income_bin', 'Cred_len_bin'], drop

# Step 5: Done! df1 now contains Affordability + bins
df1.head()
```

Out[80]:

	Age	Income	Emp_length	Amount	Rate	Status	Percent_income	Default	Cred_length	Home_
0	22	59000	123.0	35000	16.02	1	0.59	1	3	
1	21	9600	5.0	1000	11.14	0	0.10	0	2	
2	25	9600	1.0	5500	12.87	1	0.57	0	3	
3	23	65500	4.0	35000	15.23	1	0.53	0	2	
4	24	54400	8.0	35000	14.27	1	0.55	1	4	

5 rows × 35 columns

In [81]:
```
# Identify duplicate columns
print(df1.columns[df1.columns.duplicated()])
```

```
Index([], dtype='object')
```

In [82]:
```
# Remove duplicate columns
df1 = df1.loc[:, ~df1.columns.duplicated()]
print(df1.columns)
```

```
Index(['Age', 'Income', 'Emp_length', 'Amount', 'Rate', 'Status',
       'Percent_income', 'Default', 'Cred_length', 'Home_MORTGAGE',
       'Home_OTHER', 'Home_OWN', 'Home_RENT', 'Intent_DEBTCONSOLIDATION',
       'Intent_EDUCATION', 'Intent_HOMEIMPROVEMENT', 'Intent_MEDICAL',
       'Intent_PERSONAL', 'Intent_VENTURE', 'Age_bin_25-30', 'Age_bin_30-35',
       'Age_bin_35-40', 'Age_bin_40-45', 'Age_bin_45-50', 'Age_bin_50-55',
       'Age_bin_55-60', 'Age_bin_60-65', 'Age_bin_65-70', 'Age_bin_70-75',
       'Income_bin_Medium', 'Income_bin_High', 'Income_bin_Very High',
       'Cred_len_bin_Moderate', 'Cred_len_bin_Long', 'Cred_len_bin_Very Long'],
      dtype='object')
```

Adding affordability ratio, Creit length to income ratio and debt to income ratio

The debt-to-income ratio is a common financial metric used to assess an individual's debt relative to their income. It is often employed in financial decision-making, especially about loans. The credit length to income ratio may indicate the duration of an individual's credit history relative to their income. The affordability ratio evaluates the financial strain a loan places on a borrower's income.

In [85]:
```
#affordability ratio
df1['Affordability'] = df1['Amount'] / df1['Income']
```

In [87]:
```
#credit length-income ratio
df1['CredLen_to_Income'] = df1['Cred_length'] / df1['Income']
```

In [88]:
```
#debt-to-income ratio
df1['Debt_to_Income'] = df1['Amount'] / df1['Income']
```

In [89]:
```
print(df1.columns)# checking the succeffull addition of new columns
```

```
Index(['Age', 'Income', 'Emp_length', 'Amount', 'Rate', 'Status',
       'Percent_income', 'Default', 'Cred_length', 'Home_MORTGAGE',
       'Home_OTHER', 'Home_OWN', 'Home_RENT', 'Intent_DEBTCONSOLIDATION',
       'Intent_EDUCATION', 'Intent_HOMEIMPROVEMENT', 'Intent_MEDICAL',
       'Intent_PERSONAL', 'Intent_VENTURE', 'Age_bin_25-30', 'Age_bin_30-35',
       'Age_bin_35-40', 'Age_bin_40-45', 'Age_bin_45-50', 'Age_bin_50-55',
       'Age_bin_55-60', 'Age_bin_60-65', 'Age_bin_65-70', 'Age_bin_70-75',
       'Income_bin_Medium', 'Income_bin_High', 'Income_bin_Very High',
       'Cred_len_bin_Moderate', 'Cred_len_bin_Long', 'Cred_len_bin_Very Long',
       'Affordability', 'CredLen_to_Income', 'Debt_to_Income'],
      dtype='object')
```

Synthetic Minority Over-sampling Technique(SMOTE) Application

This step is necessary to deal with data imbalance

In [90]:
```
from imblearn.over_sampling import SMOTE
```

In [91]:
```
#defining the features
X = df1.drop('Default', axis=1)  # Features
y = df1['Default']  # Target
```

In [92]:
```
# Applying SMOTE to balance the classes
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
```

In [94]:
```
# Checking the shape of the resampled data
print(f"Original shape: {X.shape}, Resampled shape: {X_resampled.shape}")
```

```
Original shape: (32581, 37), Resampled shape: (53672, 37)
```

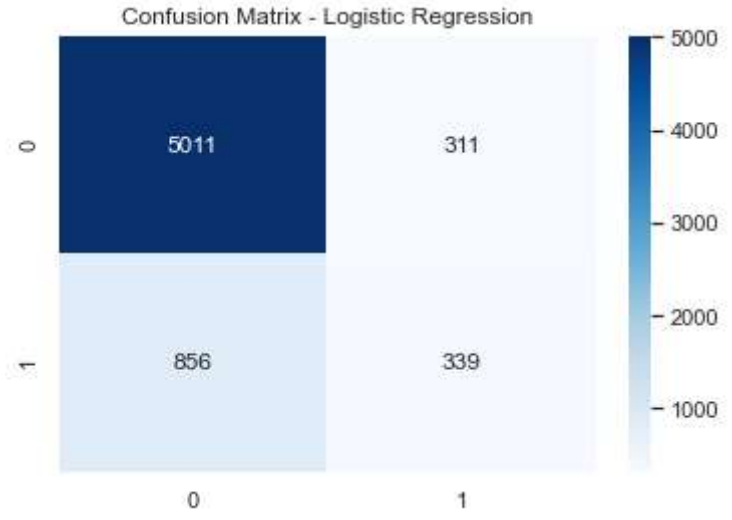
Retraining and remodelling all 4 models

1. Logistic regression

In [97]:
```
# Initialize
lr_model1 = LogisticRegression(max_iter=1000, random_state=42)

# Train
lr_model1.fit(X_train, y_train)

y_pred_lr1 = lr_model1.predict(X_test)
print("Logistic Regression Results:\n")
print(confusion_matrix(y_test, y_pred_lr1))
print(classification_report(y_test, y_pred_lr1))

# Confusion Matrix
sns.heatmap(confusion_matrix(y_test, y_pred_lr1), annot=True, fmt='d', cmap='Blue
plt.title("Confusion Matrix - Logistic Regression")
```

```
Logistic Regression Results:

[[5011  311]
 [ 856  339]]
              precision    recall  f1-score   support

           0       0.85      0.94      0.90      5322
           1       0.52      0.28      0.37      1195

    accuracy                           0.82      6517
   macro avg       0.69      0.61      0.63      6517
weighted avg       0.79      0.82      0.80      6517
```

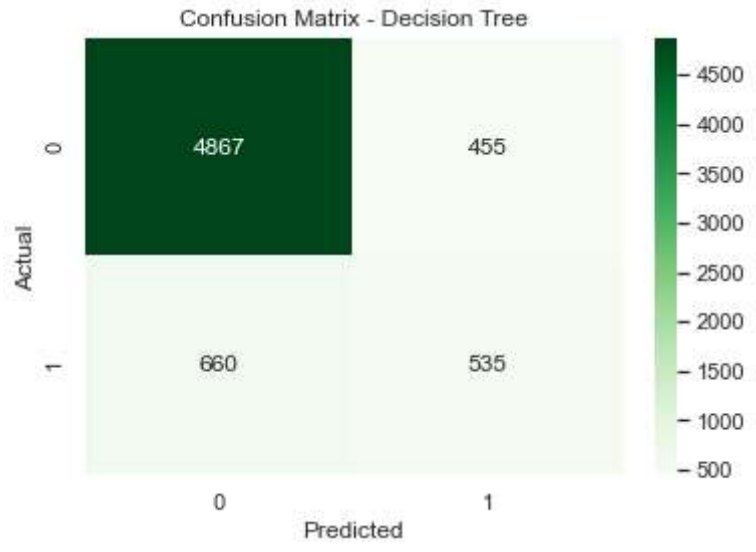
Out[97]:
```
Text(0.5, 1.0, 'Confusion Matrix - Logistic Regression')
```



2. Decision Tree Model

In [98]:
```
from sklearn.tree import DecisionTreeClassifier

# Initialize with pruning
dt_model1 = DecisionTreeClassifier(max_depth=5, random_state=42)

# Train
dt_model1.fit(X_train, y_train)
#making predictions using the test data
y_pred_dt1 = dt_model1.predict(X_test)
print("Decision Tree Results:\n")
print(confusion_matrix(y_test, y_pred_dt1))
print(classification_report(y_test, y_pred_dt1))

# Confusion Matrix
sns.heatmap(confusion_matrix(y_test, y_pred_dt1), annot=True, fmt='d', cmap='Gree
plt.title("Confusion Matrix - Decision Tree")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

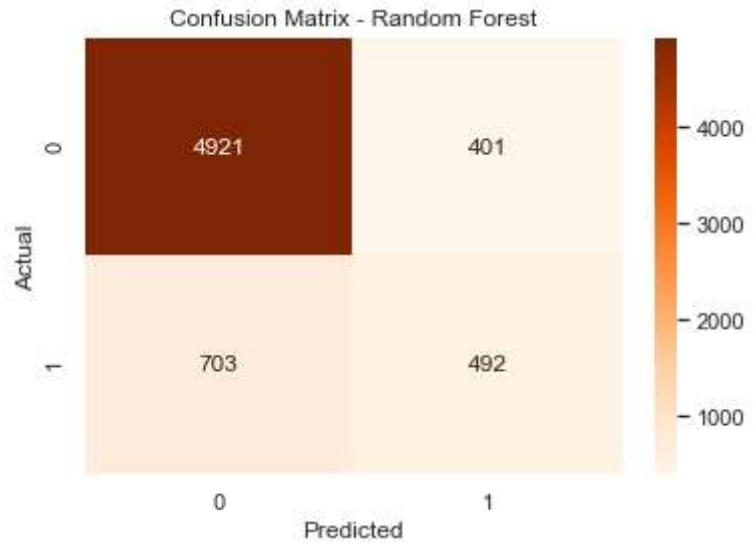
```
Decision Tree Results:

[[4867  455]
 [ 660  535]]
              precision    recall  f1-score   support

           0       0.88      0.91      0.90      5322
           1       0.54      0.45      0.49      1195

    accuracy                           0.83      6517
   macro avg       0.71      0.68      0.69      6517
weighted avg       0.82      0.83      0.82      6517
```



3. Random Forest Mode

In [100]:
```
#initializing the model
rf_model1 = RandomForestClassifier(random_state=42, n_estimators=100)
#fitting the model
rf_model1.fit(X_train, y_train)
#making predictions
y_pred_rf1 = rf_model1.predict(X_test)
print("Random Forest Results:\n")
print(confusion_matrix(y_test, y_pred_rf1))
print(classification_report(y_test, y_pred_rf1))

# Confusion Matrix
sns.heatmap(confusion_matrix(y_test, y_pred_rf1), annot=True, fmt='d', cmap='Orar
plt.title("Confusion Matrix - Random Forest")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

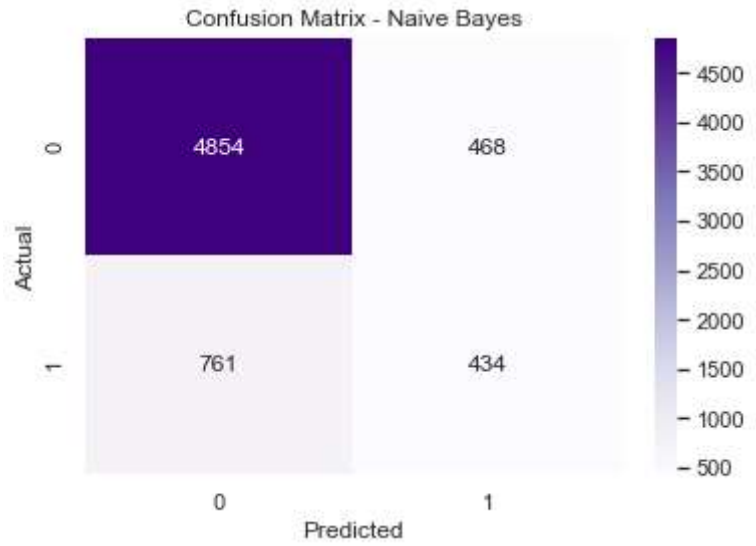
```
Random Forest Results:

[[4921  401]
 [ 703  492]]
              precision    recall  f1-score   support

           0       0.88      0.92      0.90      5322
           1       0.55      0.41      0.47      1195

    accuracy                           0.83      6517
   macro avg       0.71      0.67      0.69      6517
weighted avg       0.82      0.83      0.82      6517
```



4. Naive Bayes Model

In [101]:
```
#initiliazing the model
nb_model1 = GaussianNB()
# fittong the model with training data
nb_model1.fit(X_train, y_train)
# Making predictions
y_pred_nb1 = nb_model1.predict(X_test)
print("Naive Bayes Results:\n")
print(confusion_matrix(y_test, y_pred_nb1))
print(classification_report(y_test, y_pred_nb1))

# Confusion Matrix
sns.heatmap(confusion_matrix(y_test, y_pred_nb1), annot=True, fmt='d', cmap='Purp
plt.title("Confusion Matrix - Naive Bayes")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

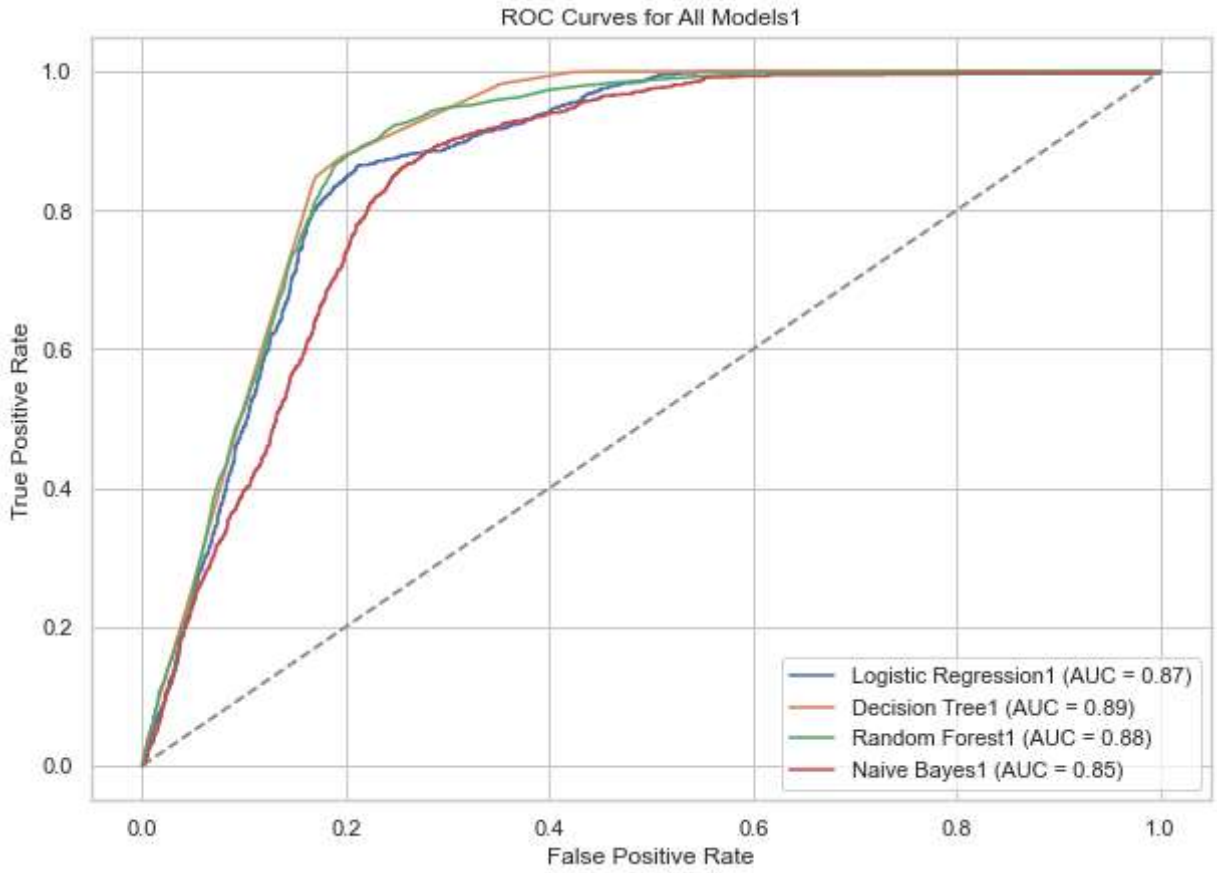
```
Naive Bayes Results:

[[4854  468]
 [ 761  434]]
              precision    recall  f1-score   support

           0       0.86      0.91      0.89      5322
           1       0.48      0.36      0.41      1195

    accuracy                           0.81      6517
   macro avg       0.67      0.64      0.65      6517
weighted avg       0.79      0.81      0.80      6517
```



ROC-AUC SCORE AND ROC CURVE MODELS EVALUATION

In [102]:
```
# Get predicted probabilities for the positive class (class = 1)
lr_probs1 = lr_model1.predict_proba(X_test)[:, 1]
dt_probs1 = dt_model1.predict_proba(X_test)[:, 1]
rf_probs1 = rf_model1.predict_proba(X_test)[:, 1]
nb_probs1 = nb_model1.predict_proba(X_test)[:, 1]

# Calculate ROC-AUC scores
lr_auc1 = roc_auc_score(y_test, lr_probs1)
dt_auc1 = roc_auc_score(y_test, dt_probs1)
rf_auc1 = roc_auc_score(y_test, rf_probs1)
nb_auc1 = roc_auc_score(y_test, nb_probs1)

# Print AUC Scores
print(f"Logistic Regression AUC: {lr_auc1:.3f}")
print(f"Decision Tree AUC: {dt_auc1:.3f}")
print(f"Random Forest AUC: {rf_auc1:.3f}")
print(f"Naive Bayes AUC: {nb_auc1:.3f}")

# Plot ROC Curves
fpr11, tpr11, _ = roc_curve(y_test, lr_probs1)
fpr22, tpr22, _ = roc_curve(y_test, dt_probs1)
fpr33, tpr33, _ = roc_curve(y_test, rf_probs1)
fpr44, tpr44, _ = roc_curve(y_test, nb_probs1)

plt.figure(figsize=(10, 7))
plt.plot(fpr11, tpr11, label=f'Logistic Regression1 (AUC = {lr_auc1:.2f})')
plt.plot(fpr22, tpr22, label=f'Decision Tree1 (AUC = {dt_auc1:.2f})')
plt.plot(fpr33, tpr33, label=f'Random Forest1 (AUC = {rf_auc1:.2f})')
plt.plot(fpr44, tpr44, label=f'Naive Bayes1 (AUC = {nb_auc1:.2f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')  # Diagonal line
plt.title('ROC Curves for All Models1')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

```
Logistic Regression AUC: 0.870
Decision Tree AUC: 0.889
Random Forest AUC: 0.884
Naive Bayes AUC: 0.847
```



RESULTS

According to the results, all four models demonstrate satisfactory performance in predicting default risk; however, tree-based methods are particularly notable. Logistic Regression exhibits excellent accuracy (82%) and robust precision for non-defaulters (0.85); however, its recall for defaulters is low (0.28), signifying a significant number of genuine defaults are overlooked. Naive Bayes has comparable performance with a little improved recall of 0.36, although remains inadequate in identifying defaulters. Both the Decision Tree and Random Forest exhibit enhanced balance, demonstrating improved recall for class 1 (0.45 and 0.41, respectively) and elevated total F1-scores for the positive class. Despite comparable accuracy (83%), Random Forest surpasses in stability and generalization through the aggregation of numerous trees, whereas Decision Tree exhibits a marginally superior AUC (0.89 vs. 0.884). Based on the classification metrics and ROC-AUC scores, the Decision Tree model is the most effective in this analysis. It optimally balances accuracy, interpretability, and the capacity to identify defaults, rendering it the most appropriate option for predicting loan default risk in this scenario.

IMplications of feature engineering, SMOTE application and pruning on the models

The implementation of SMOTE, feature engineering, and pruning markedly enhanced model performance relative to the baseline. Initially, the Random Forest model outperformed others, attaining an AUC of 0.884 and demonstrating robust overall classification; however, all models encountered difficulties in identifying defaulters, particularly the Decision Tree, which achieved an AUC of 0.696. Following modifications, the Decision Tree's AUC significantly increased to 0.89, accompanied by better recall (0.45) and F1-score, establishing it as the premier model. Random Forest exhibited robust performance, characterized by a more refined ROC curve, indicating

superior generalization. Feature engineering generated additional useful attributes, such as the affordability ratio, enhancing the algorithms' capacity to identify risk trends. SMOTE mitigated class imbalance, enhancing the models' sensitivity to defaulters, especially for the Decision Tree and Random Forest algorithms. Pruning the Decision Tree mitigated overfitting, resulting in enhanced generalization and stability. Overall, improvements significantly favored tree-based models, elevating Decision Tree from the least effective to the most proficient model for default prediction, while preserving the performance of Logistic Regression and Naive Bayes.

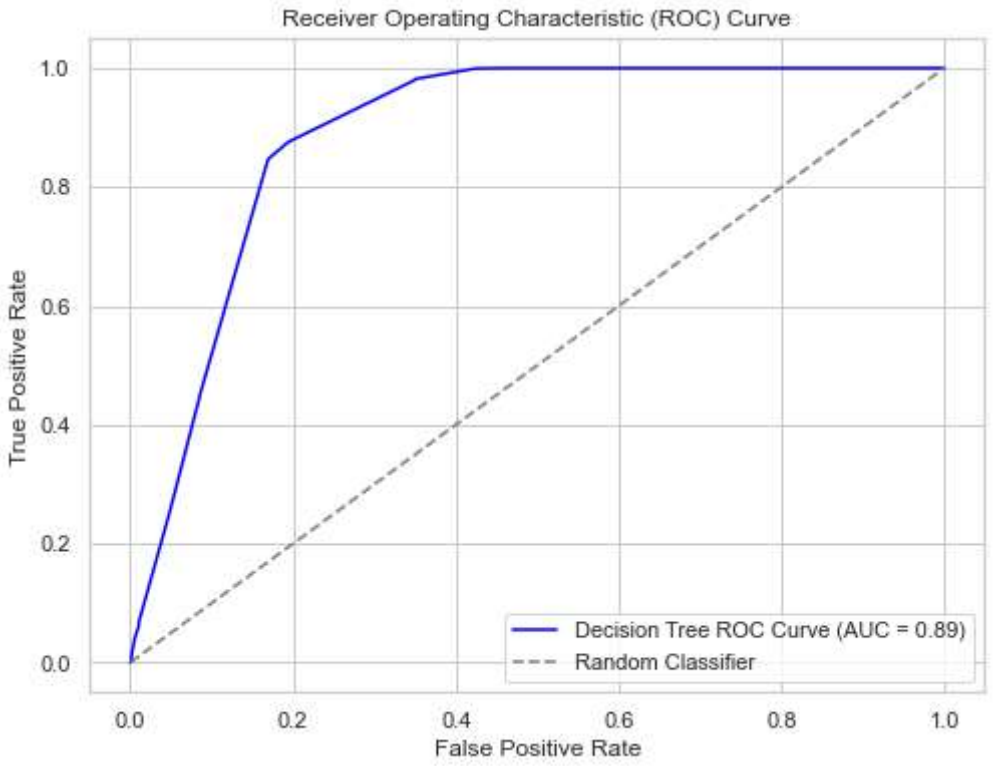
DECISION TREE MODEL VALIDATION

After identifying the decision tree as the best predictive mdel for this study, we will go ahead and validate the model using our testing data.

In [107]:

```
# MODEL VALIDATION
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_scor
import matplotlib.pyplot as plt

# Initialize with pruning
dt_model1 = DecisionTreeClassifier(max_depth=5, random_state=42)

#Performing cross-validation (evaluate model on different subsets of data)
cv_scores = cross_val_score(dt_model1, X_train, y_train, cv=5, scoring='accuracy'
print(f"Cross-validation accuracy scores: {cv_scores}")
print(f"Mean cross-validation accuracy: {cv_scores.mean()}")

#Training the model on the entire training data
dt_model1.fit(X_train, y_train)

#Predicting on the test data
y_pred_dt1 = dt_model1.predict(X_test)
y_pred_prob1 = dt_model1.predict_proba(X_test)[:, 1]  # Probability scores for RO

#Evaluating model performance
print("Classification Report:")
print(classification_report(y_test, y_pred_dt1))

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_dt1))

# Calculate ROC-AUC score
roc_auc1 = roc_auc_score(y_test, y_pred_prob1)
print(f"ROC-AUC score: {roc_auc1}")

# Plotting the  ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob1)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'Decision Tree ROC Curve (AUC = {roc_auc1
plt.plot([0, 1], [0, 1], color='gray', linestyle='--', label='Random Classifier')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
Cross-validation accuracy scores: [0.83004028 0.83675427 0.83272588 0.82505275
0.83288565]
Mean cross-validation accuracy: 0.8314917661863996
Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.91      0.90      5322
           1       0.54      0.45      0.49      1195

    accuracy                           0.83      6517
   macro avg       0.71      0.68      0.69      6517
weighted avg       0.82      0.83      0.82      6517

Confusion Matrix:
[[4867  455]
 [ 660  535]]
ROC-AUC score: 0.8889589907842869
```



from the results, the Decision Tree model exhibits robust and consistent performance, with cross-validation accuracy scores fluctuating between 82.5% and 83.7%, and an average cross-validation accuracy of 83.15%. The consistency across several data splits underscores the model's dependability in practical applications. The classification report indicates the model's exceptional capacity to accurately identify non-default consumers, attaining a precision of 88% and a remarkable recall of 91% for non-defaults. The overall F1-score of 0.90 for this class signifies a robust equilibrium between precision and recall, crucial for efficient risk assessment. Despite the model's relatively modest performance in forecasting defaults (recall of 45%), its exceptional ability to accurately classify non-defaulters aids financial institutions in effectively managing low-risk clients and improving resource allocation. An overall accuracy of 83%, accompanied by a weighted

In [ ]: