CREDIT RISK PREDICTION

Problem Statement:

The goal is to build a Credit Risk Prediction Model that determines whether a borrower will default (1) or not default (0) on a loan. This helps lenders mitigate financial risks.

In [1]:
```
import pandas as pd # for data manipulation
import numpy as np # for Mathematical Operations
```

In [2]:
```
path = 'C:/Users/Sanayak/Desktop/Credit Risk Dataset/credit_risk_dataset.csv'
df_loan = pd.read_csv(path)
df_loan.sample(10)
```

Out[2]:

	person_age	person_income	person_home_ownership	person_emp_length	loan_intent	loan_grade	loan_amnt
27289	27	130000	MORTGAGE	1.0	HOMEIMPROVEMENT	B	21000
8720	25	60000	MORTGAGE	6.0	HOMEIMPROVEMENT	B	4000
1844	21	30000	MORTGAGE	0.0	MEDICAL	A	7125
10305	22	66396	MORTGAGE	6.0	EDUCATION	C	15000
32564	53	45000	RENT	0.0	PERSONAL	C	5600
19527	32	49000	RENT	2.0	HOMEIMPROVEMENT	B	12000
14410	23	105600	MORTGAGE	2.0	PERSONAL	A	25000
6423	22	40000	RENT	1.0	MEDICAL	B	6000
28555	27	80000	MORTGAGE	7.0	MEDICAL	B	7400
18573	33	67800	RENT	3.0	HOMEIMPROVEMENT	B	18000

Variables Definition:

1. Person_age (Customer's age): Helps identify age groups that may be more prone to default.

2. Person_income (Annual income): The client's repayment capacity is strongly associated with their income.
3. Person_home_ownership (Home ownership): Indicates whether the client owns a home, rents, is mortgaged, etc. This influences financial stability.
4. Person_emp_length (Years of employment): Longer employment tenure often corresponds to lower default risk.
5. Loan_intent (Loan intent): The purpose of the loan (e.g., education, car, home) can influence the risk, as different loan types have varying default probabilities.
6. Loan_grade (Loan grade): An internal rating assigned to the loan indicating the initial perceived risk level.
7. Loan_amnt (Loan amount): Higher loan amounts mean greater exposure to risk.
8. Loan_int_rate (Interest rate): Higher interest rates are often associated with higher default risk.
9. loan_status (Loan status – target variable): 0 means non-default, 1 means default.
10. Loan_percent_income (Percent of income dedicated to the loan): A metric relating the loan amount to the borrower's income, indicating financial burden.
11. Cb_person_default_on_file (Historical defaults): Indicates if the client has previously defaulted.
12. Cb_person_cred_hist_length (Length of credit history): A longer credit history provides better insight into a client's behavior.

In [3]:
```
print('Number of Rows by Columns:',df_loan.shape)# Get the shape of the dataset
print('Number of duplicates:',df_loan.duplicated().sum())# Check for duplicated values
```

```
Number of Rows by Columns: (32581, 12)
Number of duplicates: 165
```

In [4]:
```
df_loan.drop_duplicates(inplace=True)#drop the duplicated entries and update the dataset
print(df_loan.duplicated().sum())
```

```
0
```

In [5]:
```
print('The number of missing values for each column are: ')
print(df_loan.isna().sum())#Check for null values
```

```
The number of missing values for each column are:
person_age                        0
person_income                     0
person_home_ownership             0
person_emp_length               887
loan_intent                       0
loan_grade                        0
loan_amnt                         0
loan_int_rate                  3095
loan_status                       0
loan_percent_income               0
cb_person_default_on_file         0
cb_person_cred_hist_length        0
dtype: int64
```

In [6]:
```
df_loan.describe()
```

Out[6]:

	person_age	person_income	person_emp_length	loan_amnt	loan_int_rate	loan_status	loan_percent_income	cb_pe
count	32416.000000	3.241600e+04	31529.00000	32416.000000	29321.000000	32416.000000	32416.000000	
mean	27.747008	6.609164e+04	4.79051	9593.845632	11.017265	0.218688	0.170250	
std	6.354100	6.201558e+04	4.14549	6322.730241	3.241680	0.413363	0.106812	
min	20.000000	4.000000e+03	0.00000	500.000000	5.420000	0.000000	0.000000	
25%	23.000000	3.854200e+04	2.00000	5000.000000	7.900000	0.000000	0.090000	
50%	26.000000	5.500000e+04	4.00000	8000.000000	10.990000	0.000000	0.150000	
75%	30.000000	7.921800e+04	7.00000	12250.000000	13.470000	0.000000	0.230000	
max	144.000000	6.000000e+06	123.00000	35000.000000	23.220000	1.000000	0.830000	

In [7]:
```
# Fill null with median value for the individual columns
df_loan[['person_emp_length','loan_int_rate']] = df_loan[['person_emp_length',
                                                  'loan_int_rate']].fillna(df_loan[['person_emp_length',
                                                                                   'loan_int_rate']].median(
```

In [8]:
```
df_loan[['person_emp_length','loan_int_rate']].isna().sum()
```

Out[8]:
```
person_emp_length    0
loan_int_rate        0
dtype: int64
```

In [9]:
```
df_loan.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 32416 entries, 0 to 32580
Data columns (total 12 columns):
 #   Column                      Non-Null Count  Dtype  
---  ------                      --------------  -----  
 0   person_age                  32416 non-null  int64  
 1   person_income               32416 non-null  int64  
 2   person_home_ownership       32416 non-null  object 
 3   person_emp_length           32416 non-null  float64
 4   loan_intent                 32416 non-null  object 
 5   loan_grade                  32416 non-null  object 
 6   loan_amnt                   32416 non-null  int64  
 7   loan_int_rate               32416 non-null  float64
 8   loan_status                 32416 non-null  int64  
 9   loan_percent_income         32416 non-null  float64
 10  cb_person_default_on_file   32416 non-null  object 
 11  cb_person_cred_hist_length  32416 non-null  int64  
dtypes: float64(3), int64(5), object(4)
memory usage: 3.2+ MB
```

Observations:

The Age column has an Highest value of 144 and the employment length has an Highest value of 123 which might be caused due to human error

In [10]:
```
# Define a function that removes the 95 percentile
def remove_outliers(df,columns):
    for col in columns:
        threshold = df[col].quantile(0.95)
        df = df[df[col] <= threshold]
    return df

cleaned_df = remove_outliers(df_loan, columns=['person_age','person_emp_length'])
print('Original DataFrame:')
print(df_loan[['person_age','person_emp_length']].max())
```

```
print('Cleaned DataFrame:')
print(cleaned_df[['person_age','person_emp_length']].max())
```

```
Original DataFrame:
person_age           144.0
person_emp_length    123.0
dtype: float64
Cleaned DataFrame:
person_age           40.0
person_emp_length    12.0
dtype: float64
```

Outliers:

Outliers were identified in:

`person_age` : Maximum = **144** (unrealistic).

`person_emp_length` : Maximum = **123**.

To fix this, values above the 95th percentile were removed.

Post-cleaning:

`person_age` capped at **40**.

`person_emp_length` capped at **12**.

In [11]:
```
cleaned_df.describe()
```

Out[11]:

	person_age	person_income	person_emp_length	loan_amnt	loan_int_rate	loan_status	loan_percent_income	cb_pe
count	29543.000000	2.954300e+04	29543.000000	29543.000000	29543.000000	29543.000000	29543.000000	
mean	26.557729	6.401581e+04	4.207325	9497.601801	11.028543	0.222015	0.171306	
std	4.410420	4.431807e+04	3.123315	6262.114024	3.076598	0.415608	0.106809	
min	20.000000	4.080000e+03	0.000000	500.000000	5.420000	0.000000	0.000000	
25%	23.000000	3.800000e+04	2.000000	5000.000000	8.490000	0.000000	0.090000	
50%	25.000000	5.500000e+04	4.000000	8000.000000	10.990000	0.000000	0.150000	
75%	29.000000	7.800000e+04	6.000000	12000.000000	13.160000	0.000000	0.230000	
max	40.000000	1.200000e+06	12.000000	35000.000000	23.220000	1.000000	0.830000	

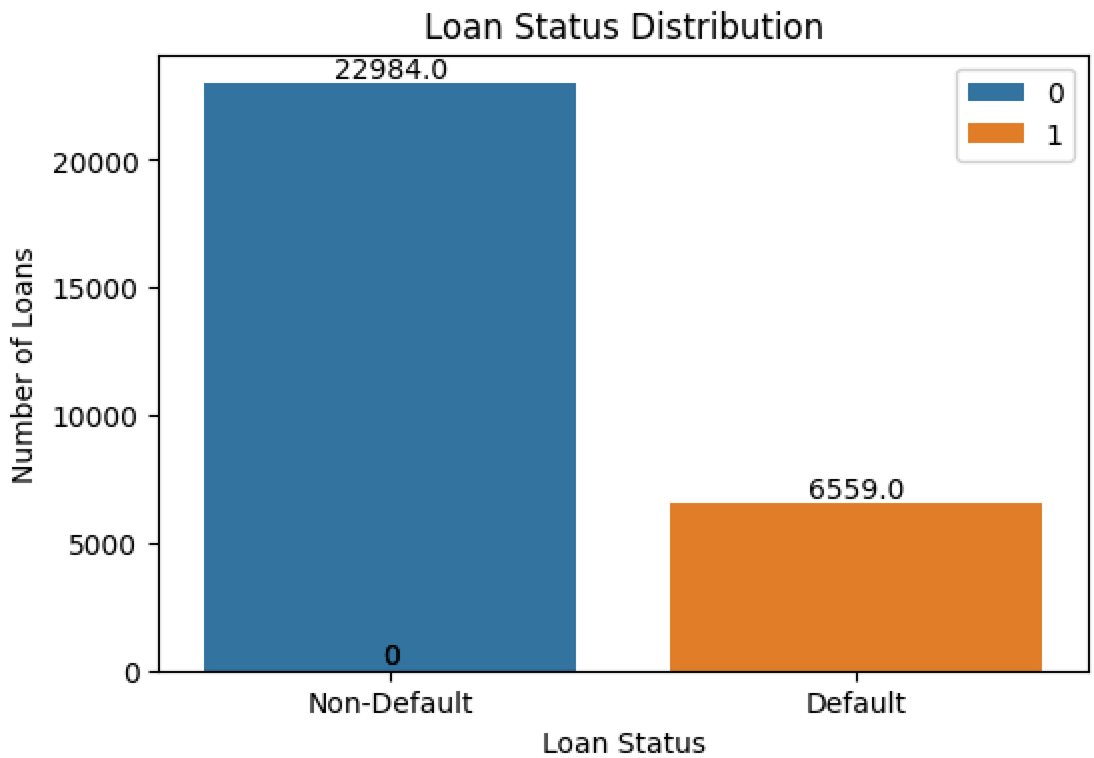
Exploratory Data Analysis

In [12]:

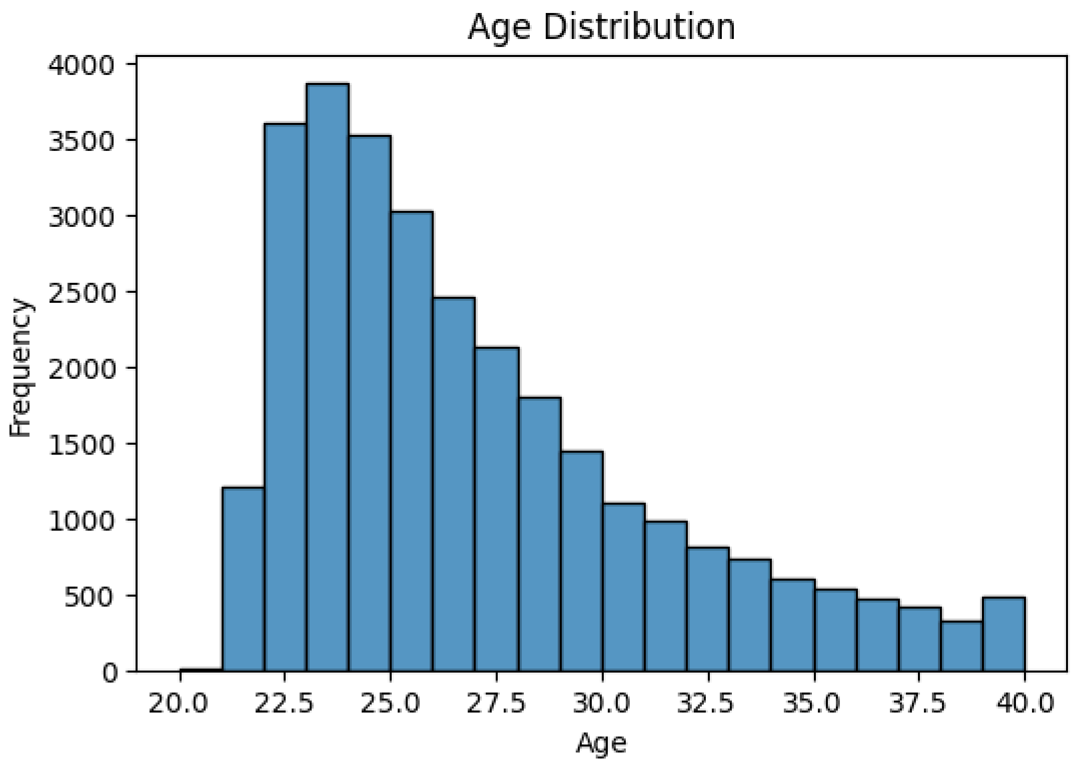
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [13]:

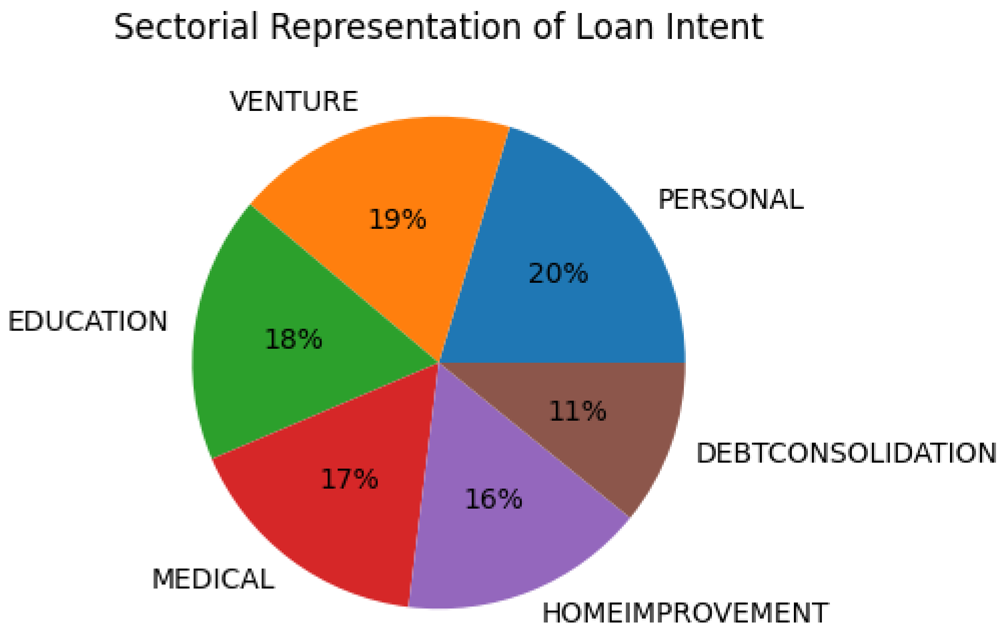
```
plt.figure(figsize=(6,4))
ax = sns.countplot(x='loan_status', data=cleaned_df,hue='loan_status')
for p in ax.patches:
    ax.annotate(f'{p.get_height()}',
                (p.get_x() + p.get_width() / 2., p.get_height()), ha='center',va='bottom')
plt.title('Loan Status Distribution')
plt.xlabel('Loan Status')
plt.ylabel('Number of Loans')
plt.xticks(ticks=[0,1], labels=['Non-Default','Default'])
plt.legend()
plt.show()
```



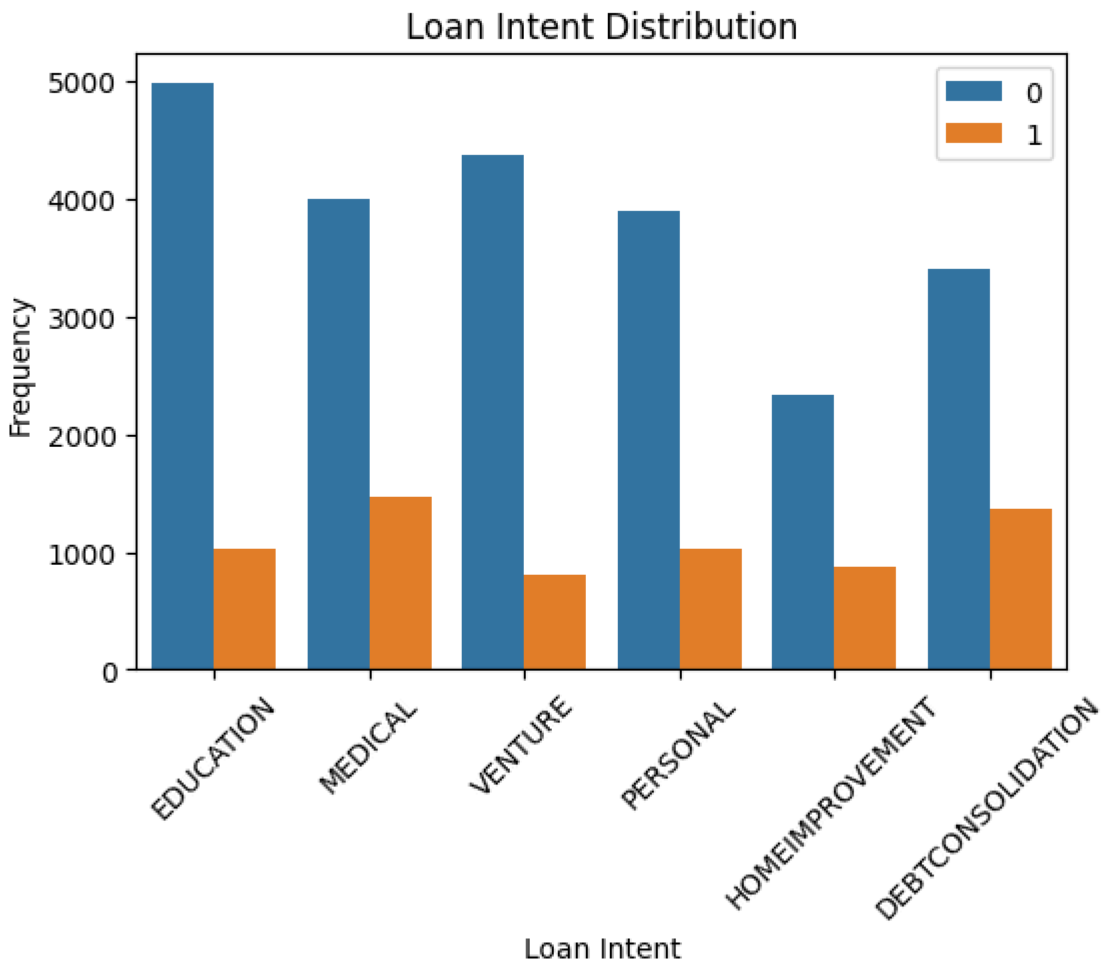
In [14]:
```
plt.figure(figsize=(6,4))
sns.histplot(cleaned_df['person_age'],kde=False,bins=20)
plt.title('Age Distribution')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()
```



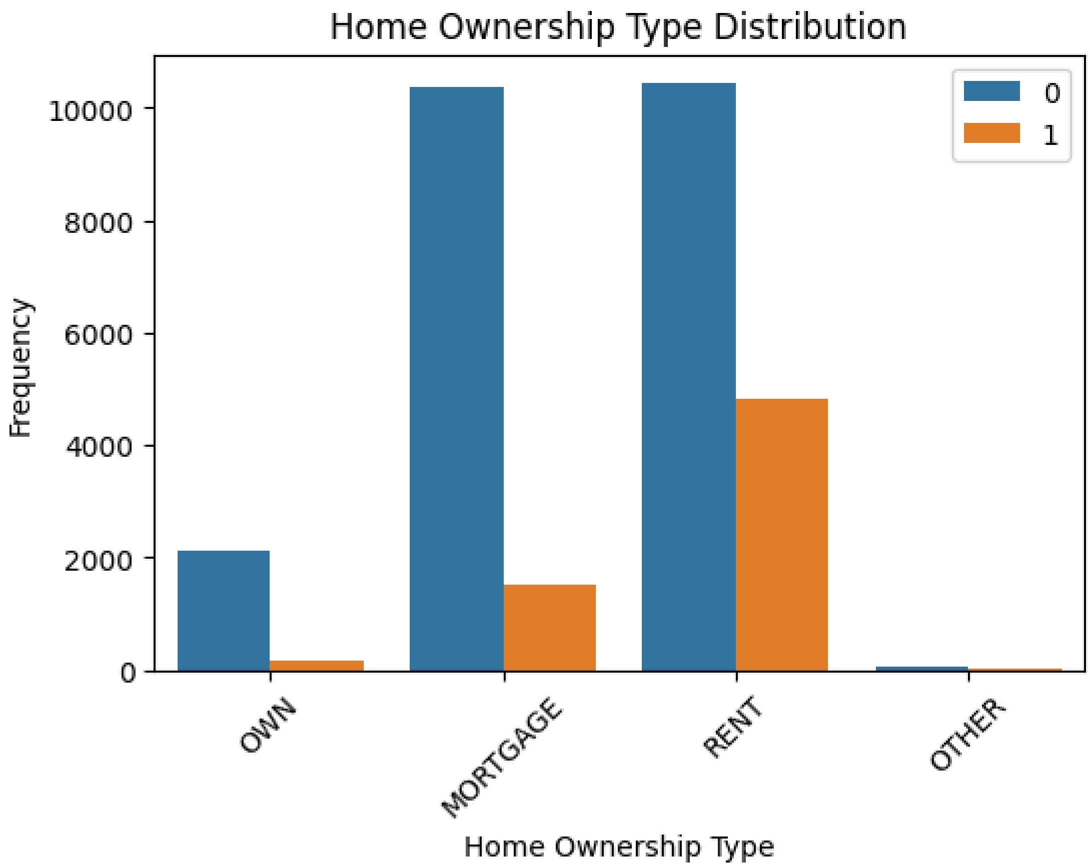
In [15]:
```
loan_intent_counts=cleaned_df['loan_intent'].value_counts()
labels = ['PERSONAL','VENTURE','EDUCATION','MEDICAL','HOMEIMPROVEMENT','DEBTCONSOLIDATION']
plt.figure(figsize=(6,4))
plt.pie(loan_intent_counts,labels=labels,autopct="%1.f%%")
plt.title('Sectorial Representation of Loan Intent')
plt.show()
```



In [16]:
```
plt.figure(figsize=(6,4))
sns.countplot(x='loan_intent', data=cleaned_df,hue='loan_status')
plt.title('Loan Intent Distribution')
plt.xlabel('Loan Intent')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



In [17]:
```
plt.figure(figsize=(6,4))
sns.countplot(x='person_home_ownership', data=cleaned_df,hue='loan_status')
plt.title('Home Ownership Type Distribution')
plt.xlabel('Home Ownership Type')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



In [18]:
```
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report,confusion_matrix,accuracy_score
```

In [19]:
```
# Split in to feature(X) and target variable(y)
X = cleaned_df.drop('loan_status',axis=1)
y = cleaned_df['loan_status']

# Split in to training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y,
```

```
                                                    test_size = 0.25,
                                                    random_state = 42)
```

In [20]:
```
# Identify the categorical and numerical features
categorical_features = X.select_dtypes(include=['object']).columns
numerical_features = X.select_dtypes(include=['int64','float64']).columns

# Create preprocessing pipeline for categorical and numerical features
categorical_transform = Pipeline(steps=[('ohe', OneHotEncoder(handle_unknown='ignore'))])
numerical_transform = Pipeline(steps=[('scaler', StandardScaler())])

#Combine the processing pipelines
preprocessor = ColumnTransformer(transformers=[('num', numerical_transform,
                                                 numerical_features),('cat',categorical_transform,
                                                                     categorical_features)])
```

Data Preprocessing

- **Categorical Features**:
  - Encoded using **OneHotEncoder**.
- **Numerical Features**:
  - Standardized using **StandardScaler**.
- Split data into **train (75%)** and **test (25%)** sets.

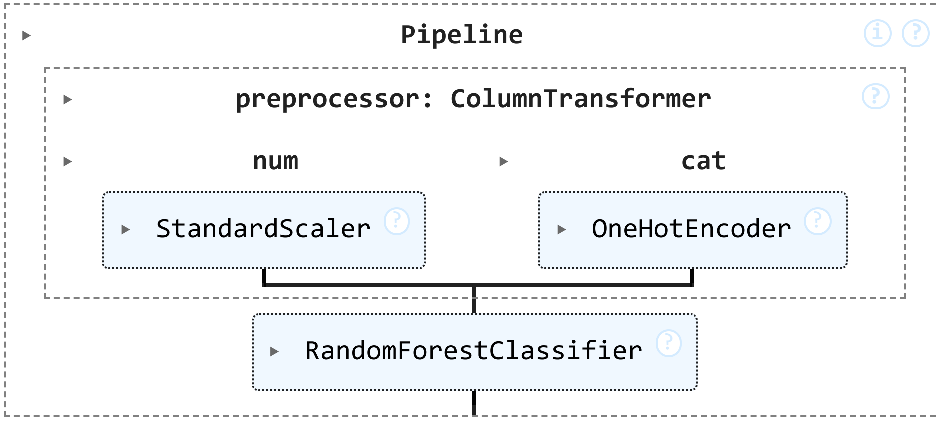
In [21]:
```
# Create a pipeline for the xgboost classifier
xgb_pipeline = Pipeline(steps=[('preprocessor', preprocessor),('classifier',XGBClassifier(random_state=42))])

# Create a pipeline for the randomforest classifier
rf_pipeline = Pipeline(steps=[('preprocessor', preprocessor),('classifier',RandomForestClassifier(random_state=42))])

#Train the xgboost model
xgb_pipeline.fit(X_train, y_train)

# Train the randomforest classifier
rf_pipeline.fit(X_train, y_train)
```

Out[21]:



Model Training

Two classifiers were built using pipelines:

1. **XGBoost**.
2. **Random Forest**.

Both models were trained and tested on the cleaned dataset.

In [22]:
```
# Make predictions on the testset
xgb_y_pred = xgb_pipeline.predict(X_test)
rf_y_pred = rf_pipeline.predict(X_test)


xgb_accuracy = accuracy_score(xgb_y_pred, y_test)
rf_accuracy = accuracy_score(rf_y_pred, y_test)


xgb_class_report = classification_report(xgb_y_pred, y_test)
rf_class_report = classification_report(rf_y_pred,y_test)
```

In [23]:
```
print(f'XGBoost Model Accuary: {xgb_accuracy:.2f}')
print(f'RandomForest Model Accuary: {rf_accuracy:.2f}')

print('XGBoost Classification Report:\n', xgb_class_report)
print('RandomForest Classification Report:\n', rf_class_report)
```

```
XGBoost Model Accuary: 0.93
RandomForest Model Accuary: 0.93
XGBoost Classification Report:
               precision    recall  f1-score   support

           0       0.99      0.93      0.96      6059
           1       0.75      0.95      0.84      1327

    accuracy                           0.93      7386
   macro avg       0.87      0.94      0.90      7386
weighted avg       0.95      0.93      0.94      7386

RandomForest Classification Report:
               precision    recall  f1-score   support

           0       0.99      0.92      0.96      6131
           1       0.72      0.97      0.83      1255

    accuracy                           0.93      7386
   macro avg       0.86      0.95      0.89      7386
weighted avg       0.95      0.93      0.94      7386
```

- Both models achieved **93% accuracy**.
- XGBoost performed slightly better on recall for defaults.

In [24]:

```
from sklearn.model_selection import GridSearchCV

# Define parameter grid for XGBoost and Random Forest
param_grid_xgb = {'classifier__max_depth': [3,5,7],
                 'classifier__learning_rate':[0.1,0.01,0.001],
                 'classifier__n_estimators':[100,200,300]}

param_grid_rf = {'classifier__n_estimators':[100,200,300],
                'classifier__max_depth':[5,10,15],
                'classifier__min_samples_split':[2,5,10],
                'classifier__min_samples_leaf':[1,2,4]}

# Create GridSearch objects
grid_search_xgb = GridSearchCV(xgb_pipeline, param_grid_xgb, cv=5, scoring='accuracy')
```

```
grid_search_rf = GridSearchCV(rf_pipeline, param_grid_rf, cv=5, scoring='accuracy')

# Fit the GridSearch object to training data
grid_search_xgb.fit(X_train, y_train)
grid_search_rf.fit(X_train, y_train)

# Print the best parameters and score for each model
print('Best parameters for XGBoost:', grid_search_xgb.best_params_)
print('Best Score for XGBoost:', grid_search_xgb.best_score_)
print('Best parameters for Random Forest:', grid_search_rf.best_params_)
print('Best Score for Random Forest:', grid_search_rf.best_score_)
```

```
Best parameters for XGBoost: {'classifier__learning_rate': 0.1, 'classifier__max_depth': 7, 'classifier__n_estimator
s': 200}
Best Score for XGBoost: 0.9346935196478373
Best parameters for Random Forest: {'classifier__max_depth': 15, 'classifier__min_samples_leaf': 1, 'classifier__min_
samples_split': 5, 'classifier__n_estimators': 300}
Best Score for Random Forest: 0.9314440046211994
```

In [25]:
```
# use the best model to make predictions and evaluate
best_xgb_model = grid_search_xgb.best_estimator_
best_rf_model = grid_search_rf.best_estimator_


xgb_predictions = best_xgb_model.predict(X_test)
rf_predictions = best_rf_model.predict(X_test)

xgb_class_report_grid = classification_report(xgb_predictions,y_test)
rf_class_report_grid = classification_report(rf_predictions,y_test)

# Print the classification report for both models
print('XGBoost Classification Report:\n', xgb_class_report_grid)
print('RandomForest Classification Report:\n', rf_class_report_grid)
```

```
XGBoost Classification Report:
               precision    recall  f1-score   support

           0       0.99      0.93      0.96      6099
           1       0.74      0.97      0.84      1287

    accuracy                           0.94      7386
   macro avg       0.87      0.95      0.90      7386
weighted avg       0.95      0.94      0.94      7386

RandomForest Classification Report:
               precision    recall  f1-score   support

           0       0.99      0.92      0.96      6144
           1       0.72      0.98      0.83      1242

    accuracy                           0.93      7386
   macro avg       0.86      0.95      0.89      7386
weighted avg       0.95      0.93      0.94      7386
```

Hyperparameter Tuning

- Performed GridSearchCV to optimize model parameters:
    - **XGBoost**:
        - Best Parameters: `max_depth=7` , `learning_rate=0.1` , `n_estimators=200` .
        - Best Accuracy: **93.47%**.
    - **Random Forest**:
        - Best Parameters: `max_depth=15` , `min_samples_split=5` , `n_estimators=300` .
        - Best Accuracy: **93.14%**.

Final Results

- After tuning, XGBoost outperformed Random Forest slightly in recall and accuracy.

- **XGBoost** is the preferred model for Credit Risk Prediction.

Additional Considerations:

-**Model Selection:** Experiment with different models.

-**Regularization:** Apply techniques like L1 and L2 regularization to prevent overfitting.

Conclusion:

The project successfully built and optimized a **Credit Risk Prediction Model**.

Key takeaways:

- Age and loan intent play significant roles in default likelihood.
- Data cleaning and feature engineering improved model performance.
- XGBoost, with hyperparameter tuning, delivered the best results.

In [26]:
```
import joblib
```

In [27]:
```
joblib.dump(best_xgb_model, 'best_xgb_model.pkl')
```

Out[27]:
```
['best_xgb_model.pkl']
```

Prepared By: **Victor Itinah Iniobong**

Tools Used: Python, Pandas, Scikit-Learn, XGBoost, Matplotlib, Seaborn.