Project Title: Classifying The Credit Scores

Problem Statement:

As a data scientist in a global finance company, your objective is to develop a machine learning model that predicts individuals' credit scores based on their financial and credit-related information. The company aims to automate and enhance the credit scoring process using intelligent systems.

1. Importing Necessary Libraries and Dataset Download

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
# data ='https://raw.githubusercontent.com/rashakil-ds/Public-Datasets/main/Bank%20Data.csv' #github link
data = 'Bank Data.csv' #if dataset not found on the link
```

In [3]:
```
df = pd.read_csv(data)
```

2. Data Exploration and Preprocessing

- Conduct exploratory data analysis (EDA) to understand the distribution of features and the target variable.
- Handle any missing values, outliers, or data inconsistencies.
- Encode categorical variables if necessary.
- Explore the distribution of the target variable.

In [4]:
```
pd.set_option('display.max_columns', None)
df.head()
```

Out[4]:

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card	Interes
0	0x160a	CUS_0xd40	September	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	3	4	
1	0x160b	CUS_0xd40	October	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	3	4	
2	0x160c	CUS_0xd40	November	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	3	4	
3	0x160d	CUS_0xd40	December	Aaron Maashoh	24_	821-00-0265	Scientist	19114.12	NaN	3	4	
4	0x1616	CUS_0x21b1	September	Rick Rothackerj	28	004-07-5839	_______	34847.84	3037.986667	2	4	

In [5]:
```
df.rename(columns={'Credit_Mix':'Credit_Score'} , inplace=True)
```

In [6]:
```
df.columns
```

Out[6]:
```
Index(['ID', 'Customer_ID', 'Month', 'Name', 'Age', 'SSN', 'Occupation',
       'Annual_Income', 'Monthly_Inhand_Salary', 'Num_Bank_Accounts',
       'Num_Credit_Card', 'Interest_Rate', 'Num_of_Loan', 'Type_of_Loan',
       'Delay_from_due_date', 'Num_of_Delayed_Payment', 'Changed_Credit_Limit',
       'Num_Credit_Inquiries', 'Credit_Score', 'Outstanding_Debt',
       'Credit_Utilization_Ratio', 'Credit_History_Age',
       'Payment_of_Min_Amount', 'Total_EMI_per_month',
       'Amount_invested_monthly', 'Payment_Behaviour', 'Monthly_Balance'],
      dtype='object')
```

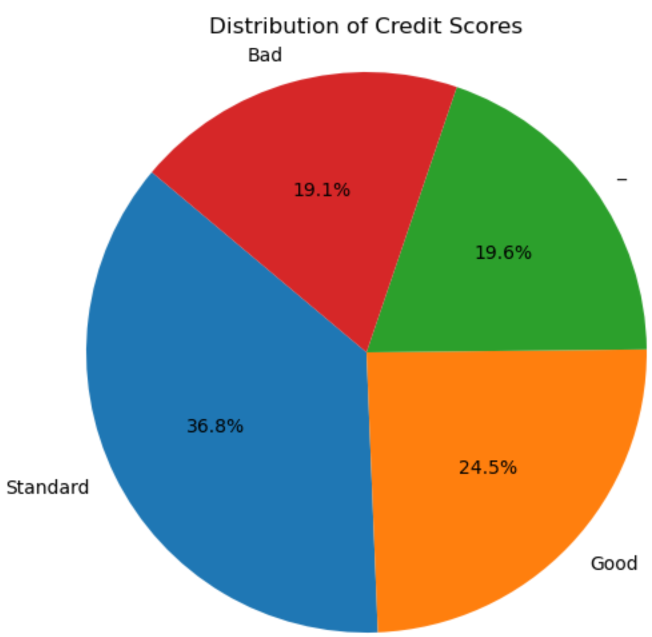
In [7]:
```
df.head()
```

Out[7]:

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card	Interes
0	0x160a	CUS_0xd40	September	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	3	4	
1	0x160b	CUS_0xd40	October	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	3	4	
2	0x160c	CUS_0xd40	November	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	3	4	
3	0x160d	CUS_0xd40	December	Aaron Maashoh	24_	821-00-0265	Scientist	19114.12	NaN	3	4	
4	0x1616	CUS_0x21b1	September	Rick Rothackerj	28	004-07-5839	_______	34847.84	3037.986667	2	4	

In [8]:
```
# Count the frequency of each unique value in the 'Credit_Score' column
score_counts = df['Credit_Score'].value_counts()

# Create a pie chart
plt.figure(figsize=(8, 6))
plt.pie(score_counts, labels=score_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Credit Scores')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle
plt.show()
```



In [9]:
```
df['Credit_Score'].value_counts()
```

Out[9]:
```
Credit_Score
Standard    18379
Good        12260
_            9805
Bad          9556
Name: count, dtype: int64
```

In [10]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 27 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   ID                        50000 non-null  object 
 1   Customer_ID               50000 non-null  object 
 2   Month                     50000 non-null  object 
 3   Name                      44985 non-null  object 
 4   Age                       50000 non-null  object 
 5   SSN                       50000 non-null  object 
 6   Occupation                50000 non-null  object 
 7   Annual_Income             50000 non-null  object 
 8   Monthly_Inhand_Salary     42502 non-null  float64
 9   Num_Bank_Accounts         50000 non-null  int64  
 10  Num_Credit_Card           50000 non-null  int64  
 11  Interest_Rate             50000 non-null  int64  
 12  Num_of_Loan               50000 non-null  object 
 13  Type_of_Loan              44296 non-null  object 
 14  Delay_from_due_date       50000 non-null  int64  
 15  Num_of_Delayed_Payment    46502 non-null  object 
 16  Changed_Credit_Limit      50000 non-null  object 
 17  Num_Credit_Inquiries      48965 non-null  float64
 18  Credit_Score              50000 non-null  object 
 19  Outstanding_Debt          50000 non-null  object 
 20  Credit_Utilization_Ratio  50000 non-null  float64
 21  Credit_History_Age        45530 non-null  object 
 22  Payment_of_Min_Amount     50000 non-null  object 
 23  Total_EMI_per_month       50000 non-null  float64
 24  Amount_invested_monthly   47729 non-null  object 
 25  Payment_Behaviour         50000 non-null  object 
 26  Monthly_Balance           49438 non-null  object 
dtypes: float64(4), int64(4), object(19)
memory usage: 10.3+ MB
```

In [11]:
```
df['Annual_Income'] = pd.to_numeric(df['Annual_Income'], errors='coerce')
df['Num_of_Loan'] = pd.to_numeric(df['Num_of_Loan'], errors='coerce')
df['Num_of_Delayed_Payment'] = pd.to_numeric(df['Num_of_Delayed_Payment'], errors='coerce')
df['Changed_Credit_Limit'] = pd.to_numeric(df['Changed_Credit_Limit'], errors='coerce')
df['Outstanding_Debt'] = pd.to_numeric(df['Outstanding_Debt'], errors='coerce')
df['Amount_invested_monthly'] = pd.to_numeric(df['Amount_invested_monthly'], errors='coerce')
df['Monthly_Balance'] = pd.to_numeric(df['Monthly_Balance'], errors='coerce')
```

In [12]:
```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder

# Initialize LabelEncoder
label_encoder = LabelEncoder()
ordinal_encoder = OrdinalEncoder()

df['Credit_Score_Copy'] = df['Credit_Score']
df['Credit_Score'] = label_encoder.fit_transform(df['Credit_Score'])
df['Payment_of_Min_Amount'] = label_encoder.fit_transform(df['Payment_of_Min_Amount'])

encoded_to_category = {label: category for label, category in zip(df['Credit_Score'], df['Credit_Score_Copy'])}
df.drop('Credit_Score_Copy', axis=1, inplace=True)
encoded_to_category
```

Out[12]:
```
{1: 'Good', 3: '_', 2: 'Standard', 0: 'Bad'}
```

In [13]:
```
df.head()
```

Out[13]:

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card	Interes
0	0x160a	CUS_0xd40	September	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	3	4	
1	0x160b	CUS_0xd40	October	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	3	4	
2	0x160c	CUS_0xd40	November	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	3	4	
3	0x160d	CUS_0xd40	December	Aaron Maashoh	24_	821-00-0265	Scientist	19114.12	NaN	3	4	
4	0x1616	CUS_0x21b1	September	Rick Rothackerj	28	004-07-5839	_______	34847.84	3037.986667	2	4	

In [14]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 27 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   ID                        50000 non-null  object 
 1   Customer_ID               50000 non-null  object 
 2   Month                     50000 non-null  object 
 3   Name                      44985 non-null  object 
 4   Age                       50000 non-null  object 
 5   SSN                       50000 non-null  object 
 6   Occupation                50000 non-null  object 
 7   Annual_Income             46480 non-null  float64
 8   Monthly_Inhand_Salary     42502 non-null  float64
 9   Num_Bank_Accounts         50000 non-null  int64  
 10  Num_Credit_Card           50000 non-null  int64  
 11  Interest_Rate             50000 non-null  int64  
 12  Num_of_Loan               47564 non-null  float64
 13  Type_of_Loan              44296 non-null  object 
 14  Delay_from_due_date       50000 non-null  int64  
 15  Num_of_Delayed_Payment    45075 non-null  float64
 16  Changed_Credit_Limit      48941 non-null  float64
 17  Num_Credit_Inquiries      48965 non-null  float64
 18  Credit_Score              50000 non-null  int32  
 19  Outstanding_Debt          49509 non-null  float64
 20  Credit_Utilization_Ratio  50000 non-null  float64
 21  Credit_History_Age        45530 non-null  object 
 22  Payment_of_Min_Amount     50000 non-null  int32  
 23  Total_EMI_per_month       50000 non-null  float64
 24  Amount_invested_monthly   45554 non-null  float64
 25  Payment_Behaviour         50000 non-null  object 
 26  Monthly_Balance           49432 non-null  float64
dtypes: float64(11), int32(2), int64(4), object(10)
memory usage: 9.9+ MB
```

In [15]:
```
df.describe()
```

Out[15]:

	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card	Interest_Rate	Num_of_Loan	Delay_from_due_date	Num_of_Delayed_Pay
count	4.648000e+04	42502.000000	50000.000000	50000.000000	50000.000000	47564.000000	50000.000000	45075.0(
mean	1.651169e+05	4182.004291	16.838260	22.921480	68.772640	3.267787	21.052640	30.74
std	1.341967e+06	3174.109304	116.396848	129.314804	451.602363	67.139627	14.860397	220.1{
min	7.005930e+03	303.645417	-1.000000	0.000000	1.000000	-100.000000	-5.000000	-3.0(
25%	1.943560e+04	1625.188333	3.000000	4.000000	8.000000	1.000000	10.000000	9.0(
50%	3.757587e+04	3086.305000	6.000000	5.000000	13.000000	3.000000	18.000000	14.0(
75%	7.276004e+04	5934.189094	7.000000	7.000000	20.000000	5.000000	28.000000	18.0(
max	2.413726e+07	15204.633333	1798.000000	1499.000000	5799.000000	1496.000000	67.000000	4399.0(

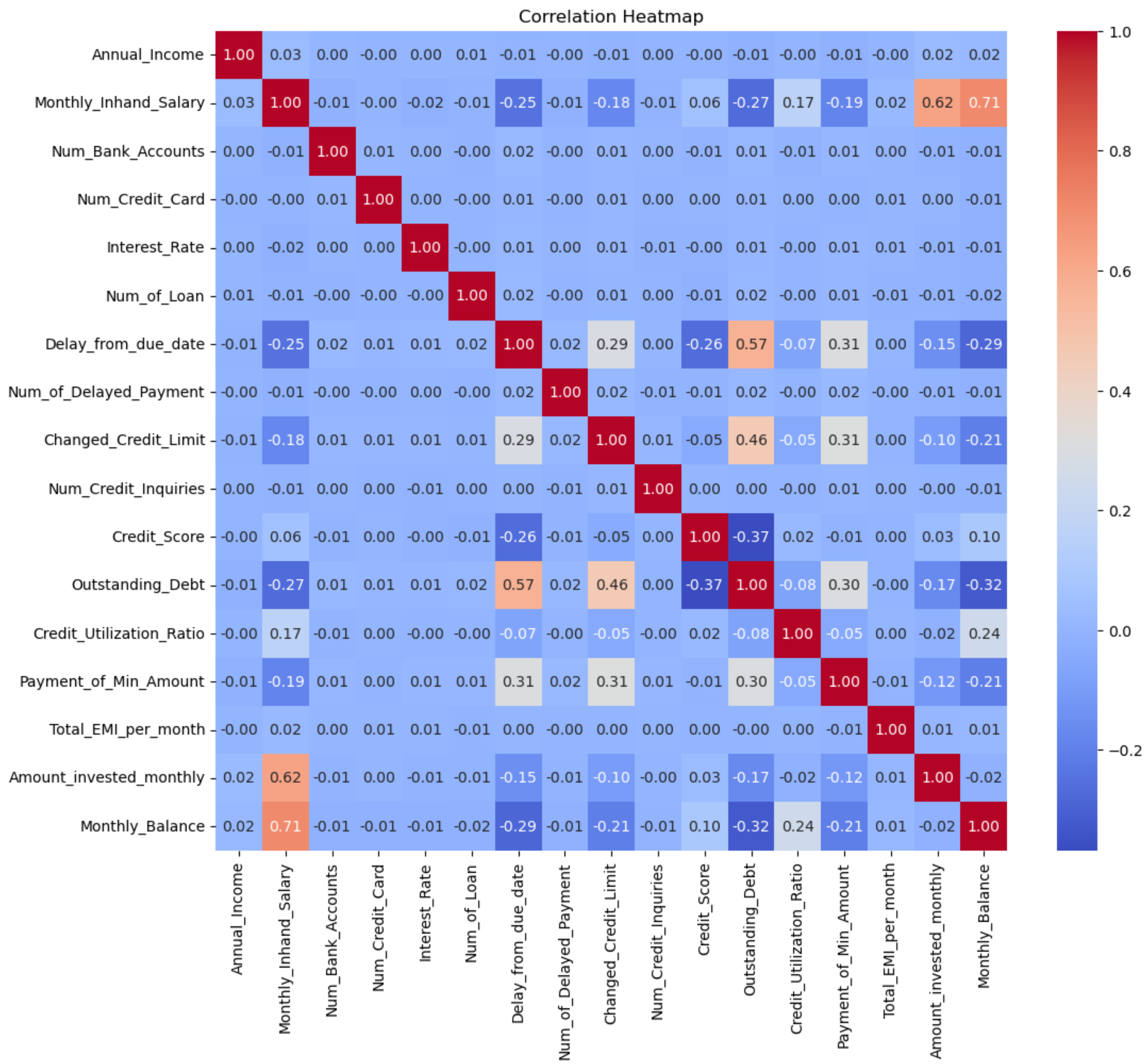
In [16]:
```
# Filter out non-numeric columns
numeric_data = df.select_dtypes(include=['int32','int64', 'float64'])

# Calculate the correlation matrix
correlation_matrix = numeric_data.corr()
correlation_matrix['Credit_Score']
```

Out[16]:
```
Annual_Income              -0.001759
Monthly_Inhand_Salary       0.058856
Num_Bank_Accounts          -0.007512
Num_Credit_Card             0.001223
Interest_Rate              -0.004881
Num_of_Loan                -0.010771
Delay_from_due_date        -0.261131
Num_of_Delayed_Payment     -0.005393
Changed_Credit_Limit       -0.046034
Num_Credit_Inquiries        0.001656
Credit_Score                1.000000
Outstanding_Debt           -0.367771
Credit_Utilization_Ratio    0.021859
Payment_of_Min_Amount      -0.007459
Total_EMI_per_month         0.000156
Amount_invested_monthly     0.030876
Monthly_Balance             0.096127
Name: Credit_Score, dtype: float64
```

In [17]:
```
import seaborn as sns
import matplotlib.pyplot as plt


# Plot the heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap')
plt.show()
```



In [18]:
```
# Drop unnecessary columns
unnecessary_columns = ['ID','Age','Customer_ID','Name','SSN','Month','Occupation','Monthly_Inhand_Salary','Type_of_Loan','Num_of_
df.drop(columns=unnecessary_columns, inplace=True)
```

In [19]:
```
df.head()
```

Out[19]:

	Annual_Income	Num_Bank_Accounts	Num_Credit_Card	Interest_Rate	Num_of_Loan	Delay_from_due_date	Changed_Credit_Limit	Num_Credit_Inquiries	Cre
0	19114.12	3	4	3	4.0	3	11.27	2022.0	
1	19114.12	3	4	3	4.0	3	13.27	4.0	
2	19114.12	3	4	3	4.0	-1	12.27	4.0	
3	19114.12	3	4	3	4.0	4	11.27	4.0	
4	34847.84	2	4	6	1.0	3	5.42	5.0	

In [20]:
```
df.isnull().sum()
```

Out[20]:
```
Annual_Income               3520
Num_Bank_Accounts              0
Num_Credit_Card                0
Interest_Rate                  0
Num_of_Loan                 2436
Delay_from_due_date            0
Changed_Credit_Limit        1059
Num_Credit_Inquiries        1035
Credit_Score                   0
Outstanding_Debt             491
Credit_Utilization_Ratio       0
Payment_of_Min_Amount          0
Total_EMI_per_month            0
Monthly_Balance              568
dtype: int64
```

In [21]:
```
# Define custom imputation function
def custom_imputer(column):
    if column.dtype == 'object':
        imputer = SimpleImputer(strategy='most_frequent')
    else:
        imputer = SimpleImputer(strategy='mean')
    return imputer.fit_transform(column.values.reshape(-1, 1)).flatten()

# Impute missing values in the original DataFrame
for column in df.columns:
    if df[column].isnull().any():
        df[column] = custom_imputer(df[column])
```

In [22]:
```
df.isnull().sum()
```

Out[22]:
```
Annual_Income               0
Num_Bank_Accounts           0
Num_Credit_Card             0
Interest_Rate               0
Num_of_Loan                 0
Delay_from_due_date         0
Changed_Credit_Limit        0
Num_Credit_Inquiries        0
Credit_Score                0
Outstanding_Debt            0
Credit_Utilization_Ratio    0
Payment_of_Min_Amount       0
Total_EMI_per_month         0
Monthly_Balance             0
dtype: int64
```

In [23]:
```
df.duplicated().sum()
```

Out[23]:
```
0
```

In [24]:
```
X = df.drop(columns=['Credit_Score'])
y = df[['Credit_Score']]
```

In [25]:
```
# Standardize numerical features
scaler = StandardScaler()
numerical_features = X.columns
X[numerical_features] = scaler.fit_transform(X[numerical_features])
```

In [26]:
```
X.head()
```

Out[26]:

	Annual_Income	Num_Bank_Accounts	Num_Credit_Card	Interest_Rate	Num_of_Loan	Delay_from_due_date	Changed_Credit_Limit	Num_Credit_Inquiries	Out
0	-0.112843	-0.118890	-0.146323	-0.145644	0.011182	-1.214828	0.133439	10.218501	
1	-0.112843	-0.118890	-0.146323	-0.145644	0.011182	-1.214828	0.431574	-0.133791	
2	-0.112843	-0.118890	-0.146323	-0.145644	0.011182	-1.484002	0.282506	-0.133791	
3	-0.112843	-0.118890	-0.146323	-0.145644	0.011182	-1.147534	0.133439	-0.133791	
4	-0.100683	-0.127481	-0.146323	-0.139001	-0.034632	-1.214828	-0.738606	-0.128661	

3. Model Selection:

Choose suitable machine learning classification models for predicting credit scores. Suggested models include:

- Logistic Regression
- Random Forest Classifier
- Support Vector Machine (SVM)
- Gradient Boosting Classifier (e.g., XGBoost)

In [27]:
```
# Initialize regression models
models = {
    'Logistic Regression': LogisticRegression(),
    'Random Forest Classifier': RandomForestClassifier(),
    'Support Vector Machine' : SVC(),
    'Gradient Boosting Classifier (XGBoost)': XGBClassifier()
}
```

4. Model Training:

- Train each selected model using the training dataset.
- Utilize evaluation metrics suitable for classification tasks, such as accuracy, precision, recall, F1 score, and confusion matrix.

In [28]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [29]:
```
# Train each selected model on the training dataset
for name, model in models.items():
    model.fit(X_train, y_train)
```

In [30]:
```
for name, model in models.items():
    y_pred = model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='macro')
    recall = recall_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')
    conf_matrix = confusion_matrix(y_test, y_pred)

    print(f'{name}:')
    print(f'Accuracy: {accuracy}')
    print(f'Precision: {precision}')
    print(f'Recall: {recall}')
    print(f'F1 Score: {f1}')
    print(f'Confusion Matrix:\n{conf_matrix}')
    print('---')
    print('\n')
```

```
Logistic Regression:
Accuracy: 0.6299
Precision: 0.544467208693761
Recall: 0.5941208375942246
F1 Score: 0.5496522761575433
Confusion Matrix:
[[1528   14  287   91]
 [   0 1943  494   21]
 [ 171  570 2758  156]
 [ 404  591  902   70]]
---


Random Forest Classifier:
Accuracy: 0.7547
Precision: 0.6326552682190215
Recall: 0.7091864094186126
F1 Score: 0.6501012024969982
Confusion Matrix:
[[1793    0   54   73]
 [   0 2343   49   66]
 [  56  129 3341  129]
 [ 436  599  862   70]]
---


Support Vector Machine:
Accuracy: 0.6779
Precision: 0.5142518581835572
Recall: 0.6439350469151699
F1 Score: 0.5692197548806275
Confusion Matrix:
[[1679    3  235    3]
 [   0 2296  161    1]
 [ 161  687 2804    3]
 [ 432  732  803    0]]
---


Gradient Boosting Classifier (XGBoost):
Accuracy: 0.7507
Precision: 0.619097868560671
Recall: 0.702528305144135
F1 Score: 0.6324188811789677
Confusion Matrix:
[[1775    0  120   25]
 [   0 2345   90   23]
 [  84  170 3366   35]
 [ 442  615  889   21]]
---
```

5. Hyperparameter Tuning:

- Conduct hyperparameter tuning for at least one model using methods like Grid Search or Random Search.
- Explain the chosen hyperparameters and the reasoning behind them.

In [31]:
```
# Define hyperparameter grid
param_grid = {
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'n_estimators': [100, 200, 300]
}

# Initialize XGBClassifier
xgb = XGBClassifier()

# Perform Random Search
random_search = RandomizedSearchCV(xgb, param_distributions=param_grid, n_iter=10, cv=5, scoring='accuracy')
random_search.fit(X_train, y_train)

# Best hyperparameters
print("Best Hyperparameters:", random_search.best_params_)

# Evaluate on test set
y_pred = random_search.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Test Accuracy:", accuracy)
```

```
Best Hyperparameters: {'n_estimators': 300, 'max_depth': 7, 'learning_rate': 0.1}
Test Accuracy: 0.7552
```

Chosen Hyperparameters and Reasoning:

1. **Learning Rate (`learning_rate`):**
   - The learning rate controls the step size during the gradient descent optimization process.
   - Lower values of the learning rate require more boosting rounds (trees) to fit the data but can lead to better generalization and prevent overfitting.
   - Higher values of the learning rate allow faster convergence during training but might lead to overfitting if not properly tuned.
2. **Maximum Depth of Trees (`max_depth`):**
   - This parameter controls the maximum depth of each tree in the ensemble.
   - Increasing the maximum depth allows the trees to capture more complex relationships in the data, potentially leading to overfitting.
   - On the other hand, limiting the maximum depth helps prevent overfitting by simplifying the trees.
3. **Number of Estimators (`n_estimators`):**
   - This parameter determines the number of boosting rounds or trees to be built.
   - Increasing the number of estimators can improve the model's performance, but it also increases the computational cost.
   - It's essential to find an optimal balance where increasing the number of estimators no longer improves the model's performance significantly.

Reasoning:

- **Learning Rate and Maximum Depth:** These two hyperparameters are critical in controlling the complexity of the individual trees and the overall ensemble. By tuning these parameters, we aim to find the right balance between model complexity and generalization ability. A lower learning rate combined with a limited maximum depth helps prevent overfitting by constraining the model's capacity to fit noise in the data.
- **Number of Estimators:** Tuning the number of estimators is essential to find the optimal trade-off between model performance and computational efficiency. By searching for the right number of boosting rounds, we can ensure that the model achieves sufficient performance without unnecessary computational overhead.

Overall, tuning these hyperparameters allows us to optimize the XGBoost model's performance, improve its generalization ability, and prevent overfitting on the dataset.

6. Model Evaluation:

- Assess the performance of each model on the testing set.
- Discuss the strengths and limitations of each model in the context of credit score classification.

Based on the evaluation metrics for each model on the testing set, let's discuss the strengths and limitations of each model in the context of credit score classification:

1. Logistic Regression:

- **Strengths:**
  - Logistic Regression is a simple and interpretable model.
  - It provides probabilities for each class, making it easy to interpret the model's confidence in its predictions.
- **Limitations:**
  - Logistic Regression assumes a linear relationship between the features and the log-odds of the target variable, which may not capture complex relationships in the data.
  - It is sensitive to outliers and multicollinearity among features, which can affect its performance.
  - The model's performance (accuracy: 0.6299) suggests that it may struggle with capturing the complex patterns in the data, resulting in lower accuracy compared to other models.

2. Random Forest Classifier:

- **Strengths:**
  - Random Forest Classifier is an ensemble learning method that combines multiple decision trees, providing robustness and improved generalization.
  - It handles non-linear relationships well and is less sensitive to overfitting compared to individual decision trees.
  - The model's performance (accuracy: 0.7547) indicates improved accuracy compared to Logistic Regression.
- **Limitations:**
  - Random Forest models can be computationally expensive, especially with a large number of trees and features.
  - It may not perform well with imbalanced datasets or datasets with noisy features.

3. Support Vector Machine (SVM):

- **Strengths:**
  - SVM is effective in high-dimensional spaces and can capture complex relationships in the data using kernel functions.
  - It works well with both linear and non-linear decision boundaries.
- **Limitations:**
  - SVM's performance (accuracy: 0.6779) suggests it may not generalize as well as other models on this dataset.
  - It can be sensitive to the choice of the kernel function and hyperparameters, which require careful tuning.

4. Gradient Boosting Classifier (XGBoost):

- **Strengths:**
  - Gradient Boosting Classifier, particularly XGBoost, is a powerful ensemble learning method known for its high performance and scalability.
  - It sequentially builds a set of weak learners (decision trees) and combines them to make accurate predictions.
  - XGBoost handles missing values well and is less prone to overfitting compared to other tree-based models.
- **Limitations:**
  - While XGBoost generally performs well, its performance (accuracy: 0.7507) on this dataset is slightly lower compared to Random Forest.
  - Tuning XGBoost's hyperparameters can be time-consuming, especially with a large number of parameters to optimize.

Overall:

- Random Forest Classifier demonstrates the highest accuracy (0.7547) among the evaluated models, indicating its effectiveness in this classification task.
- Logistic Regression has the lowest accuracy (0.6299) and may struggle to capture the complex patterns in the data.
- SVM and XGBoost perform moderately well, but their performance could potentially be improved with further hyperparameter tuning.

7. Interpretability:

- If applicable, explore methods to interpret the model's decisions and understand the factors influencing credit score classifications.

Interpreting the decisions of machine learning models, especially complex ones like Random Forest, Gradient Boosting, and SVM, can be challenging due to their inherent complexity. However, there are several techniques and approaches that can help in interpreting model decisions and understanding the factors influencing credit score classifications:

1. **Feature Importance:**
   - For ensemble models like Random Forest and Gradient Boosting, feature importance can be calculated to understand which features have the most significant impact on credit score classifications.
   - Feature importance scores indicate the relative importance of each feature in making predictions. Higher scores suggest stronger influences on the model's decisions.
2. **Partial Dependence Plots (PDPs):**
   - PDPs show the relationship between a feature and the predicted outcome while marginalizing the effects of all other features.
   - By analyzing PDPs, we can understand how changes in a particular feature affect the predicted credit score, providing insights into the direction and magnitude of the impact.
3. **SHAP Values (SHapley Additive exPlanations):**
   - SHAP values provide a unified measure of feature importance and directionality in model predictions.
   - They explain individual predictions by quantifying the contribution of each feature to the difference between the model's prediction and the average prediction.
   - Analyzing SHAP values helps in understanding the specific factors influencing each individual's credit score classification.
4. **Local Interpretable Model-agnostic Explanations (LIME):**
   - LIME is a model-agnostic technique that explains individual predictions of black-box models by approximating their decision boundaries using interpretable models (e.g., linear models).
   - By generating local explanations for individual predictions, LIME helps in understanding why a particular individual received a specific credit score classification.
5. **Global Surrogate Models:**
   - Building interpretable surrogate models (e.g., logistic regression) that approximate the complex behavior of the original model can provide insights into the factors influencing credit score classifications.
   - These surrogate models are simpler and easier to interpret, making it possible to understand the model's decision-making process.
6. **Domain Expertise:**

- Incorporating domain knowledge and expertise in finance and credit scoring can enhance the interpretation of model decisions.
- Domain experts can provide insights into the relevance and significance of certain features and help validate the model's decisions in the context of credit scoring.