Brief about the Variables

The dataset in this project consists of the following key variables that contribute to the predictive modelling of high-risk borrowers:

1. **borrower_age**: Age of the borrower, influencing financial stability and repayment capacity.
2. **income_bracket**: The borrower’s income category (e.g., low, middle, high), which helps assess repayment ability.
3. **marital_status**: Whether the borrower is married, single, etc., which can impact financial obligations.
4. **job_type**: The employment status of the borrower, which affects income stability.
5. **education_level**: The borrower’s highest level of education, potentially correlating with financial literacy and income potential.
6. **ethnicity**: A variable that could be used to assess potential demographic risk factors (should be used ethically and carefully).
7. **gender**: Borrower's gender, which could influence financial behaviors (to be used with caution to avoid biases).
8. **country**: The country of the borrower, as regional factors influence creditworthiness.
9. **loan_amount**: The size of the loan requested, which directly influences the risk of default.
10. **credit_score**: A traditional measure of creditworthiness, correlating with the likelihood of loan repayment.
11. **repayment_history**: Past history of loan repayment behaviour, a strong predictor of future behaviour.
12. **debt_to_income_ratio**: The ratio of the borrower’s debt obligations to their income, indicating financial stress.
13. **borrower_risk**: **The target variable** indicating whether the borrower is classified as high-risk or not.

Data Preprocessing

→Importing the Data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

- `pandas`: A powerful library used for data manipulation and analysis. It's mainly used for handling structured data in the form of DataFrames.
- `numpy`: Used for numerical operations, particularly working with arrays and matrices.
- `matplotlib.pyplot`: A plotting library for creating static, animated, and interactive visualizations in Python.
- `%matplotlib inline`: This is a magic function in Jupyter notebooks that ensures the plots are displayed inline, directly below the code cell that generates them.
- `seaborn`: Built on top of Matplotlib, Seaborn makes it easy to create attractive and informative statistical graphics.

→Importing the Dataset:

```
df = pd.read_csv("/content/10.identifying_high_risk_borrowers_dummy_data (1).csv")
```

This line loads the dataset from the CSV file into a Pandas DataFrame (df). The file is located at /content/10.identifying_high_risk_borrowers_dummy_data (1).csv.

```
df.index
df.columns
```

`df.index`: This returns the index (row labels) of the DataFrame, which typically represents the row numbers or any custom index used.

`df.columns`: This returns the column names (headers) of the DataFrame.

```
print(df.size)
print(df.shape)
```

`df.size`: This returns the total number of elements (cells) in the DataFrame. It is calculated as the number of rows multiplied by the number of columns.

`df.shape`: This returns a tuple of the form `(rows, columns)` that tells you the dimensions of the DataFrame.

→Attributes of Individual Variable:

```
df.info()
```

`df.info()`: This provides a concise summary of the DataFrame, including:

- The number of non-null values in each column.
- The datatype of each column (e.g., `int64`, `float64`, `object`).

- The memory usage of the DataFrame.

→Special Characters:

```
for columns in df.columns.values.tolist():
  print(columns)
  print(df[columns].unique())
```

- This loop iterates through each column in the DataFrame and prints the column name followed by the unique values in that column (`df[columns].unique()`).
- This helps identify if any columns contain special characters or unexpected values, and you can spot data anomalies like empty strings, strange symbols, or inconsistencies in categorical data.

Unique Values: By printing `df[columns].unique()`, you are able to identify all unique values within each column. This is particularly useful when you have categorical variables and want to see what categories or levels are present.

For example, if you have a column named `Loan Status`, you would see if there are categories like `Approved`, `Pending`, `Rejected`.

Handling Special Characters: If there are special characters or unexpected values, you may want to clean them by removing or replacing them.

→Import the Data with Special Characters :

```
df = pd.read_csv("/content/10.identifying_high_risk_borrowers_dummy_data
(1).csv",na_values=[' ', '  ', '???'])
df.isnull().sum()
df.dropna(axis = 0,inplace=True)
df.isnull().sum()
```

- Here, you are reading the CSV file into the DataFrame `df`. The parameter `na_values=[' ', ' ', '???']` specifies that any of these values (empty spaces or `'???'`) should be treated as missing values (NaN). This is useful if your dataset has placeholders like empty spaces or `'???'` to represent missing or invalid values.

- `df.isnull()` checks each cell in the DataFrame for null (NaN) values and returns a DataFrame of the same shape with `True` for NaN values and `False` for non-NaN values. `.sum()` will give you the total count of missing values (NaNs) in each column of the DataFrame. This allows you to quickly identify which columns have missing data and how many.

- `df.dropna(axis=0, inplace=True)` removes rows (`axis=0`) that contain any missing values (NaNs). `axis=0` means you're working with rows (if you wanted to drop columns, you'd use `axis=1`). `inplace=True` means the operation modifies the original DataFrame (`df`) directly, instead of returning a new DataFrame with rows removed.

```
df.shape
```

```
df['loan_amount'] = pd.to_numeric(df['loan_amount'], errors='coerce')
df['credit_score'] = pd.to_numeric(df['credit_score'], errors='coerce')
```

These two lines attempt to convert the `'loan_amount'` and `'credit_score'` columns to numeric values (e.g., integers or floats).

- `pd.to_numeric(..., errors='coerce')`: This converts the column to numeric type and, if any value cannot be converted (for example, if there are non-numeric values), it replaces those values with `NaN`. `errors='coerce'` ensures that any non-convertible values are turned into `NaN`, preventing errors from halting the code execution.

- **Why this step is important**: This ensures that your numeric columns are in the correct format for further analysis and modeling. Any non-numeric values (like text in a numerical column) would cause issues with model training.

EDA Process

```
#Summarry Stastistics
df.describe()  #Provides only for numeric data
```

`df.describe()` generates summary statistics for **numeric** columns. It includes:

- `count`: The number of non-null values.
- `mean`: The average of the column values.
- `std`: Standard deviation.
- `min`: Minimum value.
- `25%`, `50%`, `75%`: The quartiles (25th, 50th, and 75th percentiles).
- `max`: The maximum value.

```
df.describe(include='object')  #Provides only for categorical data
```

`df.describe(include='object')` generates summary statistics for **categorical** columns. It includes the `count`, `unique` values, `top` (most frequent value), and `freq` (frequency of the most frequent value).

```
df.groupby('borrower_risk')[['loan_amount', 'credit_score']].mean()
```

This groups the DataFrame by the `'borrower_risk'` column and calculates the **mean** of the `loan_amount` and `credit_score` for each borrower risk category. This is useful for comparing the average loan amount and credit score across different borrower risk categories.

```
df.groupby(['borrower_risk', 'country'])['loan_amount'].mean()
```

This groups the data by both `'borrower_risk'` and `'country'` and calculates the **mean** of `loan_amount` for each combination of borrower risk and country.

```
df.groupby(['job_type'])['repayment_history'].median()
```

This groups the data by `'job_type'` and calculates the **median** of the `repayment_history` for each job type. Using the median instead of the mean is useful when dealing with skewed data, as it's less sensitive to outliers.

→Frequency Tables:

```
pd.crosstab(index=df['borrower_risk'],columns='count',dropna=True)
```

`pd.crosstab()` creates a **frequency table** for the `'borrower_risk'` column, counting how many instances of each borrower risk category exist.

- This is useful to see the distribution of borrower risk categories in the dataset.
- `dropna=True` ensures that missing values are excluded from the count.

→Joint Probability:

```
pd.crosstab(df['borrower_risk'],df['marital_status'],normalize=True,margins=True)
```

This creates a **joint probability table** showing the relationship between `'borrower_risk'` and `'marital_status'`. The `normalize=True` argument makes the table show probabilities instead of counts (i.e., it normalizes the data to sum to 1).

- `margins=True` adds the row and column totals (marginal sums).

→Conditional Probability:

```
pd.crosstab(df['marital_status'],df['borrower_risk'],normalize='index',margins=True
)
```

This calculates the **conditional probability** between `'marital_status'` and `'borrower_risk'`. The `normalize='index'` argument normalizes the rows, so the probabilities are based on each `'marital_status'` category. This shows the probability of borrower risk categories for each marital status

```
pd.crosstab(df['marital_status'],df['borrower_risk'],normalize='columns',margins=Tr
ue)
```

This creates another **conditional probability** table, but now the columns are normalized (i.e., the probabilities are based on each `'borrower_risk'` category).

→Correlation:

```
df.select_dtypes(include=['number']).corr()
```

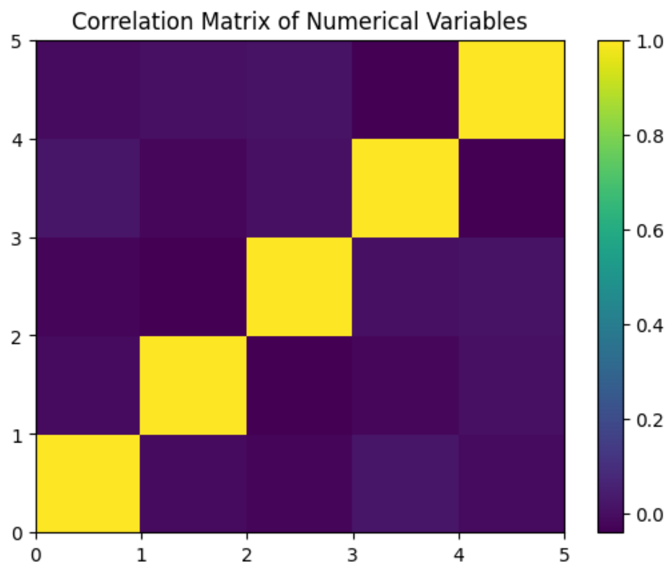
- `df.select_dtypes(include=['number'])` selects only the numeric columns from the DataFrame.
- `.corr()` calculates the **correlation matrix** between all numeric variables. The correlation values range from -1 (perfect negative correlation) to 1 (perfect positive correlation). Values near 0 indicate weak or no correlation.

```
plt.pcolor(number.corr())
plt.colorbar()
plt.title('Correlation Matrix of Numerical Variables')
plt.show()
```

The code `plt.pcolor(df.corr())` creates a **heatmap** to visualize the **correlation matrix**. The color intensity represents the strength of the correlation between numeric variables:

- **Bright colors** (e.g., dark red or purple) indicate strong correlations.
- **Fainter colors** show weaker correlations.

`plt.colorbar()` adds a **color scale** beside the heatmap, indicating the correlation range from -1 to 1 (where -1 is a perfect negative correlation, 1 is a perfect positive correlation, and 0 is no correlation).

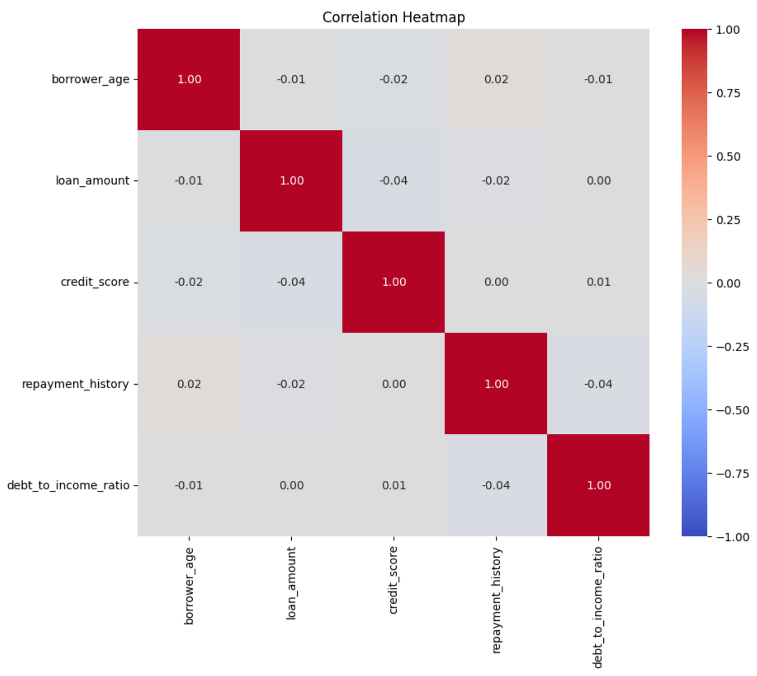


```
df['credit_score'].corr(df['repayment_history'])
df['credit_score'].corr(df['loan_amount'])
df['credit_score'].corr(df['debt_to_income_ratio'])
```

1. **`credit_score` and `repayment_history`**: **Correlation: 0.000475**
   This is a very weak positive correlation, meaning there is virtually no linear relationship between credit score and repayment history in your dataset.
2. **`credit_score` and `loan_amount`**: **Correlation: -0.0405**
   This is a very weak negative correlation, suggesting that there is almost no relationship between credit score and loan amount. As credit score slightly increases or decreases, the loan amount does not show any significant pattern.
3. **`credit_score` and `debt_to_income_ratio`**: **Correlation: 0.0084**
   This is also a very weak positive correlation, implying that there is almost no relationship between credit score and debt-to-income ratio.

Data Visualisation

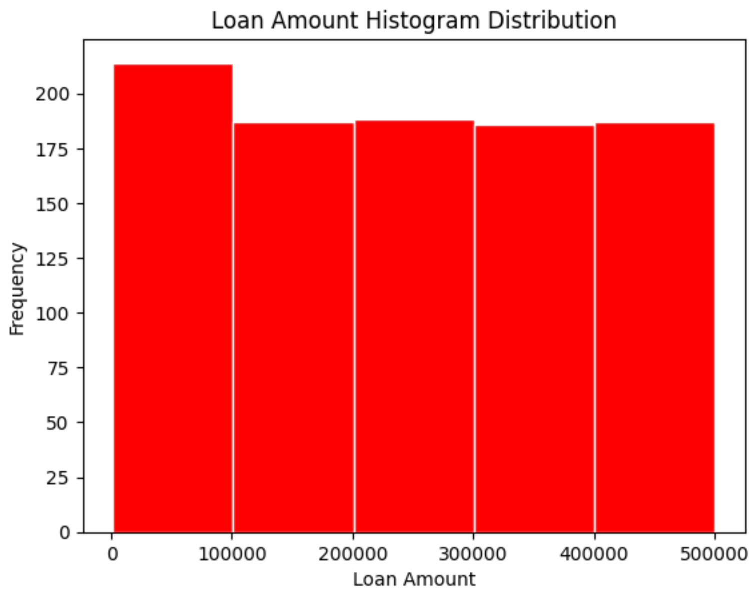
```
#CORRELATION MATRIX
numeric_df = df.select_dtypes(include=['number'])
correlation_matrix = numeric_df.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', vmin=-1,
vmax=1, cbar=True)
plt.title('Correlation Heatmap')
plt.show()
```



	borrower_age	loan_amount	credit_score	repayment_history	debt_to_income_ratio
borrower_age	1.00	-0.01	-0.02	0.02	-0.01
loan_amount	-0.01	1.00	-0.04	-0.02	0.00
credit_score	-0.02	-0.04	1.00	0.00	0.01
repayment_history	0.02	-0.02	0.00	1.00	-0.04
debt_to_income_ratio	-0.01	0.00	0.01	-0.04	1.00

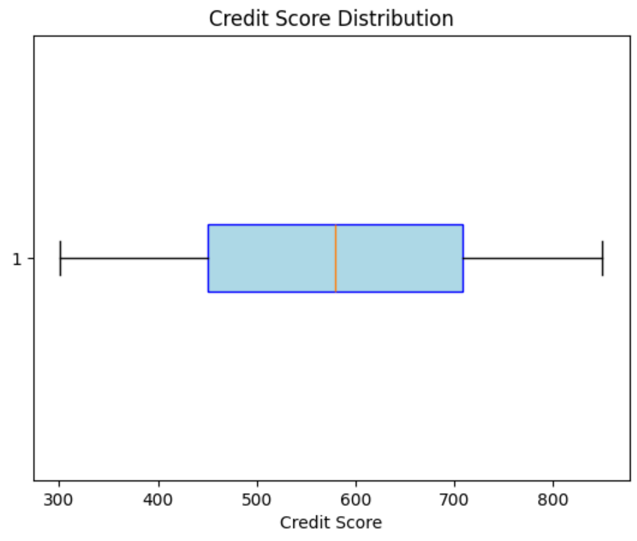
The heatmap shows weak or negligible correlations between variables, with all values close to 0. This suggests minimal relationships and low multicollinearity

```
#HISTOGRAM
plt.hist(df['loan_amount'], bins=5, edgecolor='white', color='red')
plt.title('Loan Amount Histogram Distribution')
plt.xlabel('Loan Amount')
plt.ylabel('Frequency')
plt.show()
```



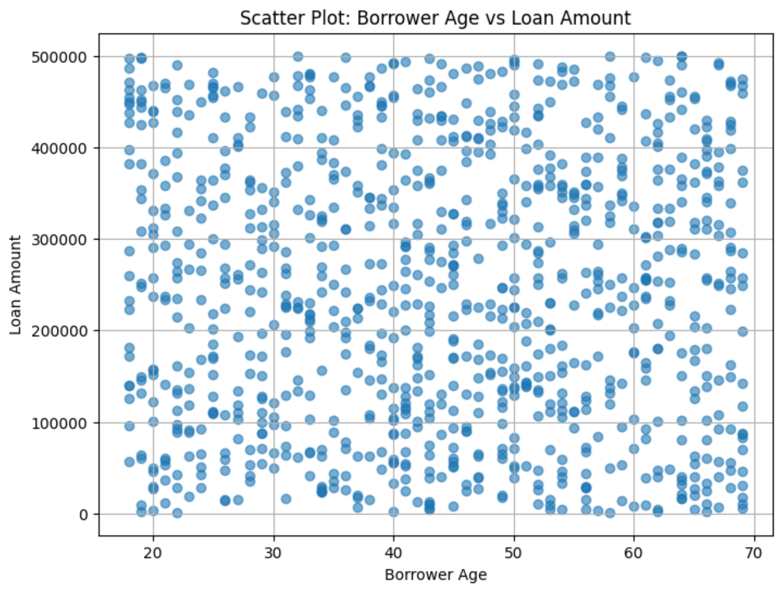
The histogram shows the distribution of loan amounts divided into five bins. Most loans fall in the lower range (0–100,000), with roughly equal frequencies in the other bins. This indicates a slightly higher concentration of smaller loan amounts.

```
#BOXPLOT
plt.boxplot(df['credit_score'], vert=False, patch_artist=True,
boxprops=dict(facecolor='lightblue', color='blue'))
plt.title('Credit Score Distribution')
plt.xlabel('Credit Score')
```



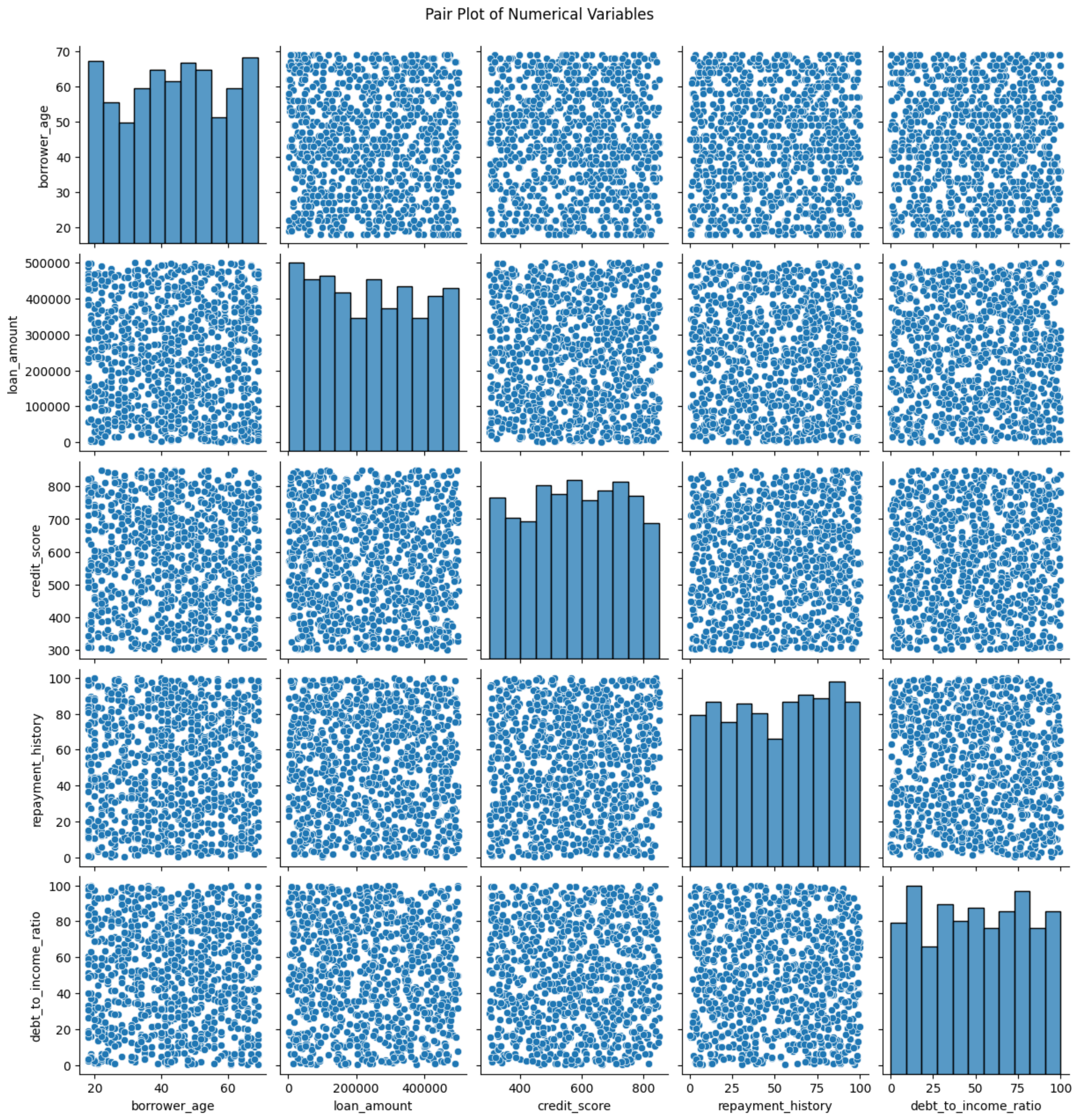
The boxplot displays the distribution of credit scores. The interquartile range (IQR) is between approximately 500 and 700, with the median near 600. There are no significant outliers, and the range spans from 300 to 800.

```
#SCATTER PLOT
plt.figure(figsize=(8, 6))
plt.scatter(df['borrower_age'], df['loan_amount'].astype(float), alpha=0.6)
plt.title('Scatter Plot: Borrower Age vs Loan Amount')
plt.xlabel('Borrower Age')
plt.ylabel('Loan Amount')
plt.grid()
plt.show()
```



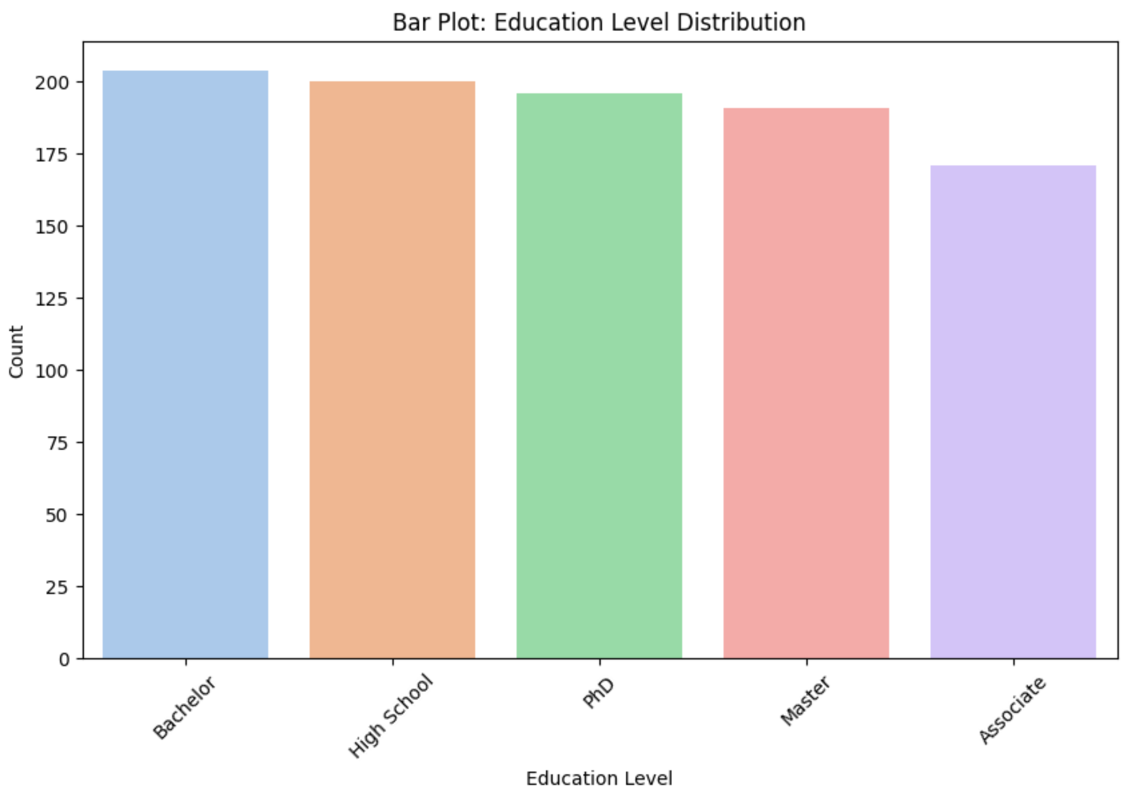
This scatter plot visualizes the relationship between borrower age and loan amount. The points are dispersed with no clear trend, suggesting no strong correlation between the two variables. Borrowers across different age groups seem to take out similar loan amounts.

```
#PAIR PLOT
sns.pairplot(df_numeric)
plt.suptitle('Pair Plot of Numerical Variables', y=1.02)
plt.show()
```



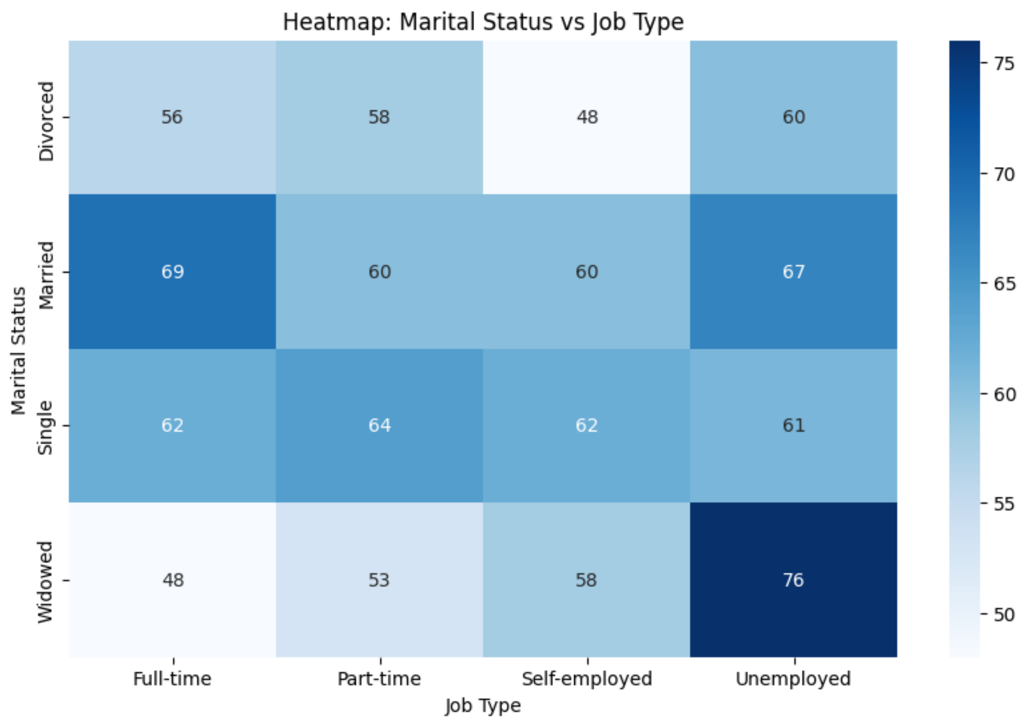
The pair plot provides a comprehensive visualization of relationships between all numerical variables in the dataset. It shows scatter plots for variable pairs and histograms for individual variables along the diagonal. This helps identify correlations, patterns, or potential outliers in the data.

```
#BAR PLOT
plt.figure(figsize=(10, 6))
education_counts = df['education_level'].value_counts()
sns.barplot(x=education_counts.index, y=education_counts.values, palette='pastel')
plt.title('Bar Plot: Education Level Distribution')
plt.xlabel('Education Level')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```



The bar plot illustrates the distribution of different education levels in the dataset. Each bar represents the count of individuals for a specific education level, with the highest and lowest counts easily identifiable. This helps understand the dataset's composition by education category.

```
#HEATMAP
cross_tab = pd.crosstab(df['marital_status'], df['job_type'])
plt.figure(figsize=(10, 6))
sns.heatmap(cross_tab, annot=True, fmt="d", cmap='Blues')
plt.title('Heatmap: Marital Status vs Job Type')
plt.xlabel('Job Type')
plt.ylabel('Marital Status')
plt.show()
```



This heatmap visualizes the relationship between marital status and job type. The values represent the count of individuals in each combination. Darker shades indicate higher counts, highlighting the most common combinations of marital status and job type in the dataset.

Model Building

```
np.unique(df['borrower_risk'])
#REDUCING THE LEVELS IN DEPENDENT VARIABLE
df['borrower_risk'] = df['borrower_risk'].map({'Low': 'Low & Medium', 'Medium':
'Low & Medium', 'High': 'High'})

unique_values = np.unique(df['borrower_risk'])
print("Updated unique values in 'borrower_risk':", unique_values)


value_counts = df['borrower_risk'].value_counts()
print("\nCounts of unique values in 'borrower_risk' after mapping:")
print(value_counts)
```

```
# Import Required Libraries
import pandas as pd
import numpy as np
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Load Your Dataset
df = pd.read_csv('your_dataset.csv')  # Replace with your dataset file

# Data Preprocessing
# Map 'borrower_risk' levels to 'Low/Medium Risk' and 'High Risk'
df['borrower_risk'] = df['borrower_risk'].map({'Low': 'Low/Medium Risk', 'Medium':
'Low/Medium Risk', 'High': 'High Risk'})

# Separate Features (X) and Target (y)
X = df.drop('borrower_risk', axis=1)  # Replace 'borrower_risk' with your target
column
y = df['borrower_risk']

# Encode Categorical Variables (if any)
X = pd.get_dummies(X, drop_first=True)

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
```

Why We Reduced the Levels in the Target Variable:

- Class Imbalance: The target variable 'borrower_risk' initially had three levels: 'Low', 'Medium', and 'High'. The distribution between these levels was likely imbalanced, leading to potential issues in model performance, as some classes (e.g., High risk) may be underrepresented.

- Simplification: Reducing the levels into two groups ('Low & Medium Risk' and 'High Risk') helps in simplifying the problem, making it a binary classification problem. This ensures that the model focuses on distinguishing between high-risk and low-to-medium-risk borrowers, which might be more aligned with real-world decision-making.

- Improved Model Performance: By merging categories, we improve class distribution and give the model a better chance to learn, particularly when dealing with highly imbalanced datasets.

```
#SMOTE AND RANDOM FOREST
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

X = df.drop('borrower_risk', axis=1)
y = df['borrower_risk']

X = pd.get_dummies(X, drop_first=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

print("Class Distribution Before SMOTE:\n", y_train.value_counts())
print("\nClass Distribution After SMOTE:\n", y_train_smote.value_counts())

model = RandomForestClassifier(random_state=42)
model.fit(X_train_smote, y_train_smote)

y_pred = model.predict(X_test)
```

```
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Why We Used SMOTE (Synthetic Minority Over-sampling Technique):

- Class Imbalance: Before applying SMOTE, the dataset had an imbalance between the classes, particularly for the 'High' risk category. The model tends to predict the majority class (in this case, 'Low & Medium Risk') better, leading to poor performance on the minority class (i.e., 'High Risk').

- SMOTE's Role: SMOTE creates synthetic samples for the minority class (i.e., 'High Risk'), thereby balancing the class distribution. After applying SMOTE, both 'Low & Medium Risk' and 'High Risk' classes had the same number of samples, which improved model learning and performance.

Using SMOTE helps to avoid bias towards the majority class and allows the model to learn to distinguish better between the two categories. However, even with SMOTE, the model's performance might still be limited due to factors like data quality, feature engineering, or the complexity of the model.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix


X = df.drop('borrower_risk', axis=1)  # Replace 'borrower_risk' with your target
column
y = df['borrower_risk']


X = pd.get_dummies(X, drop_first=True)


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)


from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

print("Class Distribution Before SMOTE:\n", y_train.value_counts())
print("\nClass Distribution After SMOTE:\n", y_train_smote.value_counts())


classifiers = {
    'Logistic Regression': LogisticRegression(random_state=42),
    'K-Nearest Neighbors (KNN)': KNeighborsClassifier(),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Support Vector Machine (SVM)': SVC(random_state=42),
    'Naive Bayes': GaussianNB(),
    'Random Forest': RandomForestClassifier(random_state=42)
}

for model_name, model in classifiers.items():
    print(f"\nTraining {model_name}...")


    model.fit(X_train_smote, y_train_smote)


    y_pred = model.predict(X_test)

    print(f"\nConfusion Matrix for {model_name}:")
    print(confusion_matrix(y_test, y_pred))

    print(f"\nClassification Report for {model_name}:")
    print(classification_report(y_test, y_pred))
```

Model Accuracy Comparison Table

Model	**Accuracy**
Logistic Regression	**52%**
K-Nearest Neighbors (KNN)	**52%**
Decision Tree	**54%**
Support Vector Machine (SVM)	**48%**
Naive Bayes	**50%**
Random Forest	**60%**

While the **Random Forest** model yielded the highest accuracy at **60%**, the overall accuracy across all models is not very high. This indicates that the current model configurations might not be sufficient to accurately predict borrower risk.

In the next step, we will tweak the model using other methods such as **hyperparameter tuning**, **feature engineering**, and exploring different classifiers to improve the accuracy. This may involve experimenting with more models and adjusting their parameters for better performance.

```
# Import necessary libraries
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score, classification_report, confusion_matrix
from sklearn.preprocessing import StandardScaler
```

```python
from imblearn.over_sampling import SMOTE
import pandas as pd
import numpy as np

# Load the dataset
file_path = "10.identifying_high_risk_borrowers_dummy_data (1).csv"  # Replace with
your file path
df = pd.read_csv(file_path)

# Clean the data
df.replace({' ': np.nan, '  ': np.nan, '???': np.nan}, inplace=True)
df['borrower_risk'] = df['borrower_risk'].map({'Low': 0, 'Medium': 0, 'High': 1})
df.fillna(df.mean(numeric_only=True), inplace=True)
df.fillna(df.mode().iloc[0], inplace=True)

# Encode categorical variables if necessary
df = pd.get_dummies(df, drop_first=True)

# Define features and target variable
X = df.drop('borrower_risk', axis=1)
y = df['borrower_risk']

# Handle class imbalance using SMOTE
smote = SMOTE(random_state=42)
X, y = smote.fit_resample(X, y)

# Feature scaling
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)

# Initialize models with probability=True for SVC
models = {
    'Logistic Regression': LogisticRegression(),
    'Naive Bayes': GaussianNB(),
    'KNN': KNeighborsClassifier(),
    'SVM': SVC(probability=True),  # Set probability=True to allow predict_proba
    'Decision Tree': DecisionTreeClassifier(),
    'Random Forest': RandomForestClassifier(random_state=42,
class_weight='balanced')
}

# Function to train, predict, and evaluate models
def evaluate_models(models, X_train, X_test, y_train, y_test):
    results = []
    for model_name, model in models.items():
        # Train the model
        model.fit(X_train, y_train)

        # Make predictions
        y_pred = model.predict(X_test)
        # Calculate metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
        recall = recall_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred)

        # Calculate ROC AUC score
        if hasattr(model, 'predict_proba'):
            roc_auc = roc_auc_score(y_test, model.predict_proba(X_test)[:,1])
        else:
            roc_auc = np.nan  # If the model doesn't support predict_proba, set ROC
AUC as NaN

        # Append results to list
        results.append({
            'Model': model_name,
            'Accuracy': accuracy * 100,
            'Precision': precision,
            'Recall': recall,
            'F1 Score': f1,
            'ROC AUC': roc_auc
        })
    return pd.DataFrame(results)

# Evaluate all models
results_df = evaluate_models(models, X_train, X_test, y_train, y_test)

# Display the results
print(results_df)
```

Improvements from the Previous Step:

- Feature Scaling: Applied Standard Scaling to the features. Scaling ensures that all features are on the same scale, which is especially important for distance-based algorithms (like KNN and SVM) to prevent features with larger ranges from dominating the model.

- Model Tuning:

→Used balanced class weights in the Random Forest model to handle class imbalance more effectively during training.

→Set probability=True in the SVM model to allow for the calculation of ROC AUC scores, which gives us a better idea of how well the model differentiates between classes.

Model Accuracy Comparison Table:

Model	Accuracy	Precision	Recall	F1 Score	ROC AUC
Logistic Regression	50.00%	0.50	1.00	0.67	0.74
Naive Bayes	50.00%	0.50	1.00	0.67	0.50
K-Nearest Neighbors (KNN)	50.00%	0.50	1.00	0.67	0.50
Support Vector Machine (SVM)	72.93%	1.00	0.46	0.63	0.74
Decision Tree	65.79%	0.66	0.65	0.66	0.66
Random Forest	58.27%	0.55	0.92	0.69	0.77

Analysis of Key Metrics:

1) **Accuracy**:

- **SVM** leads with **72.93%**, outperforming others in distinguishing classes.
- **Logistic Regression**, **Naive Bayes**, and **KNN** are at **50%**, likely due to class imbalance.
- **Decision Tree** and **Random Forest** performed better with **65.79%** and **58.27%**, respectively.

2) **Precision**:

- **SVM** has a perfect precision of **1.00**, meaning no false positives.
- **Logistic Regression**, **Naive Bayes**, and **KNN** have **0.50**, indicating only half of positive predictions are correct.

3) **Recall**:

- **Logistic Regression**, **Naive Bayes**, and **KNN** have a **1.00** recall, identifying all true positives but with high false positives.
- **SVM** has a **0.46** recall, missing half of the high-risk cases.
- **Random Forest** performs well with **0.92** recall, identifying most high-risk cases.

4) **F1 Score**:

- **SVM** and **Random Forest** show better balance with **0.63** and **0.69**, respectively.
- **Logistic Regression**, **Naive Bayes**, and **KNN** have a **0.67** score.

5) **ROC AUC**:

- **Random Forest** leads with **0.77**, followed by **SVM** at **0.74**.
- **Logistic Regression** is decent at **0.74**, while **Decision Tree** is solid at **0.66**.
- **Naive Bayes** and **KNN** have the lowest at **0.50**, struggling to separate classes effectively.

Conclusion and References

Model Selection Note:

Based on the analysis of key metrics, the choice of the best model depends on the specific priorities of the task:

- **For prioritizing accuracy**, the **SVM model** stands out with an accuracy of **72.93%**, making it the top performer in distinguishing between classes.
- **For prioritizing recall** (i.e., correctly identifying high-risk borrowers) and achieving a better balance between precision and recall, the **Random Forest model** is the best choice, with a **recall of 0.92** and an **F1 score of 0.69**.

Conclusion:

- **Random Forest** is the optimal model for a balanced approach, excelling in recall and maintaining strong performance across other metrics.
- **SVM** is the best for scenarios where overall accuracy is the primary focus.

→Key Findings:

1. Model Performance:

- Various models were evaluated for predicting high-risk borrowers, including Logistic Regression, Naive Bayes, K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Decision Tree, and Random Forest.

- The SVM model achieved the highest accuracy at 72.93%, making it the best choice for scenarios prioritizing overall accuracy.

- The Random Forest model, with a recall of 0.92 and an F1 score of 0.69, excelled in identifying high-risk borrowers, providing a good balance between precision and recall.

2. Class Imbalance Handling:

- SMOTE (Synthetic Minority Over-sampling Technique) was used to balance the dataset, addressing class imbalance by generating synthetic samples for the minority class (high-risk borrowers).

- This technique helped improve model performance, especially in handling the underrepresented high-risk class.

3. Implications for Predictive Modeling:

- Random Forest is recommended for applications where accurately identifying high-risk borrowers is critical, as its high recall ensures fewer high-risk cases are missed.

- SVM is preferred when overall accuracy is the primary goal, as it performed best in distinguishing between the high and low-risk categories.

→Scope for Improvement of the Model:

While the current models provide valuable insights, there is significant potential for further improvement to enhance prediction accuracy and robustness:

1. **Hyperparameter Tuning**:
   Optimizing the hyperparameters of models such as **Random Forest** and **SVM** through techniques like **Grid Search** or **Random Search** can improve performance. Tuning parameters like the number of trees in Random Forest or the kernel in SVM could lead to better class separation and higher accuracy.
2. **Feature Engineering**:
   Creating new features or transforming existing ones (e.g., binning continuous variables, handling outliers, or encoding additional categorical variables) could help the models capture more complex relationships in the data, leading to better predictions.
3. **Ensemble Methods**:
   Combining multiple models through techniques like **Voting Classifier** or **Stacking** can leverage the strengths of different models and improve overall performance. An ensemble of **Random Forest**, **SVM**, and other classifiers could achieve better generalization.
4. **Advanced Resampling Techniques**:
   Besides SMOTE, other resampling techniques like **ADASYN (Adaptive Synthetic Sampling)** or **NearMiss** could be explored to further balance the classes and improve model training on imbalanced datasets.
5. **Cross-Validation**:
   Implementing **k-fold cross-validation** ensures that the model performance is evaluated on multiple subsets of the data, reducing the risk of overfitting and providing a more reliable performance estimate.
6. **Deep Learning Models**:
   Exploring advanced deep learning models, such as **neural networks**, could provide an even more powerful solution, especially if the dataset grows larger and more complex. These models can learn intricate patterns that traditional models may miss.