Goals and Project Structure

Setting up the Project

Mount Drive

```
# mount to google drive
# from google.colab import drive
# drive.mount('/content/drive')
```

Import Libraries

> - will be importing mor elibraries as they become relevant in the project

```
pip install --upgrade numba ydata-profiling visions
```

```
Requirement already satisfied: numba in /usr/local/lib/python3.11/dist-packages (0.61.0)
Requirement already satisfied: ydata-profiling in /usr/local/lib/python3.11/dist-packages (4.12.2)
Requirement already satisfied: visions in /usr/local/lib/python3.11/dist-packages (0.7.6)
Collecting visions
  Using cached visions-0.8.1-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: llvmlite<0.45,>=0.44.0dev0 in /usr/local/lib/python3.11/dist-packages (from numba) (0.44.0)
Requirement already satisfied: numpy<2.2,>=1.24 in /usr/local/lib/python3.11/dist-packages (from numba) (1.26.4)
Requirement already satisfied: scipy<1.16,>=1.4.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.13.1)
Requirement already satisfied: pandas!=1.4.0,<3,>1.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (2.2.2)
Requirement already satisfied: matplotlib>=3.5 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (3.10.0)
Requirement already satisfied: pydantic>=2 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (2.10.6)
Requirement already satisfied: PyYAML<6.1,>=5.0.0 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (6.0.2)
Requirement already satisfied: jinja2<3.2,>=2.11.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (3.1.5)
Requirement already satisfied: htmlmin==0.1.12 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.1.12)
Requirement already satisfied: phik<0.13,>=0.11.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.12.4)
Requirement already satisfied: requests<3,>=2.24.0 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (2.32.3)
Requirement already satisfied: tqdm<5,>=4.48.2 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (4.67.1)
Requirement already satisfied: seaborn<0.14,>=0.10.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.13.2)
Requirement already satisfied: multimethod<2,>=1.4 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.12)
Requirement already satisfied: statsmodels<1,>=0.13.2 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.14.4)
Requirement already satisfied: typeguard<5,>=3 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (4.4.1)
Requirement already satisfied: imagehash==4.3.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (4.3.1)
Requirement already satisfied: wordcloud>=1.9.3 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.9.4)
Requirement already satisfied: dacite>=1.8 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.9.2)
Requirement already satisfied: PyWavelets in /usr/local/lib/python3.11/dist-packages (from imagehash==4.3.1->ydata-profiling) (1.8.0)
Requirement already satisfied: pillow in /usr/local/lib/python3.11/dist-packages (from imagehash==4.3.1->ydata-profiling) (11.1.0)
Requirement already satisfied: attrs>=19.3.0 in /usr/local/lib/python3.11/dist-packages (from visions) (25.1.0)
Requirement already satisfied: networkx>=2.4 in /usr/local/lib/python3.11/dist-packages (from visions) (3.4.2)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2<3.2,>=2.11.1->ydata-profiling) (3
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.5->ydata-profiling) (1.3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.5->ydata-profiling) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.5->ydata-profiling) (4.5
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.5->ydata-profiling) (1.4
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.5->ydata-profiling) (24.2)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.5->ydata-profiling) (3.2.
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.5->ydata-profiling) (
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas!=1.4.0,<3,>1.1->ydata-profiling) (20
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas!=1.4.0,<3,>1.1->ydata-profiling) (
Requirement already satisfied: joblib>=0.14.1 in /usr/local/lib/python3.11/dist-packages (from phik<0.13,>=0.11.1->ydata-profiling) (1.4
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.11/dist-packages (from pydantic>=2->ydata-profiling) (0.
Requirement already satisfied: pydantic-core==2.27.2 in /usr/local/lib/python3.11/dist-packages (from pydantic>=2->ydata-profiling) (2.2
Requirement already satisfied: typing-extensions>=4.12.2 in /usr/local/lib/python3.11/dist-packages (from pydantic>=2->ydata-profiling)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata-prof
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata-profiling) (3.10
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata-profiling)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata-profiling)
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.11/dist-packages (from statsmodels<1,>=0.13.2->ydata-profiling) (1
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib>=3.5->ydata-pr
```

```
import numpy as np, pandas as pd, matplotlib.pyplot as plt, seaborn as sns
from scipy import stats
from ydata_profiling import ProfileReport
```

Load Dataset

```
url = '/content/drive/MyDrive/Data_Science_Projects/Loan Default/Loan_default.csv'
df = pd.read_csv(url)
df_copy = df.copy()
# quick glance at the data and ensure it is loaded properly
df.head()
```

	LoanID	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Education	Emplo
0	I38PQUQS96	56	85994	50587	520	80	4	15.23	36	0.44	Bachelor's	
1	HPSK72WA7R	69	50432	124440	458	15	1	4.81	60	0.68	Master's	
2	C1OZ6DPJ8Y	46	84208	129188	451	26	3	21.17	24	0.31	Master's	U
3	V2KKSFM3UN	32	31713	44799	743	0	3	7.07	24	0.23	High School	
4	EY08JDHTZP	60	20437	9139	633	8	4	6.51	48	0.73	Bachelor's	U

Inspecting the Data

Shape

```
shape = df.shape
print(f'The dataset has {shape[0]} rows and {shape[1]} columns.')
```

```
The dataset has 255347 rows and 18 columns.
```

Data Types

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 255347 entries, 0 to 255346
Data columns (total 18 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   LoanID          255347 non-null  object 
 1   Age             255347 non-null  int64  
 2   Income          255347 non-null  int64  
 3   LoanAmount      255347 non-null  int64  
 4   CreditScore     255347 non-null  int64  
 5   MonthsEmployed  255347 non-null  int64  
 6   NumCreditLines  255347 non-null  int64  
 7   InterestRate    255347 non-null  float64
 8   LoanTerm        255347 non-null  int64  
 9   DTIRatio        255347 non-null  float64
 10  Education       255347 non-null  object 
 11  EmploymentType  255347 non-null  object 
 12  MaritalStatus   255347 non-null  object 
 13  HasMortgage     255347 non-null  object 
 14  HasDependents   255347 non-null  object 
 15  LoanPurpose     255347 non-null  object 
 16  HasCoSigner     255347 non-null  object 
 17  Default         255347 non-null  int64  
dtypes: float64(2), int64(8), object(8)
memory usage: 35.1+ MB
```

Missing Values

```
missing_vals = df.isna().sum()

if missing_vals.any():
  print('There are missing values in the dataset.')
else:
  print('There are no missing values in the dataset.')
```

```
7
8 missing_vals
```

```
There are no missing values in the dataset.
```

	0
LoanID	0
Age	0
Income	0
LoanAmount	0
CreditScore	0
MonthsEmployed	0
NumCreditLines	0
InterestRate	0
LoanTerm	0
DTIRatio	0
Education	0
EmploymentType	0
MaritalStatus	0
HasMortgage	0
HasDependents	0
LoanPurpose	0
HasCoSigner	0
Default	0

dtype: int64

Duplicate Values

```
1 duplicate_vals = df.duplicated().sum()
2
3 if duplicate_vals > 0:
4   print(f'There are {duplicate_vals} duplicate rows in the dataset.')
5 else:
6   print('There are no duplicate rows in the dataset.')
```

```
There are no duplicate rows in the dataset.
```

Numerical Data

```
1 df.describe()
```

	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRa
count	255347.000000	255347.000000	255347.000000	255347.000000	255347.000000	255347.000000	255347.000000	255347.000000	255347.000
mean	43.498306	82499.304597	127578.865512	574.264346	59.541976	2.501036	13.492773	36.025894	0.500
std	14.990258	38963.013729	70840.706142	158.903867	34.643376	1.117018	6.636443	16.969330	0.230
min	18.000000	15000.000000	5000.000000	300.000000	0.000000	1.000000	2.000000	12.000000	0.100
25%	31.000000	48825.500000	66156.000000	437.000000	30.000000	2.000000	7.770000	24.000000	0.300
50%	43.000000	82466.000000	127556.000000	574.000000	60.000000	2.000000	13.460000	36.000000	0.500
75%	56.000000	116219.000000	188985.000000	712.000000	90.000000	3.000000	19.250000	48.000000	0.700
max	69.000000	149999.000000	249999.000000	849.000000	119.000000	4.000000	25.000000	60.000000	0.900

Looking at our means we see on average:

- Most customers are ~ 44 years old and make ~$82,500 a year
- The average loan amount per customer is $127,578 for a term of ~17 months at an interest rate of approximately 13.5%
- The average credit score of a customer is 574 with a 0.5 DTI ratio

Categorical Data

```
1 df.describe(include='object')
```

	LoanID	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner
count	255347	255347	255347	255347	255347	255347	255347	255347
unique	255347	4	4	3	2	2	5	2
top	ZTH91CGL0B	Bachelor's	Part-time	Married	Yes	Yes	Business	Yes
freq	1	64366	64161	85302	127677	127742	51298	127701

From looking at our categorical data we see that:

- ~25% of customers have a Bachelor's degree, with this being the most common level of education achieved by the customer
- ~25% of customers also work part time
- ~50% of customers have a mortgage, a dependent, and/or a cosigner

Detecting Outliers

- First we find the Interquartile Range (middle 50% of our data)
- Using the IQR we determine our upper and lower bounds
    - $\text{LowerBound} = \text{Q1} - 1.5 \times \text{IQR}$
    - $\text{UpperBound} = \text{Q3} + 1.5 \times \text{IQR}$
- Any value **below** the *lower bound* and **above** the *upper bound* is an outlier

```
# exclude the 'Default' column
# 'Default' column uses 1, 0 instead of yes, no
# not needed in outlier detection
df_no_default = df.drop('Default', axis=1)

def find_outliers(df):
  outlier_count = {}
  for col in df.select_dtypes(include='number'):
      Q1 = df[col].quantile(0.25)
      Q3 = df[col].quantile(0.75)
      IQR = Q3 - Q1
      lower_bound = Q1 - 1.5 * IQR
      upper_bound = Q3 + 1.5 * IQR
      outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
      outlier_count[col] = outliers.shape[0]
  return outlier_count

outliers_iqr = find_outliers(df_no_default)
outliers_df = pd.DataFrame(list(outliers_iqr.items()), columns=['Column', 'Outlier Count'])
outliers_df
```

	Column	Outlier Count
0	Age	0
1	Income	0
2	LoanAmount	0
3	CreditScore	0
4	MonthsEmployed	0
5	NumCreditLines	0
6	InterestRate	0
7	LoanTerm	0
8	DTIRatio	0

Next steps: Generate code with `outliers_df` | View recommended plots | New interactive sheet

Exploratory Data Analysis

```
1 eda = ProfileReport(df,  title = 'EDA REPORT ')
2 eda
```

Summarize dataset: 100% 76/76 [00:21<00:00, 4.48it/s, Completed]

Generate report structure: 100% 1/1 [00:04<00:00, 4.65s/it]

Render HTML: 100% 1/1 [00:01<00:00, 1.51s/it]

EDA REPORT

Overview Variables Interactions Correlations Missing values Sample

Overview

Brought to you by YData

Overview | Alerts 1 | Reproduction

Dataset statistics

Number of variables	18
Number of observations	255347
Missing cells	0
Missing cells (%)	0.0%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	35.1 MiB
Average record size in memory	144.0 B

Variable types

Text	1
Numeric	7
Categorical	7
Boolean	3

Variables

Select Columns

Some Observations:

> - Our data is not normally distributed, rather most if it is roughly uniform, which means we need to use non-parametric models
> - Our data also has low correlation, no one feature is affected by the other

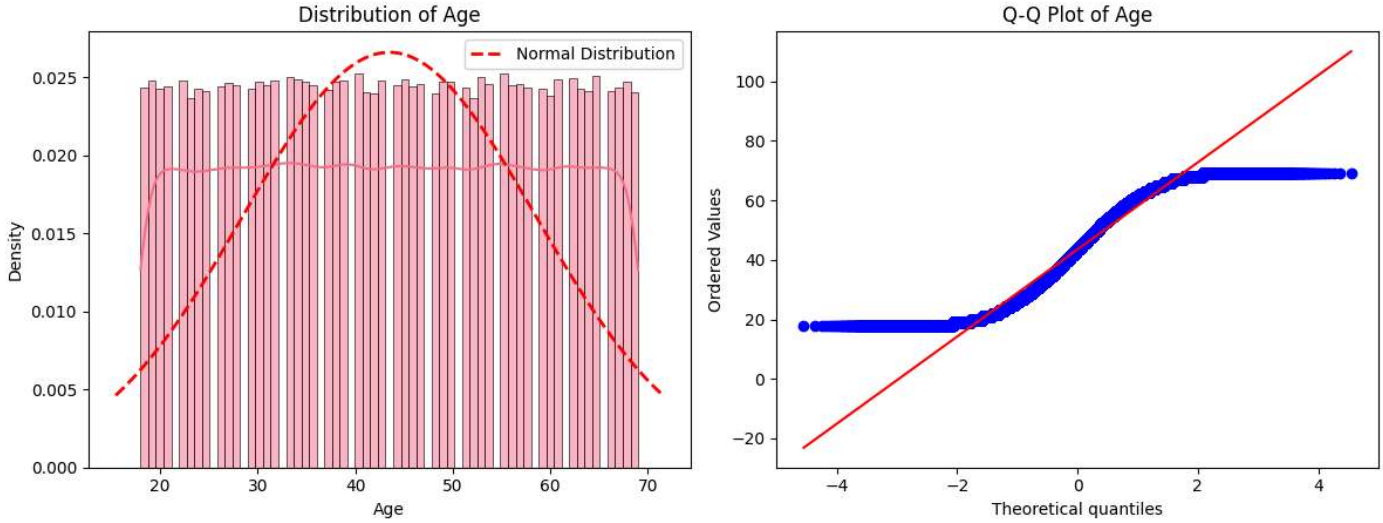
Normality Check

```
def plot_normality_check(data, feature, figsize=(12, 5)):
    """
    Create side-by-side plots to check normality of a numerical feature:
    1. Histogram with KDE and normal distribution overlay
    2. Q-Q plot

    Parameters:
    -----------
    data : pandas.DataFrame
        The dataframe containing the feature
    feature : str
        The name of the feature to analyze
    figsize : tuple, optional
        Figure size for the plot (width, height)

    Returns:
    --------
    None: Displays the plots
    """

    # Create figure and axes for side-by-side plots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=figsize)

    # Get the data for the feature
    feature_data = data[feature].dropna()

    # Plot 1: Histogram with KDE
    sns.histplot(data=feature_data, stat='density', kde=True, ax=ax1)

    # Add normal distribution curve
    xmin, xmax = ax1.get_xlim()
    x = np.linspace(xmin, xmax, 100)
    mu, std = feature_data.mean(), feature_data.std()
    p = stats.norm.pdf(x, mu, std)
    ax1.plot(x, p, 'r--', linewidth=2, label='Normal Distribution')

    # Customize first plot
    ax1.set_title(f'Distribution of {feature}')
    ax1.set_xlabel(feature)
    ax1.set_ylabel('Density')
    ax1.legend()

    # Plot 2: Q-Q plot
    stats.probplot(feature_data, dist="norm", plot=ax2)
    ax2.set_title(f'Q-Q Plot of {feature}')

    # Add overall title and adjust layout
    plt.suptitle(f'Normality Check for {feature}', y=1.05)
    plt.tight_layout()

    # Show plot
    plt.show()

    # Print Shapiro-Wilk test results
    stat, p_value = stats.shapiro(feature_data.sample(min(5000, len(feature_data))))
    print(f'\nShapiro-Wilk Test Results for {feature}:')
    print(f'Statistic: {stat:.4f}')
    print(f'p-value: {p_value:.4f}')
    if p_value < 0.05:
        print('The feature is not normally distributed (rejects null hypothesis)')
    else:
        print('The feature appears to be normally distributed (fails to reject null hypothesis)')

    # Print skewness and kurtosis
    print(f'\nSkewness: {stats.skew(feature_data):.4f}')
```

```
    print(f'Kurtosis: {stats.kurtosis(feature_data):.4f}')

# Example usage for multiple features:
def check_all_numeric_features(data, exclude_cols=None):
    """
    Check normality for all numeric features in the dataset

    Parameters:
    -----------
    data : pandas.DataFrame
        The dataframe to analyze
    exclude_cols : list, optional
        List of column names to exclude from analysis

    Returns:
    --------
    None: Displays plots for each numeric feature
    """

    # Get numeric columns
    numeric_cols = data.select_dtypes(include=['int64', 'float64']).columns

    # Exclude specified columns
    if exclude_cols:
        numeric_cols = [col for col in numeric_cols if col not in exclude_cols]

    # Plot normality check for each numeric feature
    for feature in numeric_cols:
        plot_normality_check(data, feature)
        print('\n' + '='*50 + '\n')
```

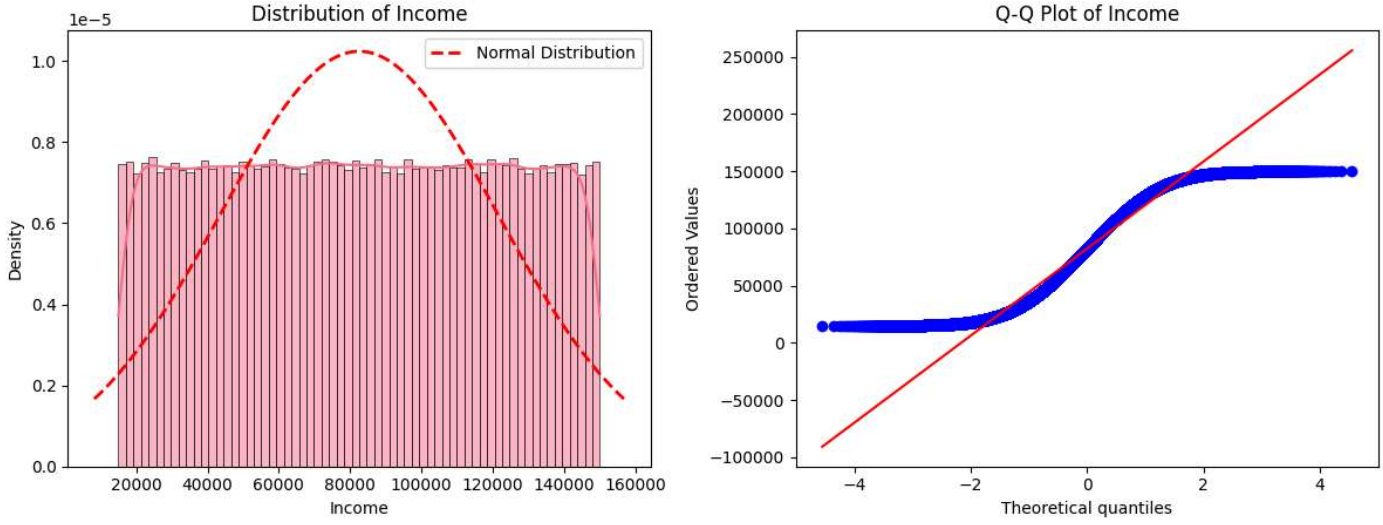
```
numeric_cols = df.select_dtypes(include=['int64', 'float64'])

check_all_numeric_features(numeric_cols, exclude_cols=['LoanID', 'Default', 'NumCreditLines', 'LoanTerm'])
```

Normality Check for Age



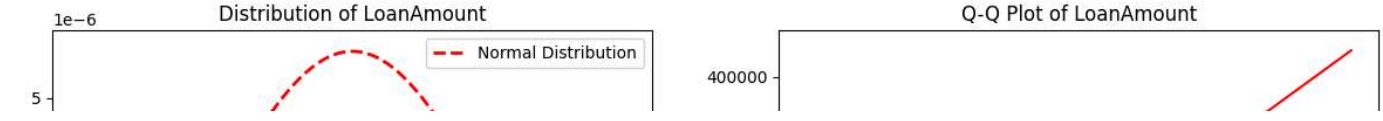

```
Shapiro-Wilk Test Results for Age:
Statistic: 0.9528
p-value: 0.0000
The feature is not normally distributed (rejects null hypothesis)

Skewness: 0.0007
Kurtosis: -1.1984

==================================================
```

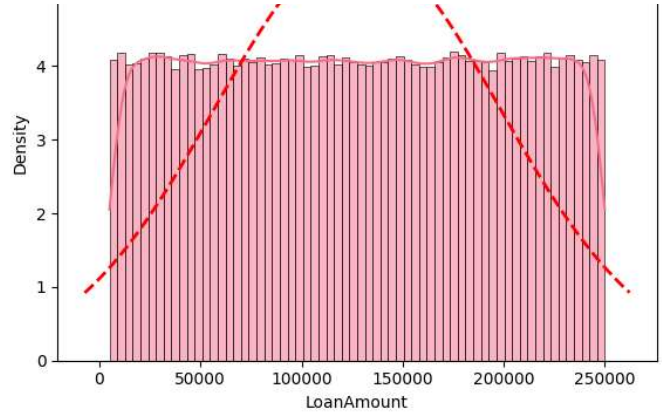
Normality Check for Income

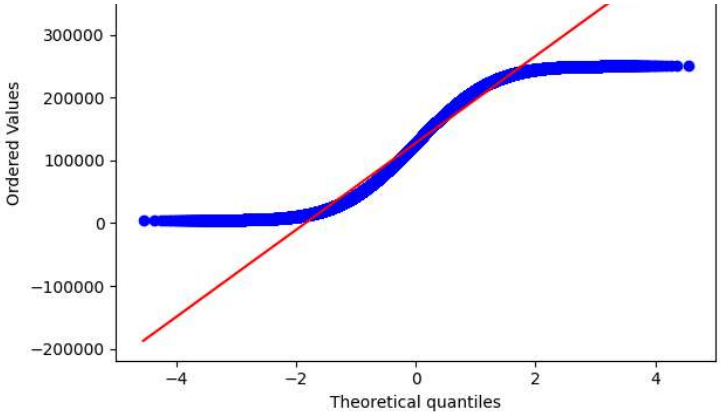



```
Shapiro-Wilk Test Results for Income:
Statistic: 0.9559
p-value: 0.0000
The feature is not normally distributed (rejects null hypothesis)

Skewness: -0.0004
Kurtosis: -1.1984

==================================================
```

Normality Check for LoanAmount

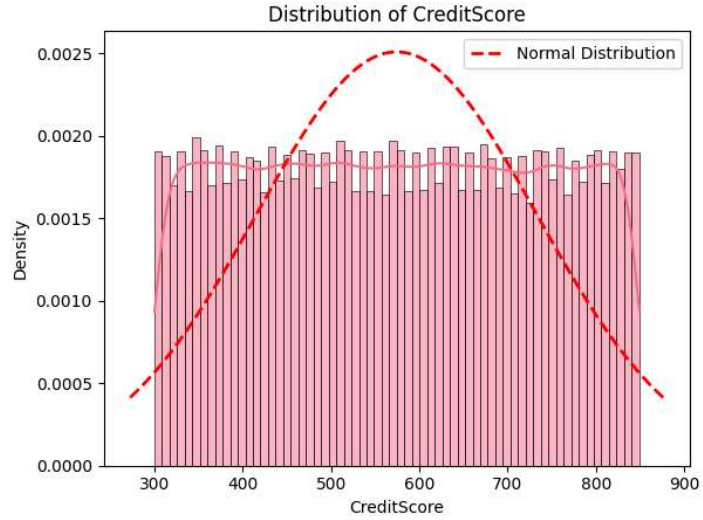


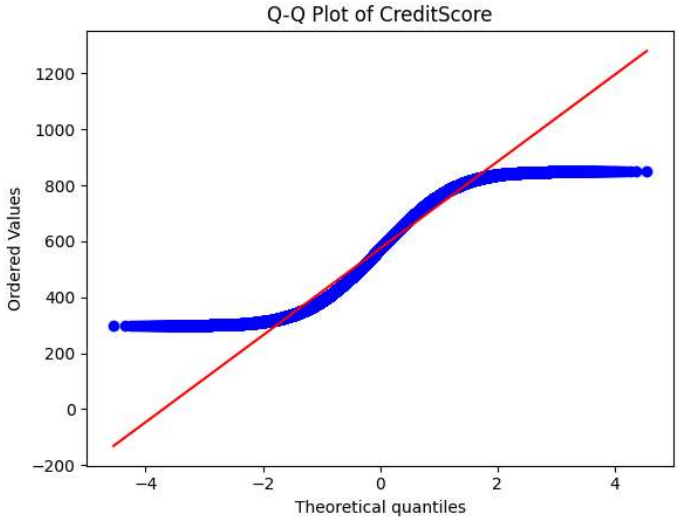





```
Shapiro-Wilk Test Results for LoanAmount:
Statistic: 0.9550
p-value: 0.0000
The feature is not normally distributed (rejects null hypothesis)

Skewness: -0.0018
Kurtosis: -1.2037

==================================================
```

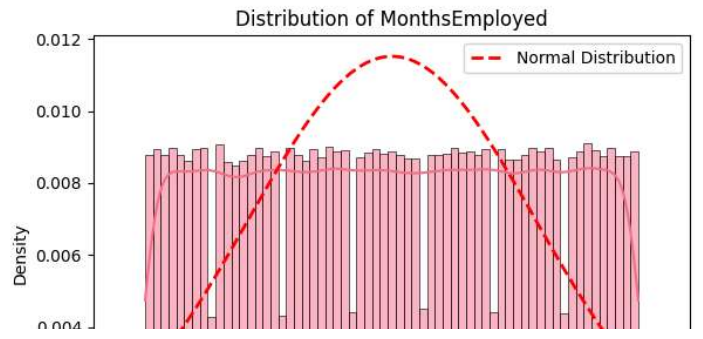
Normality Check for CreditScore

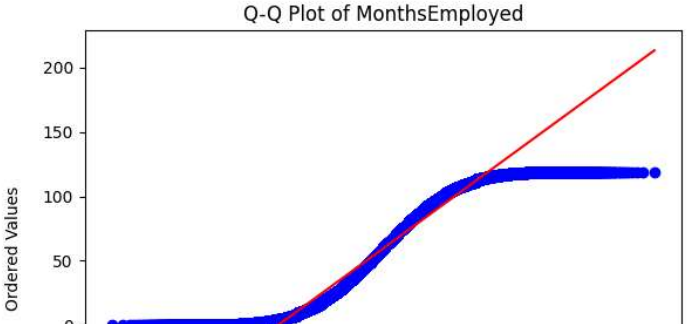


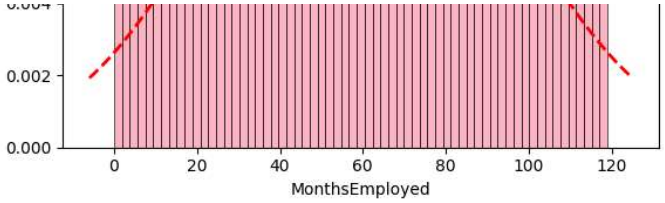


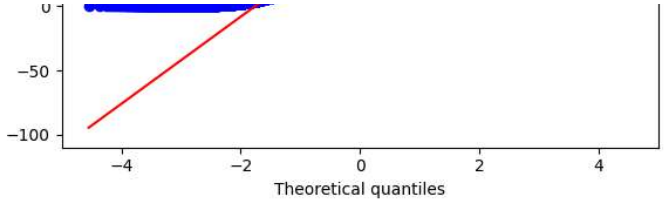

```
Shapiro-Wilk Test Results for CreditScore:
Statistic: 0.9526
p-value: 0.0000
The feature is not normally distributed (rejects null hypothesis)

Skewness: 0.0047
Kurtosis: -1.2003

==================================================
```

Normality Check for MonthsEmployed



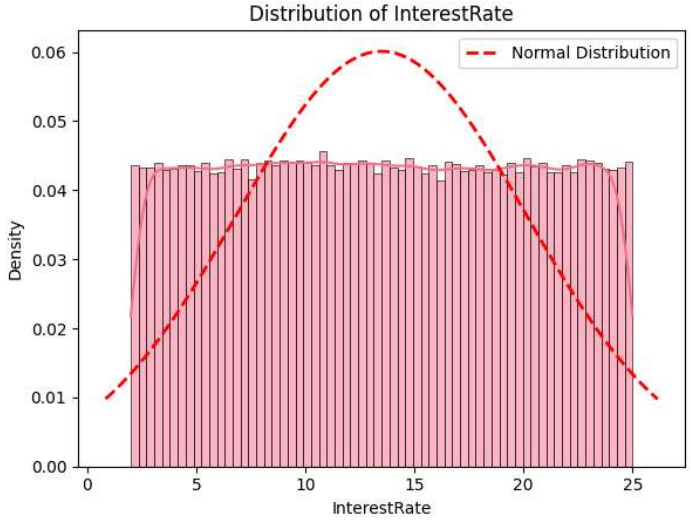


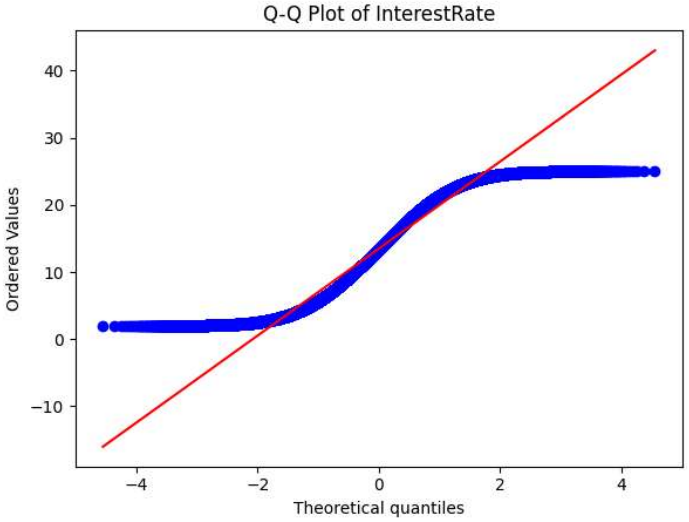





```
Shapiro-Wilk Test Results for MonthsEmployed:
Statistic: 0.9531
p-value: 0.0000
The feature is not normally distributed (rejects null hypothesis)

Skewness: -0.0021
Kurtosis: -1.1996

==================================================
```

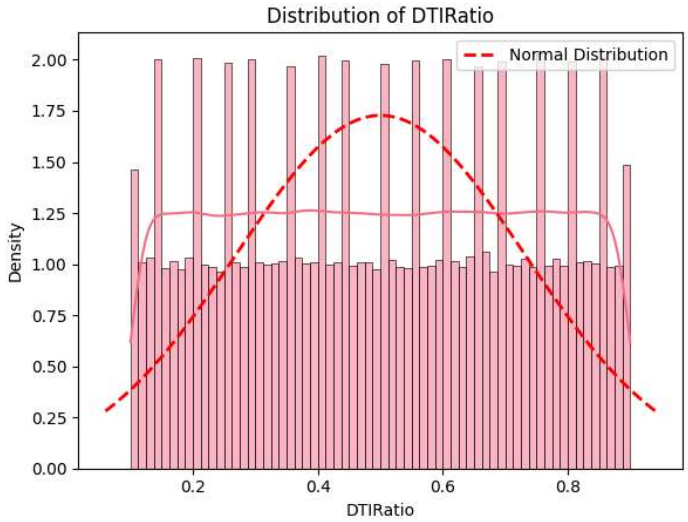
Normality Check for InterestRate

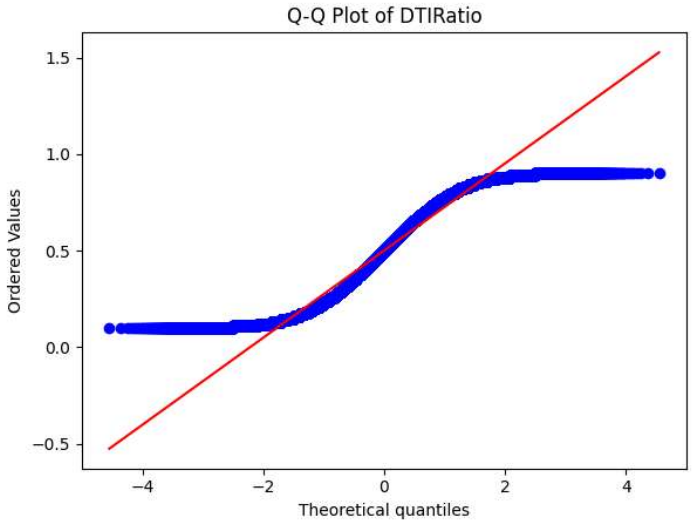





```
Shapiro-Wilk Test Results for InterestRate:
Statistic: 0.9566
p-value: 0.0000
The feature is not normally distributed (rejects null hypothesis)

Skewness: 0.0046
Kurtosis: -1.1972

==================================================
```

Normality Check for DTIRatio





```
Shapiro-Wilk Test Results for DTIRatio:
Statistic: 0.9572
p-value: 0.0000
The feature is not normally distributed (rejects null hypothesis)

Skewness: -0.0015
Kurtosis: -1.1997

==================================================
```

We can see here that the p-value for each column is zero, right away this tells us that our data is not normal in any numerical column. Similarly looking at the graphs, we see our histogram data does not match the bell-curve of normalised data, and similarly our data does not sit neatly on the trend lines in our Q-Q plot, all signs that we are not working with normalised data.

From here we can apply transofrmations to our data to make it more optimized for models that rely on normal data. Since we do not need our data to be normal in the analysis phase, we can simply continue with the data in it's current state, and later when choosing a model to train we can use one that does not rely on normalised data.

Analyzing Risk

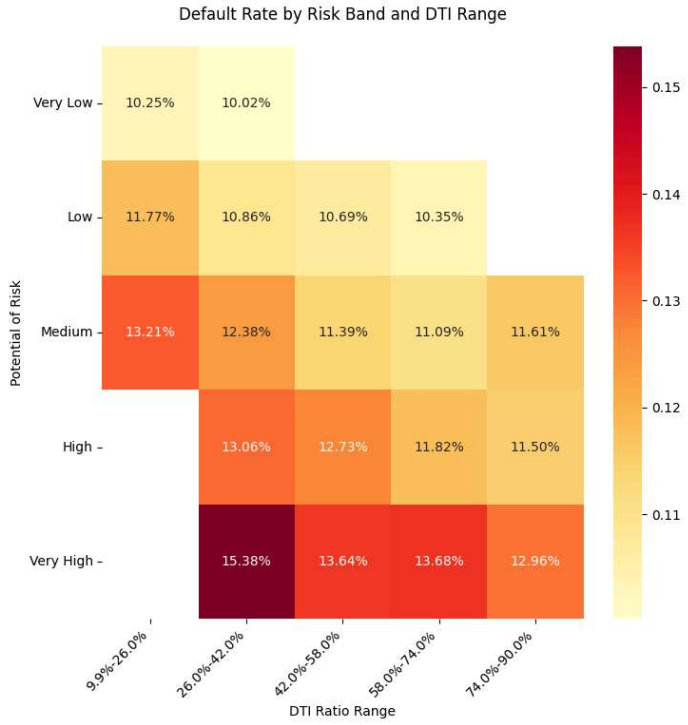
```
# Create new features
df['LoanToIncomeRatio'] = df['LoanAmount'] / df['Income']
df['RiskScore'] = (df['DTIRatio'] / df['CreditScore']) * 1000
df['MonthlyLoanBurden'] = (df['LoanAmount'] * (df['InterestRate']/1200)) / (1 - (1 + df['InterestRate']/1200)**(-df['LoanTerm']))
df['PaymentToIncome_Ratio'] = (df['MonthlyLoanBurden'] * 100) / (df['Income'] / 12)
```

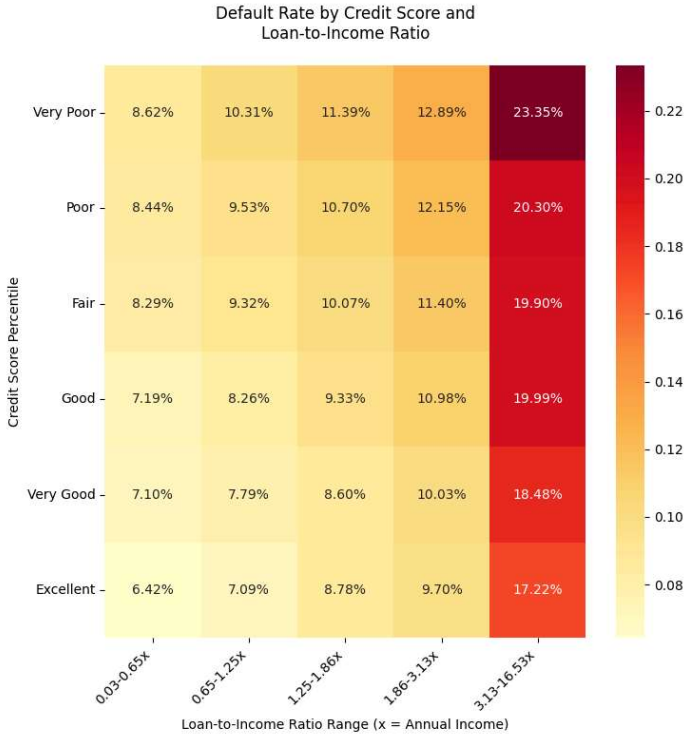
Risk Score and Loan-ToIncome Ratio

```
# Create bands for plot
df['RiskBand'] = pd.qcut(df['RiskScore'], q=5, labels=['Very Low', 'Low', 'Medium', 'High', 'Very High'])
df['CreditScoreBand'] = pd.cut(df['CreditScore'],
                               bins=[300, 400, 500, 600, 700, 800, 850],
                               labels=['Very Poor', 'Poor', 'Fair', 'Good', 'Very Good', 'Excellent'])

# Create DTI and LTI bins
dti_bins = pd.qcut(df['DTIRatio'], q=5)
df['LoanToIncomeRatio'] = df['LoanAmount'] / df['Income']
lti_bins = pd.qcut(df['LoanToIncomeRatio'], q=5)

print("\nDTI Ratio Ranges:")
print("-" * 50)
dti_ranges = [f"{interval.left:.1%}-{interval.right:.1%}"
              for interval in dti_bins.unique().categories]
for i, interval in enumerate(dti_bins.unique().categories, 1):
    print(f"Band {i}: {interval.left:.1%} to {interval.right:.1%}")

print("\nLoan-to-Income Ratio Ranges:")
print("-" * 50)
lti_ranges = [f"{interval.left:.2f}-{interval.right:.2f}x"
              for interval in lti_bins.unique().categories]
for i, interval in enumerate(lti_bins.unique().categories, 1):
    print(f"Band {i}: {interval.left:.2f}x to {interval.right:.2f}x annual income")

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 8))

# 1. Default Rate by Risk Band and DTI
risk_dti_default = df.groupby(['RiskBand', dti_bins], observed=False)['Default'].mean().unstack()
risk_dti_default.columns = dti_ranges

sns.heatmap(risk_dti_default, annot=True, fmt='.2%', cmap='YlOrRd', ax=ax1)
ax1.set_title('Default Rate by Risk Band and DTI Range', pad=20)
ax1.set_xlabel('DTI Ratio Range')
ax1.set_ylabel('Potential of Risk')
ax1.set_xticklabels(ax1.get_xticklabels(), rotation=45, ha='right')
ax1.set_yticklabels(ax1.get_yticklabels(), rotation=0)

# 2. Loan-to-Income Ratio vs Default Rate by Credit Score Band
lti_default = df.groupby(['CreditScoreBand', lti_bins], observed=False)['Default'].mean().unstack()
lti_default.columns = lti_ranges

sns.heatmap(lti_default, annot=True, fmt='.2%', cmap='YlOrRd', ax=ax2)
ax2.set_title('Default Rate by Credit Score and\nLoan-to-Income Ratio', pad=20)
ax2.set_xlabel('Loan-to-Income Ratio Range (x = Annual Income)')
ax2.set_ylabel('Credit Score Percentile')
ax2.set_xticklabels(ax2.get_xticklabels(), rotation=45, ha='right')
ax2.set_yticklabels(ax2.get_yticklabels(), rotation=0)

plt.tight_layout()
plt.show()

# Calculate Risk Band Statistics
```

```
risk_band_stats = df.groupby('RiskBand', observed=False)['Default'].agg([
    ('count', 'count'),
    ('default_rate', 'mean'),
    ('num_defaults', lambda x: x.sum())
]).round(4)

# Convert default rate to percentage
risk_band_stats['default_rate'] = (risk_band_stats['default_rate'] * 100).round(2)

# Add percentage of total loans column
risk_band_stats['percent_of_total'] = (risk_band_stats['count'] / risk_band_stats['count'].sum() * 100).round(2)

# Print Risk Band Statistics
print("\nRisk Band Statistics:")
print("-" * 50)
print(f"\nTotal number of loans: {df.shape[0]}")
print("\nBreakdown by Risk Band:")
print(risk_band_stats)
print("\nColumns:")
print("- count: Number of loans in this risk band")
print("- default_rate: Percentage of loans that defaulted in this band")
print("- num_defaults: Number of defaults in this band")
print("- percent_of_total: Percentage of total loans in this band")
```

```
DTI Ratio Ranges:
--------------------------------------------------
Band 1: 9.9% to 26.0%
Band 2: 26.0% to 42.0%
Band 3: 42.0% to 58.0%
Band 4: 58.0% to 74.0%
Band 5: 74.0% to 90.0%

Loan-to-Income Ratio Ranges:
--------------------------------------------------
Band 1: 0.03x to 0.65x annual income
Band 2: 0.65x to 1.25x annual income
Band 3: 1.25x to 1.86x annual income
Band 4: 1.86x to 3.13x annual income
Band 5: 3.13x to 16.53x annual income
```






```
Risk Band Statistics:
--------------------------------------------------

Total number of loans: 255347

Breakdown by Risk Band:
           count  default_rate  num_defaults  percent_of_total
RiskBand
Very Low   51075         10.21          5213              20.0
Low        51064         10.98          5607              20.0
Medium     51072         11.56          5905              20.0
High       51069         12.02          6140              20.0
Very High  51067         13.29          6788              20.0

Columns:
- count: Number of loans in this risk band
- default_rate: Percentage of loans that defaulted in this band
- num_defaults: Number of defaults in this band
- percent_of_total: Percentage of total loans in this band
```

Here we calculate the Potential of Risk as follows:

$$\text{Potenital} = \frac{DTIRatio}{CreditScore} \times 1000$$

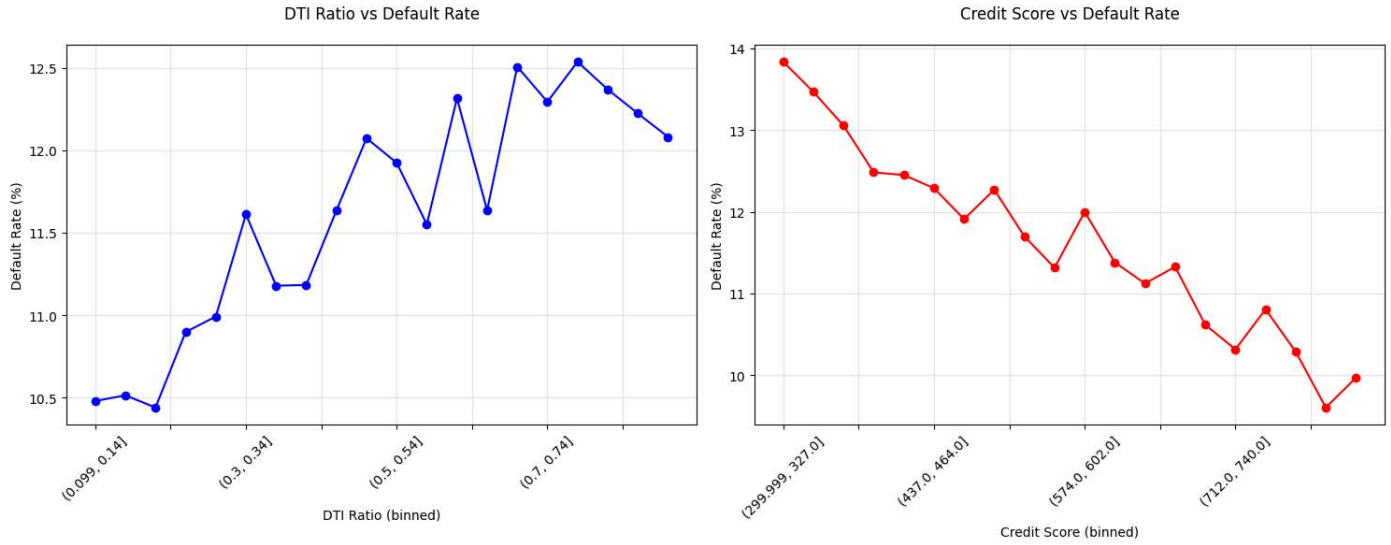
We see that:

- Just like we predicted, customers with a higher risk score are way more likely to default
- Although customers with a high DTI ratio (58% +) are usually more likely to default, the customers within the range of (26%-42%) show the highest default rates across all risk potentials
- This tells us that using DTI ratio to predict which customers will default may not be as predictable as other factors, like credit score and Loan-To-Income Ratio that show clear trends

    > - (Higher credit score and lower LTI ratio has less chance of defaulting and vice versa)

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Plot 1: DTI Ratio vs Default Rate
# Create 20 bins of DTI Ratio and calculate mean default rate for each bin
dti_bins = pd.qcut(df['DTIRatio'], q=20)
default_by_dti = df.groupby(dti_bins, observed=False)['Default'].mean() * 100  # Convert to percentage

# Plot the DTI ratio vs default rate
default_by_dti.plot(kind='line', marker='o', color='blue', ax=ax1)
ax1.set_title('DTI Ratio vs Default Rate', pad=20)
ax1.set_xlabel('DTI Ratio (binned)')
ax1.set_ylabel('Default Rate (%)')
ax1.tick_params(axis='x', rotation=45)
ax1.grid(True, alpha=0.3)

# Plot 2: Credit Score vs Default Rate
# Create 20 bins of Credit Score and calculate mean default rate for each bin
score_bins = pd.qcut(df['CreditScore'], q=20)
default_by_score = df.groupby(score_bins, observed=False)['Default'].mean() * 100  # Convert to percentage

# Plot the Credit Score vs default rate
default_by_score.plot(kind='line', marker='o', color='red', ax=ax2)
ax2.set_title('Credit Score vs Default Rate', pad=20)
ax2.set_xlabel('Credit Score (binned)')
ax2.set_ylabel('Default Rate (%)')
ax2.tick_params(axis='x', rotation=45)
ax2.grid(True, alpha=0.3)

# Adjust layout to prevent overlap
plt.tight_layout()

# Print some summary statistics
print("\nDefault Rate Statistics:")
print("-" * 50)
print("\nDTI Ratio Quartiles and Default Rates:")
dti_quartiles = pd.qcut(df['DTIRatio'], q=4)
dti_default_rates = df.groupby(dti_quartiles, observed=False)['Default'].agg(['mean', 'count'])
dti_default_rates['mean'] = dti_default_rates['mean'] * 100
print(dti_default_rates)

print("\nCredit Score Quartiles and Default Rates:")
score_quartiles = pd.qcut(df['CreditScore'], q=4)
score_default_rates = df.groupby(score_quartiles, observed=False)['Default'].agg(['mean', 'count'])
score_default_rates['mean'] = score_default_rates['mean'] * 100
print(score_default_rates)
```

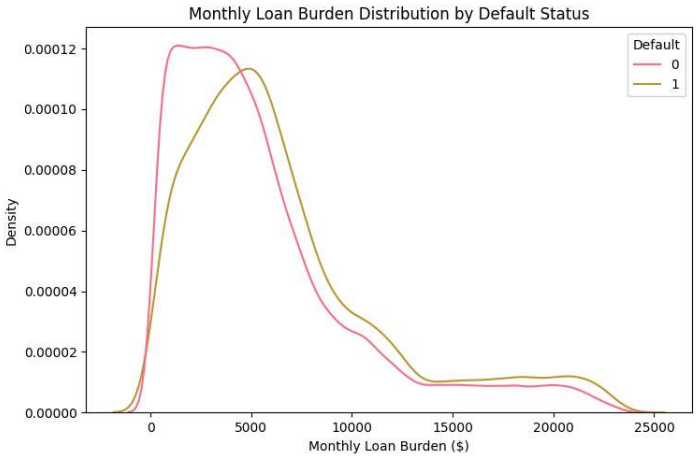
```
Default Rate Statistics:
--------------------------------------------------

DTI Ratio Quartiles and Default Rates:
                   mean  count
DTIRatio
(0.099, 0.3]  10.659331  65248
(0.3, 0.5]    11.535095  63970
(0.5, 0.7]    11.987896  63781
(0.7, 0.9]    12.306730  62348

Credit Score Quartiles and Default Rates:
                       mean  count
CreditScore
(299.999, 437.0]  13.055815  64140
(437.0, 574.0]    11.894425  63803
(574.0, 712.0]    11.290222  63896
(712.0, 849.0]    10.197141  63508
```

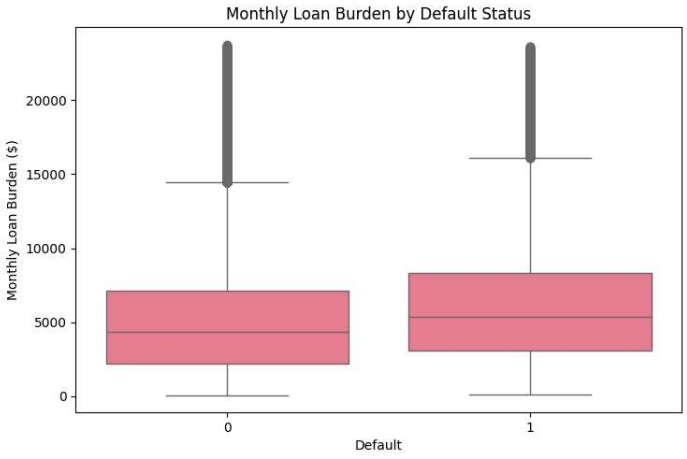


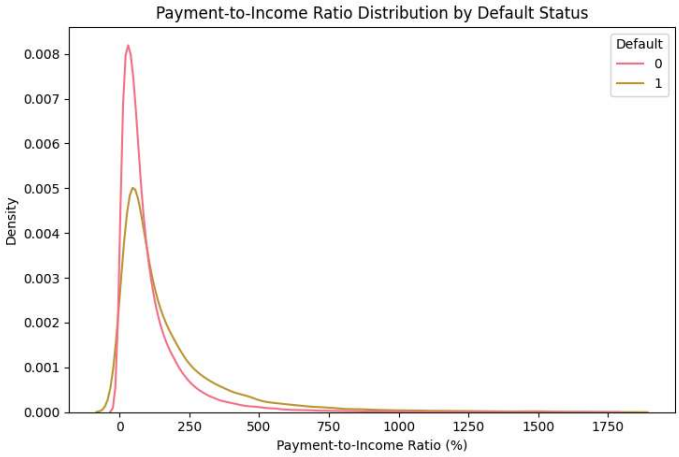
Plotting risk factors(DTI Ratio and Credit Score) by Default Rate we see clear trends indicating that a Higer DTI ratio and A lower credit score increases a customers chance of defaulting on their loan.

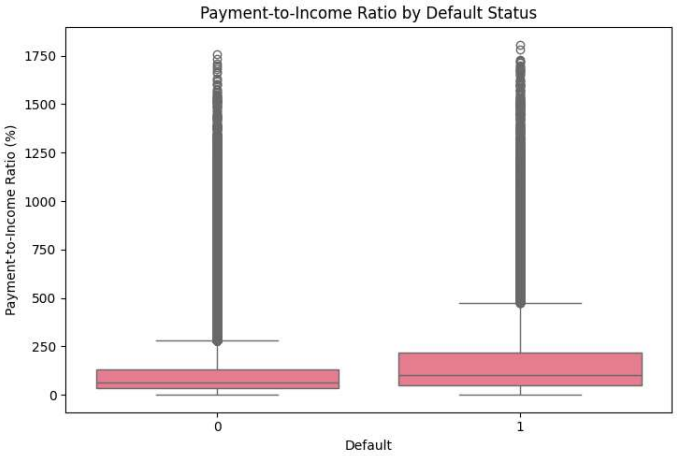
```
# Set up the plotting style
sns.set_palette("husl")

# Create figure with multiple subplots
fig = plt.figure(figsize=(15, 10))

# 1. Monthly Loan Burden Analysis
plt.subplot(2, 2, 1)
sns.kdeplot(data=df, x='MonthlyLoanBurden', hue='Default', common_norm=False)
plt.title('Monthly Loan Burden Distribution by Default Status')
plt.xlabel('Monthly Loan Burden ($)')
plt.ylabel('Density')

plt.subplot(2, 2, 2)
sns.boxplot(data=df, x='Default', y='MonthlyLoanBurden')
plt.title('Monthly Loan Burden by Default Status')
plt.ylabel('Monthly Loan Burden ($)')

# 2. Payment-to-Income Ratio Analysis
plt.subplot(2, 2, 3)
sns.kdeplot(data=df, x='PaymentToIncome_Ratio', hue='Default', common_norm=False)
plt.title('Payment-to-Income Ratio Distribution by Default Status')
plt.xlabel('Payment-to-Income Ratio (%)')
plt.ylabel('Density')

plt.subplot(2, 2, 4)
sns.boxplot(data=df, x='Default', y='PaymentToIncome_Ratio')
```

```
plt.title('Payment-to-Income Ratio by Default Status')
plt.ylabel('Payment-to-Income Ratio (%)')

plt.tight_layout()
plt.show()

# Create risk band analysis
plt.figure(figsize=(12, 5))

# Monthly Payment Risk Bands
payment_bins = df['MonthlyLoanBurden'].quantile([0, 0.25, 0.5, 0.75, 1.0])
df['Payment_RiskBand'] = pd.qcut(df['MonthlyLoanBurden'], q=4,
                                 labels=['Low', 'Medium', 'High', 'Very High'])

plt.subplot(1, 2, 1)
default_rates = df.groupby('Payment_RiskBand')['Default'].mean() * 100
sns.barplot(x=default_rates.index, y=default_rates.values)
plt.title('Default Rate by Monthly Payment Risk Band')
plt.ylabel('Default Rate (%)')
plt.xticks(rotation=45)

# Payment-to-Income Risk Bands
pti_bins = [0, 20, 30, 40, float('inf')]
pti_labels = ['Low', 'Medium', 'High', 'Very High']
df['PTI_RiskBand'] = pd.cut(df['PaymentToIncome_Ratio'], bins=pti_bins, labels=pti_labels)

plt.subplot(1, 2, 2)
default_rates = df.groupby('PTI_RiskBand')['Default'].mean() * 100
sns.barplot(x=default_rates.index, y=default_rates.values)
plt.title('Default Rate by Payment-to-Income Risk Band')
plt.ylabel('Default Rate (%)')
plt.xticks(rotation=45)

plt.tight_layout()
plt.show()

# Create correlation heatmap
correlation_features = ['MonthlyLoanBurden', 'PaymentToIncome_Ratio',
                        'LoanAmount', 'Income', 'CreditScore', 'InterestRate', 'Default']
correlation_matrix = df[correlation_features].corr()

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Heatmap of Key Features')
plt.tight_layout()
plt.show()

# Print summary statistics
print("\nMonthly Loan Burden Statistics:")
print("-" * 50)
print(df['MonthlyLoanBurden'].describe().round(2))

print("\nPayment-to-Income Ratio Statistics:")
print("-" * 50)
print(df['PaymentToIncome_Ratio'].describe().round(2))

# Print default rates by risk bands
print("\nDefault Rates by Payment Risk Band:")
print("-" * 50)
print(df.groupby('Payment_RiskBand')['Default'].agg(['count', 'mean']).round(4))

print("\nDefault Rates by Payment-to-Income Risk Band:")
print("-" * 50)
print(df.groupby('PTI_RiskBand')['Default'].agg(['count', 'mean']).round(4))
```

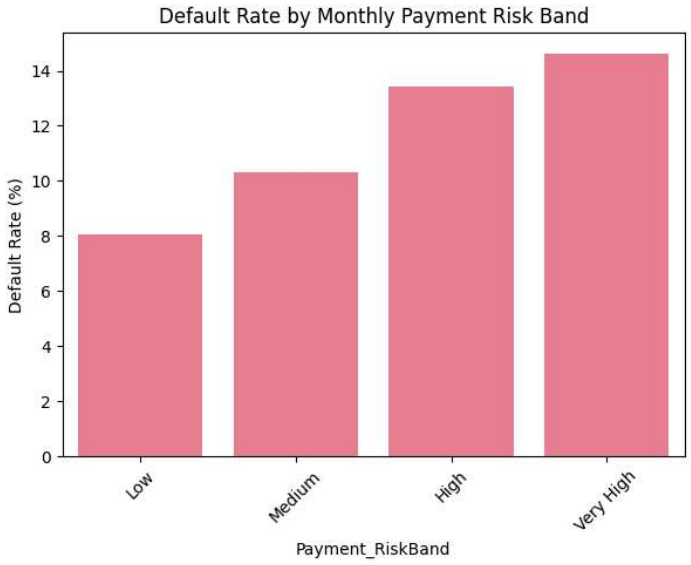


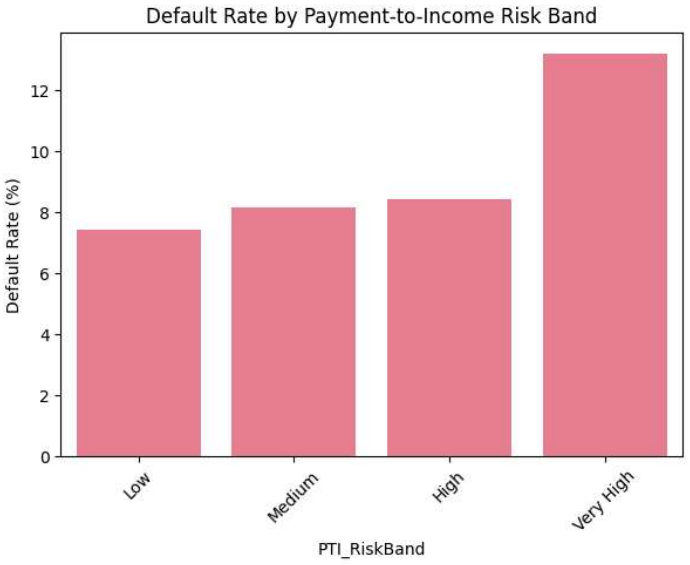


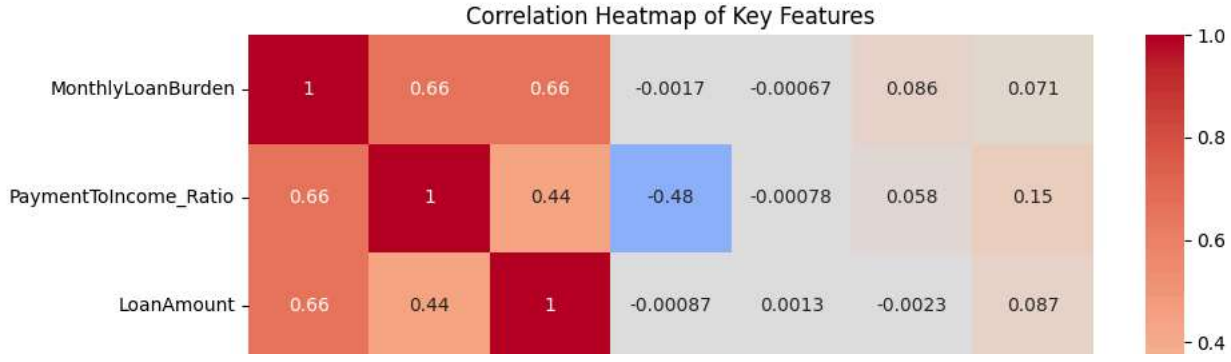


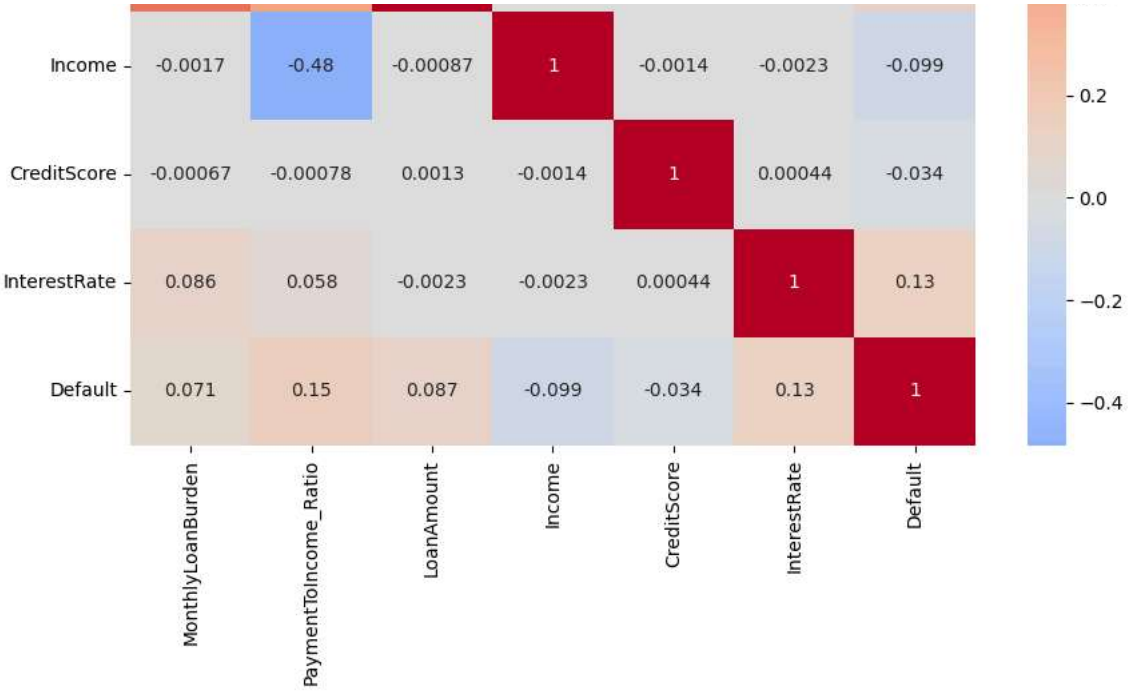



```
<ipython-input-75-60e55da54c11>:43: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future v
  default_rates = df.groupby('Payment_RiskBand')['Default'].mean() * 100
<ipython-input-75-60e55da54c11>:55: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future v
  default_rates = df.groupby('PTI_RiskBand')['Default'].mean() * 100
```










```
Monthly Loan Burden Statistics:
--------------------------------------------------
count    255347.00
mean       5649.06
std        4731.40
min          90.80
25%        2317.46
50%        4442.06
75%        7259.85
max       23735.67
Name: MonthlyLoanBurden, dtype: float64

Payment-to-Income Ratio Statistics:
--------------------------------------------------
count    255347.00
mean        115.73
std         146.64
min           0.82
25%          33.61
50%          68.25
75%         139.79
max        1808.61
Name: PaymentToIncome_Ratio, dtype: float64

Default Rates by Payment Risk Band:
--------------------------------------------------
                  count    mean
Payment_RiskBand
Low               63837  0.0806
Medium            63837  0.1032
High              63836  0.1343
Very High         63837  0.1464

Default Rates by Payment-to-Income Risk Band:
--------------------------------------------------
               count    mean
PTI_RiskBand
Low            35993  0.0742
Medium         20325  0.0817
High           20750  0.0844
Very High     178279  0.1322
<ipython-input-75-60e55da54c11>:87: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future v
  print(df.groupby('Payment_RiskBand')['Default'].agg(['count', 'mean']).round(4))
<ipython-input-75-60e55da54c11>:91: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future v
  print(df.groupby('PTI_RiskBand')['Default'].agg(['count', 'mean']).round(4))
```

We calculate Monthly Loan Burden as follows:

$$\text{MonthlyLoanBurden} = \frac{\text{LoanAmount} \times (\frac{\text{InterestRate}}{1200})}{1 - (1 + \frac{\text{InterestRate}}{1200})^{-\text{LoanTerm}}}$$

> - Using the stadard amortization formula that ensures whether a loan is fully paid off by the end of its term or not based off of the loan amount, interest rate (converted to monthly), and loan term
> - However looking at our kde plot, we can only gather that most customers have a monthly loan burden of ~$5000, but this is true for both defaulted customers and those who paid off thier loan

We calculate Payment to Income Ratio as:

$$\text{PaymentToIncome\_Ratio} = \frac{\text{MonthlyLoanBurden} \times 100}{\frac{\text{Income}}{12}}$$

> - Here we express monthly loan payment as a percentage of monthly income
> - The higher the ratio is the more potential for risk
> - Again our kde plot dosent show much difference between defaulted customers and paid off customers

ML Model Evaluation

- Here we will be testing the performance of various types of ML Models and choosing the best model for our data

Pre Processing

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.impute import SimpleImputer

def handle_missing_values(df):
    """Impute missing values appropriately"""
    data = df.copy()

    # Separate numerical and categorical columns
    numeric_columns = data.select_dtypes(include=['int64', 'float64']).columns
    categorical_columns = data.select_dtypes(include=['object']).columns

    # Impute numerical columns with median
    numeric_imputer = SimpleImputer(strategy='median')
    data[numeric_columns] = numeric_imputer.fit_transform(data[numeric_columns])

    # Impute categorical columns with mode
    categorical_imputer = SimpleImputer(strategy='most_frequent')
    data[categorical_columns] = categorical_imputer.fit_transform(data[categorical_columns])

    if data.isnull().values.any():
        print("WARNING: There are still NaN values in the dataset after imputation!")
        print("Investigate and fix the source of these missing values.")
        print(data.isnull().sum()) # Print the columns with NaN values for debugging

    return data

    return data

def handle_outliers(df):
    """Handle outliers in numerical columns"""
    data = df.copy()

    # List of columns to check for outliers
    columns_to_check = ['Age', 'Income', 'LoanAmount', 'MonthsEmployed',
                        'NumCreditLines', 'InterestRate', 'DTIRatio',
                        'MonthlyLoanBurden', 'PaymentToIncome_Ratio',]

    for column in columns_to_check:
        Q1 = data[column].quantile(0.25)
        Q3 = data[column].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
```

```
        upper_bound = Q3 + 1.5 * IQR
        data[column] = data[column].clip(lower=lower_bound, upper=upper_bound)

    return data

def encode_categorical_variables(df):
    """Encode categorical variables appropriately"""

    data = df.copy()

    if not isinstance(df, pd.DataFrame):
      raise TypeError("Input must be a pandas DataFrame")

    # Binary encoding for yes/no variables
    binary_columns = ['HasMortgage', 'HasDependents', 'HasCoSigner']

    # Binary encoding
    if binary_columns:
        for col in binary_columns:
            data[col] = data[col].map({'Yes': 1, 'No': 0})

    # Direct mapping is the only strategy that works here

    data['Education'] = data['Education'].map({"High School": 1,
                                           "Bachelor's": 2,
                                           "Master's": 3,
                                           "PhD": 4})

    data['EmploymentType'] = data['EmploymentType'].map({"Unemployed": 1,
                                                 "Self-employed": 2,
                                                 "Part-time": 3,
                                                 "Full-time": 4})

    data['MaritalStatus'] = data['MaritalStatus'].map({"Single": 1,
                                                "Married": 2,
                                                "Divorced": 3})

    data['LoanPurpose'] = data['LoanPurpose'].map({"Auto": 1,
                                              "Business": 2,
                                              "Education": 3,
                                              "Home": 4,
                                              "Other": 5})

    return data


# Step 5: Feature scaling
def scale_features(df):
    """Scale numerical features"""
    data = df.copy()

    # Identify numerical columns to scale (exclude target variable and ID)
    columns_to_scale = data.select_dtypes(include=['int64', 'float64']).columns
    columns_to_scale = columns_to_scale.drop('Default')  # Exclude target variable
    if 'LoanID' in columns_to_scale:
        columns_to_scale = columns_to_scale.drop('LoanID')  # Exclude ID

    # Scale the features
    scaler = StandardScaler()
    data[columns_to_scale] = scaler.fit_transform(data[columns_to_scale])

    return data

# Main preprocessing pipeline
def preprocess_data(df, test_size=0.2, random_state=42):
    """Complete preprocessing pipeline"""

    print("Starting preprocessing pipeline...")

    print("Handling missing values...")
    df= handle_missing_values(df)

    print("Handling outliers...")
    df = handle_outliers(df)

    print("Encoding categorical variables...")
    df = encode_categorical_variables(df)
```

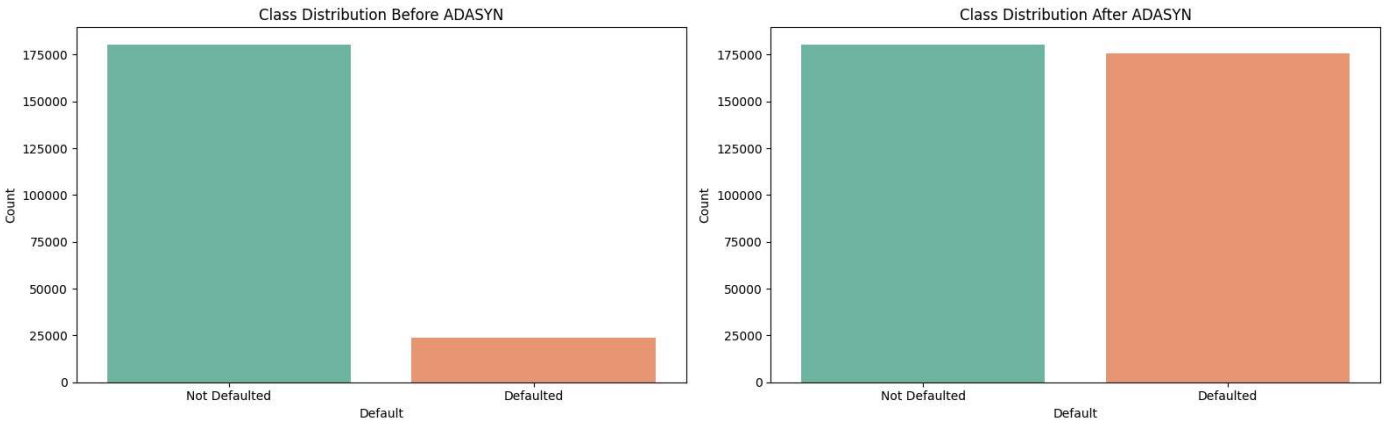
```
    print("Scaling features...")
    df = scale_features(df)

    print("Splitting the data...")

    # Remove LoanID if present
    if 'LoanID' in df.columns:
        df = df.drop('LoanID', axis=1)

    # Remove Band columns
    if 'RiskBand' in df.columns:
        df = df.drop('RiskBand', axis=1)
    if 'CreditScoreBand' in df.columns:
        df = df.drop('CreditScoreBand', axis=1)
    if 'Payment_RiskBand' in df.columns:
        df = df.drop('Payment_RiskBand', axis=1)
    if 'PTI_RiskBand' in df.columns:
        df = df.drop('PTI_RiskBand', axis=1)

    # Split features and target
    X = df.drop('Default', axis=1)
    y = df['Default']

    if X.isnull().values.any():
        print("ERROR: NaN values present in features after preprocessing.")
        print("Investigate and fix the source of these missing values.")
        print(X.isnull().sum()) # Print the columns with NaN values for debugging
        # You can choose to either impute the remaining NaNs here
        # or raise an exception to stop the process.

    # Create train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                        test_size=test_size,
                                                        random_state=random_state,
                                                        stratify=y)

    print("Preprocessing complete!")
    return X_train, X_test, y_train, y_test

# Execute preprocessing
X_train, X_test, y_train, y_test = preprocess_data(df)

# Print shapes of resulting datasets
print("\nFinal dataset shapes:")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

# Print feature names
print("\nFinal features:")
print(X_train.columns.tolist())
```

```
Starting preprocessing pipeline...
Handling missing values...
WARNING: There are still NaN values in the dataset after imputation!
Investigate and fix the source of these missing values.
LoanID                     0
Age                        0
Income                     0
LoanAmount                 0
CreditScore                0
MonthsEmployed             0
NumCreditLines             0
InterestRate               0
LoanTerm                   0
DTIRatio                   0
Education                  0
EmploymentType             0
MaritalStatus              0
HasMortgage                0
HasDependents              0
LoanPurpose                0
HasCoSigner                0
Default                    0
LoanToIncomeRatio          0
RiskScore                  0
MonthlyLoanBurden          0
```

```
PaymentToIncome_Ratio      0
RiskBand                   0
CreditScoreBand          484
Payment_RiskBand           0
PTI_RiskBand               0
dtype: int64
Handling outliers...
Encoding categorical variables...
Scaling features...
Splitting the data...
Preprocessing complete!

Final dataset shapes:
X_train shape: (204277, 20)
X_test shape: (51070, 20)
y_train shape: (204277,)
y_test shape: (51070,)

Final features:
['Age', 'Income', 'LoanAmount', 'CreditScore', 'MonthsEmployed', 'NumCreditLines', 'InterestRate', 'LoanTerm', 'DTIRatio', 'Education',
```

```python
from imblearn.over_sampling import ADASYN

# Creating subplots for class distribution before and after ADASYN
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

# Plotting class distribution before ADASYN
sns.countplot(x='Default', data=pd.concat([X_train, y_train], axis=1), palette='Set2', ax=axes[0])
axes[0].set_title('Class Distribution Before ADASYN')
axes[0].set_xlabel('Default')
axes[0].set_ylabel('Count')
axes[0].set_xticks([0, 1])
axes[0].set_xticklabels(['Not Defaulted', 'Defaulted'])

# Applying ADASYN to balance the classes
adasyn = ADASYN(sampling_strategy='minority', random_state=42)
X_resampled, y_resampled = adasyn.fit_resample(X_train, y_train)

# Plotting class distribution after ADASYN
sns.countplot(x=y_resampled, palette='Set2', ax=axes[1])
axes[1].set_title('Class Distribution After ADASYN')
axes[1].set_xlabel('Default')
axes[1].set_ylabel('Count')
axes[1].set_xticks([0, 1])
axes[1].set_xticklabels(['Not Defaulted', 'Defaulted'])

plt.tight_layout()
plt.show()

# Creating a DataFrame to compare before and after
before_count = pd.Series(y_train).value_counts()
after_count = pd.Series(y_resampled).value_counts()

# Creating a DataFrame for better visualization
comparison_df = pd.DataFrame({
    'Before': before_count,
    'After': after_count
})

comparison_df
```

```
<ipython-input-77-3a8e976050c2>:7: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend

  sns.countplot(x='Default', data=pd.concat([X_train, y_train], axis=1), palette='Set2', ax=axes[0])
<ipython-input-77-3a8e976050c2>:19: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend

  sns.countplot(x=y_resampled, palette='Set2', ax=axes[1])
```



Default	Before	After
0.0	180555	180555
1.0	23722	175618

Next steps: Generate code with comparison_df | View recommended plots | New interactive sheet

Model Comparison

We will be evaluating our models on these metrics:

- Accuracy: Percentage of all correct predictions (both defaults and non-defaults)
  - ex. 0.88 means 88% of all predictions were correct
  - Can be misleading with imbalanced data
- Precision: Of customers predicted to default, what percentage actually defaulted
  - 0.63 means 63% of predicted defaults were actual defaults
  - Important when false positives are costly (wrongly denying loans)
- Recall: Of customers who actually defaulted, what percentage were caught
  - 0.69 means 69% of actual defaults were identified
  - Critical when missing defaults is costly (losing money on unpaid loans)
- F1 Score: Harmonic mean of precision and recall
  - Balances precision-recall tradeoff
  - Higher is better, 1.0 is perfect
- ROC AUC: Model's ability to distinguish between classes
  - 0.5 is random guessing
  - 1.0 is perfect separation
  - 0.75 indicates moderate discriminative ability

```
!pip install --upgrade scikit-learn xgboost
!pip install --upgrade scikit-learn
!pip install catboost
```

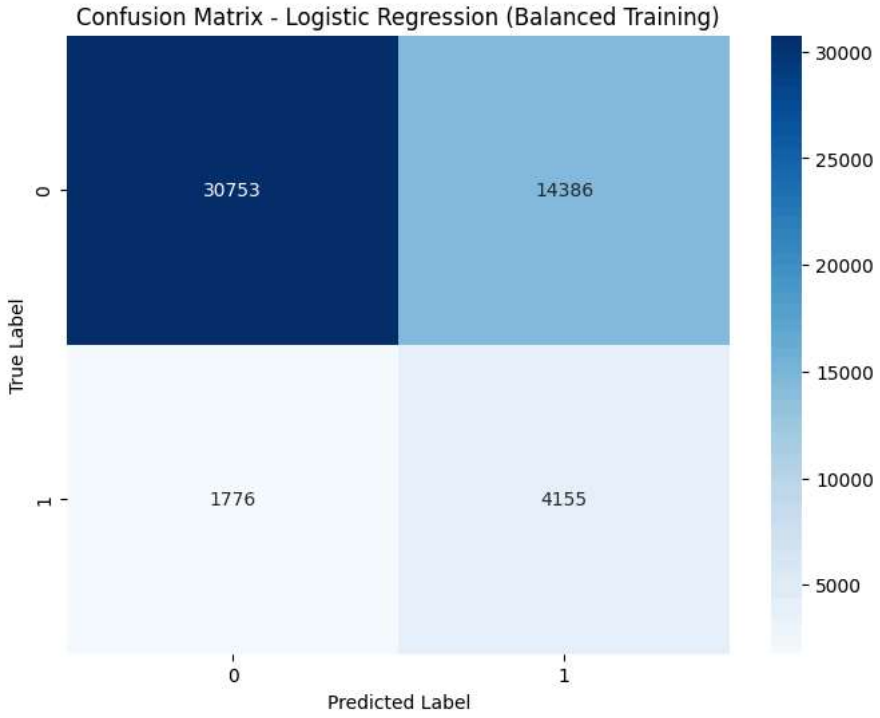
```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (1.6.1)
Requirement already satisfied: xgboost in /usr/local/lib/python3.11/dist-packages (2.1.4)
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.11/dist-packages (from xgboost) (2.21.5)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (1.6.1)
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: catboost in /usr/local/lib/python3.11/dist-packages (1.2.7)
Requirement already satisfied: graphviz in /usr/local/lib/python3.11/dist-packages (from catboost) (0.20.3)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (from catboost) (3.10.0)
Requirement already satisfied: numpy<2.0,>=1.16.0 in /usr/local/lib/python3.11/dist-packages (from catboost) (1.26.4)
Requirement already satisfied: pandas>=0.24 in /usr/local/lib/python3.11/dist-packages (from catboost) (2.2.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from catboost) (1.13.1)
Requirement already satisfied: plotly in /usr/local/lib/python3.11/dist-packages (from catboost) (5.24.1)
Requirement already satisfied: six in /usr/local/lib/python3.11/dist-packages (from catboost) (1.17.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.24->catboost) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.24->catboost) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.24->catboost) (2025.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->catboost) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib->catboost) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib->catboost) (4.55.8)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->catboost) (1.4.8)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib->catboost) (24.2)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib->catboost) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->catboost) (3.2.1)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.11/dist-packages (from plotly->catboost) (9.0.0)
```

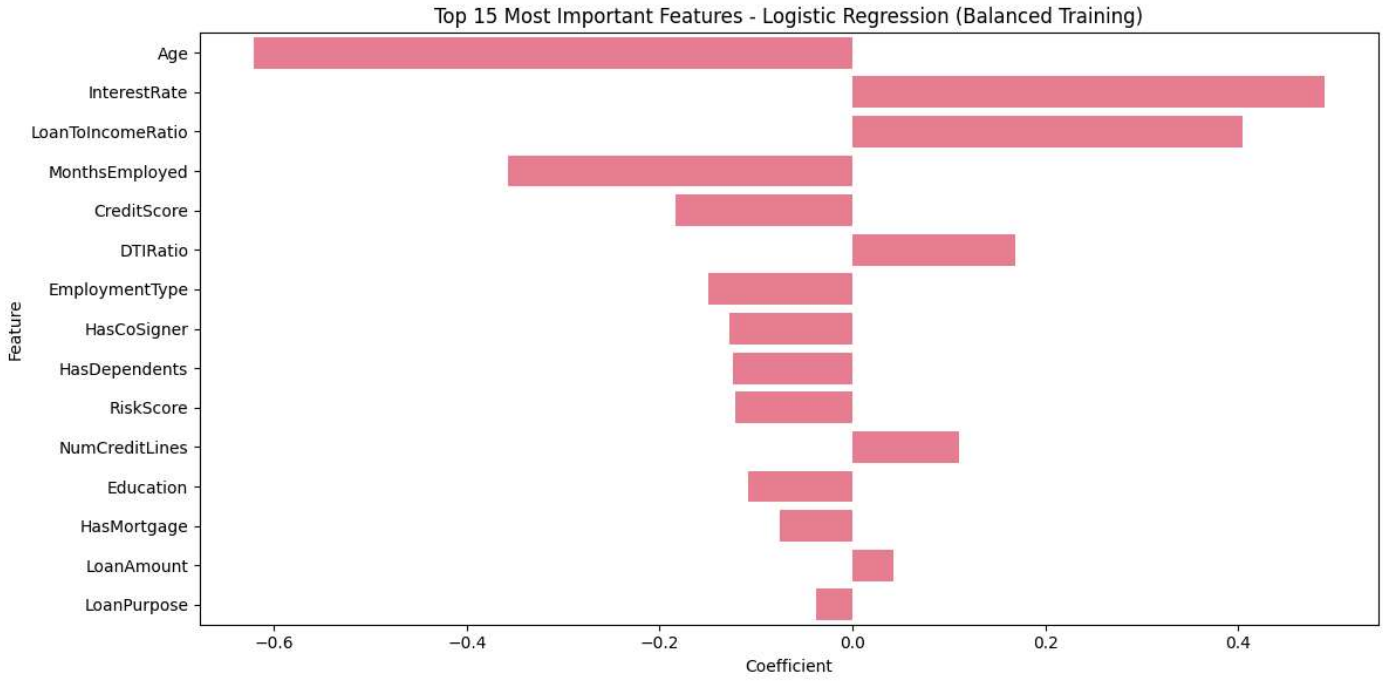
Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import roc_auc_score, confusion_matrix, classification_report
from sklearn.model_selection import cross_val_score

def evaluate_logistic_regression_balanced(X_resampled, X_test, y_resampled, y_test):
    """
    Evaluate Logistic Regression model using balanced (resampled) data for training
    """
    # Initialize model
    lr_model = LogisticRegression(
        max_iter=1000,        # Increase max iterations for convergence
        random_state=42,      # For reproducibility
        # Note: Removed class_weight='balanced' since we're using resampled data
    )

    # Train on resampled data
    print("Training Logistic Regression model on balanced data...")
    lr_model.fit(X_resampled, y_resampled)

    # Make predictions on test set
    y_pred = lr_model.predict(X_test)
    y_pred_proba = lr_model.predict_proba(X_test)[:, 1]

    # Calculate metrics
    logistic_metrics = {
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1 Score': f1_score(y_test, y_pred),
        'ROC AUC': roc_auc_score(y_test, y_pred_proba)
    }

    # Perform cross-validation on resampled data
    cv_scores = cross_val_score(lr_model, X_resampled, y_resampled, cv=5, scoring='roc_auc')

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
```

```
    plt.title('Confusion Matrix - Logistic Regression (Balanced Training)')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

    # Calculate and plot feature importance
    feature_importance = pd.DataFrame({
        'Feature': X_resampled.columns,
        'Coefficient': lr_model.coef_[0]
    })
    feature_importance['Abs_Coefficient'] = abs(feature_importance['Coefficient'])
    feature_importance = feature_importance.sort_values('Abs_Coefficient', ascending=False)

    plt.figure(figsize=(12, 6))
    sns.barplot(x='Coefficient', y='Feature', data=feature_importance.head(15))
    plt.title('Top 15 Most Important Features - Logistic Regression (Balanced Training)')
    plt.tight_layout()
    plt.show()

    # Print classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Print model performance
    print("\nLogistic Regression Model Performance (trained on balanced data):")
    for metric, value in logistic_metrics.items():
        print(f"{metric}: {value:.4f}")
    print(f"\nCross-validation ROC AUC scores: {cv_scores}")
    print(f"Mean CV ROC AUC: {cv_scores.mean():.4f} (+/- {cv_scores.std() * 2:.4f})")

    return lr_model, logistic_metrics, feature_importance

model, logistic_metrics, feature_importance = evaluate_logistic_regression_balanced(X_resampled, X_test, y_resampled, y_test)
```

Training Logistic Regression model on balanced data...





Classification Report:

	precision	recall	f1-score	support
0.0	0.95	0.68	0.79	45139
1.0	0.22	0.70	0.34	5931
accuracy			0.68	51070
macro avg	0.58	0.69	0.57	51070
weighted avg	0.86	0.68	0.74	51070

Logistic Regression Model Performance (trained on balanced data):
Accuracy: 0.6835
Precision: 0.2241
Recall: 0.7006
F1 Score: 0.3396
ROC AUC: 0.7586

Cross-validation ROC AUC scores: [0.75022605 0.74843798 0.74095518 0.74553514 0.74703596]
Mean CV ROC AUC: 0.7464 (+/- 0.0063)

Results:

- **For non-defaults (0.0):**

  - Precision: 0.95 (very good at identifying non-defaults)
  - Recall: 0.68 (captures about 68% of actual non-defaults)
  - F1-score: 0.79 (good balance) between precision and recall

- **For defaults (1.0):**

  - Precision: 0.22 (many false positives)
  - Recall: 0.70 (captures 70% of actual defaults)
  - F1-score: 0.34 (relatively low due to poor precision)

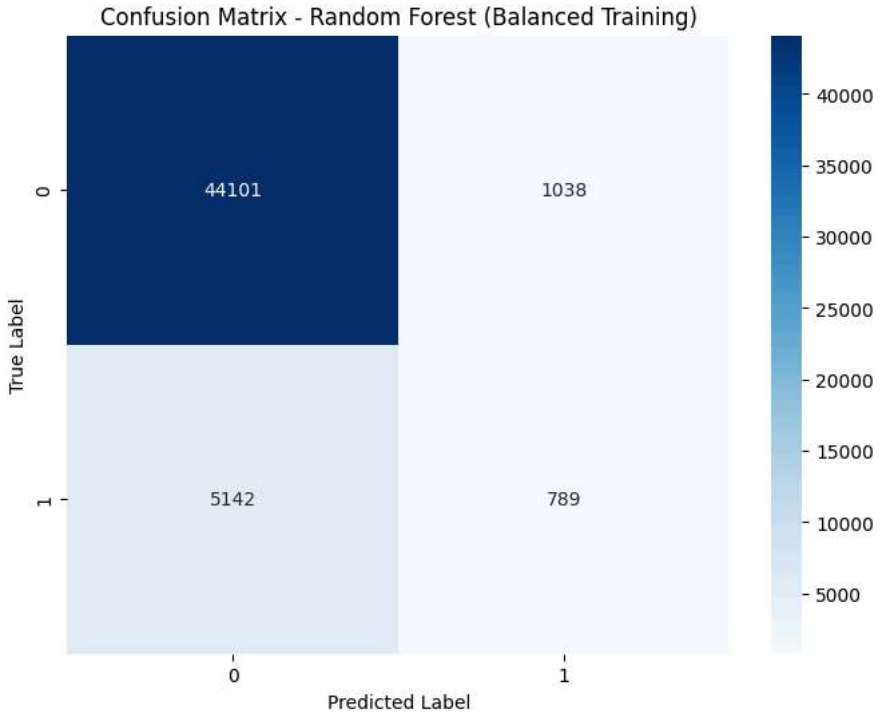
- **Most Important Feature:** Age
- **Overall:** Linear Regression is 68% accurate, But it has a high false positive rate (78% of predicted defaults are wrong).

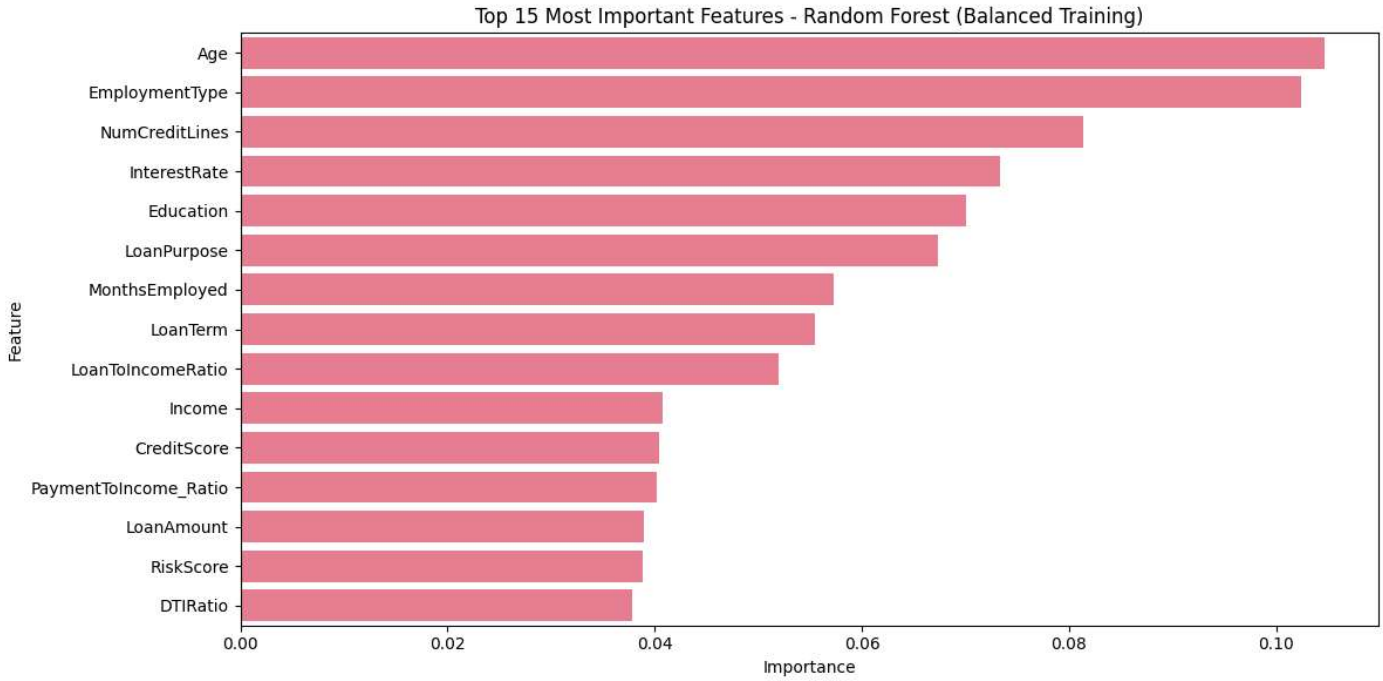
Random Forest

```
from sklearn.ensemble import RandomForestClassifier

def evaluate_random_forest_balanced(X_resampled, X_test, y_resampled, y_test):
    """
    Evaluate Random Forest model using balanced (resampled) data for training
    """
    # Initialize model
    rf_model = RandomForestClassifier(
        n_estimators=100,     # Number of trees
        max_depth=None,       # Maximum depth of trees
        min_samples_split=2,  # Minimum samples required to split
        min_samples_leaf=1,   # Minimum samples required at leaf node
        random_state=42,      # For reproducibility
        # Removed class_weight since we're using resampled data
    )

    # Train on resampled data
    print("Training Random Forest model on balanced data...")
    rf_model.fit(X_resampled, y_resampled)

    # Make predictions on test set
    y_pred = rf_model.predict(X_test)
    y_pred_proba = rf_model.predict_proba(X_test)[:, 1]

    # Calculate metrics
    rf_metrics = {
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1 Score': f1_score(y_test, y_pred),
        'ROC AUC': roc_auc_score(y_test, y_pred_proba)
    }

    # Perform cross-validation on resampled data
    cv_scores = cross_val_score(rf_model, X_resampled, y_resampled, cv=5, scoring='roc_auc')

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix - Random Forest (Balanced Training)')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

    # Calculate and plot feature importance
    feature_importance = pd.DataFrame({
        'Feature': X_resampled.columns,
        'Importance': rf_model.feature_importances_
```

```
    })
    feature_importance = feature_importance.sort_values('Importance', ascending=False)

    plt.figure(figsize=(12, 6))
    sns.barplot(x='Importance', y='Feature', data=feature_importance.head(15))
    plt.title('Top 15 Most Important Features - Random Forest (Balanced Training)')
    plt.tight_layout()
    plt.show()

    # Print classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Print model performance
    print("\nRandom Forest Model Performance (trained on balanced data):")
    for metric, value in rf_metrics.items():
        print(f"{metric}: {value:.4f}")
    print(f"\nCross-validation ROC AUC scores: {cv_scores}")
    print(f"Mean CV ROC AUC: {cv_scores.mean():.4f} (+/- {cv_scores.std() * 2:.4f})")

    return rf_model, rf_metrics, feature_importance

model, rf_metrics, feature_importance = evaluate_random_forest_balanced(X_resampled, X_test, y_resampled, y_test)
```

Training Random Forest model on balanced data...





Classification Report:

	precision	recall	f1-score	support
0.0	0.90	0.98	0.93	45139
1.0	0.43	0.13	0.20	5931
accuracy			0.88	51070
macro avg	0.66	0.56	0.57	51070
weighted avg	0.84	0.88	0.85	51070

Random Forest Model Performance (trained on balanced data):
Accuracy: 0.8790
Precision: 0.4319
Recall: 0.1330
F1 Score: 0.2034
ROC AUC: 0.7329

Cross-validation ROC AUC scores: [0.9461085 0.99616067 0.99562667 0.99617659 0.99595233]
Mean CV ROC AUC: 0.9860 (+/- 0.0399)

- **Non-defaults (0.0):**

  - Precision: 0.90 (very good at identifying non-defaults)
  - Recall: 0.98 (almost perfect at catching non-defaults)
  - F1-score: 0.93 (excellent balance)

- **Defaults (1.0):**

  - Precision: 0.43 (43% of predicted defaults are actual defaults - much better than Logistic Regression's 23%)
  - Recall: 0.13 (only catches 13% of actual defaults - much worse than Logistic Regression's 69%)
  - F1-score: 0.20 (very low due to poor recall)

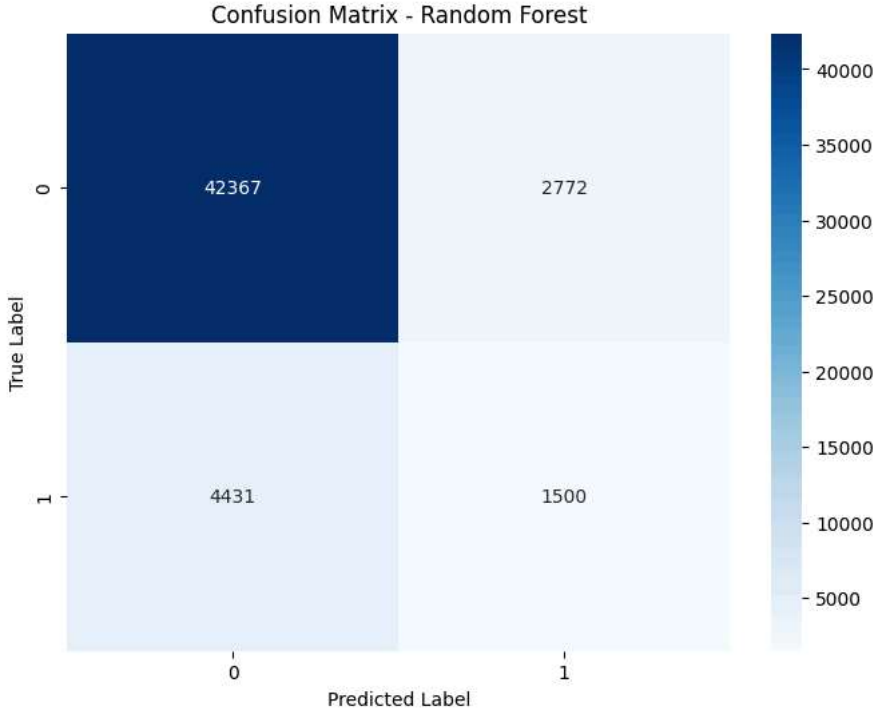
- **Most Important Feature:** Age
- **Overall:** this model is 88% acurate, but is way more effective at catching non-defaults than it is at catching defaults. The difference between the training ROC AUTH (0.73) and CV ROC AUTH (0.98) Suggest overfitting

Hyperparameter Tuning for Random Forest

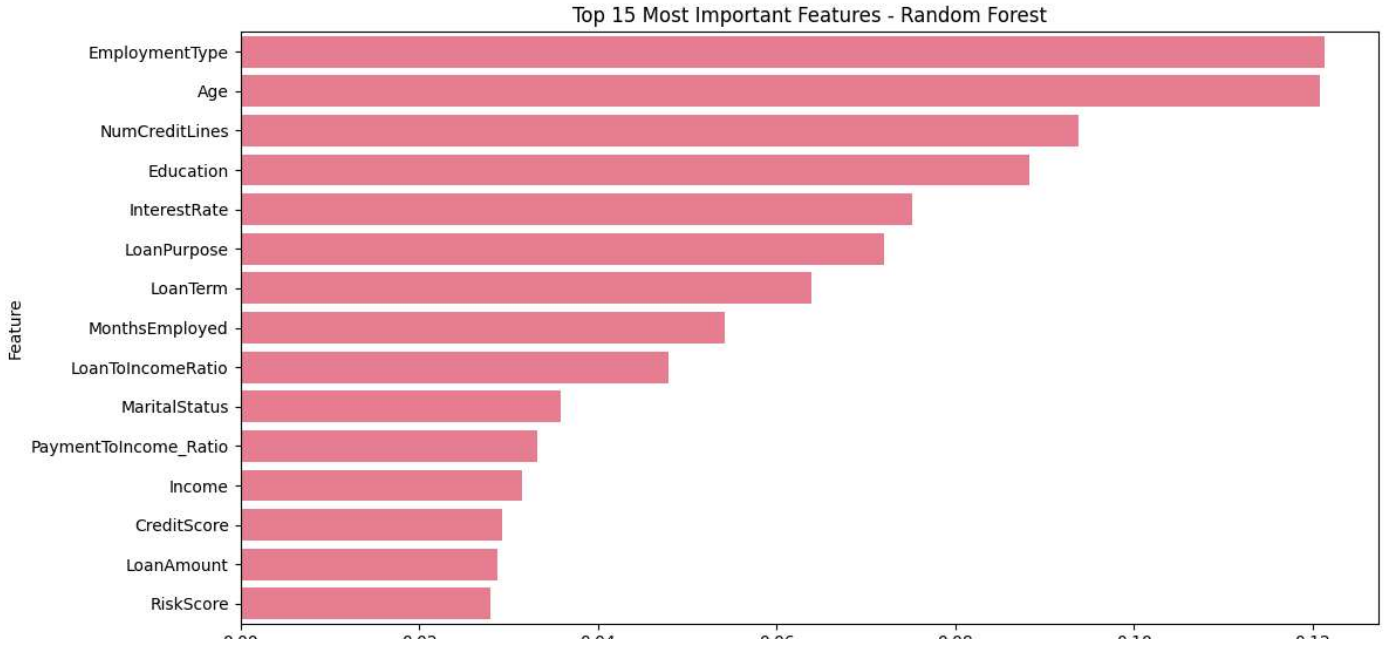
```
from sklearn.model_selection import RandomizedSearchCV

def evaluate_random_forest(X_resampled, X_test, y_resampled, y_test, tune_params=True):
    """
    Efficient Random Forest evaluation with optional quick parameter tuning
    """
    if tune_params:
        # Streamlined parameter grid
        param_dist = {
            'n_estimators': [100, 200],
            'max_depth': [10, 15, 20],
            'min_samples_split': [5, 10],
            'min_samples_leaf': [2, 4]
        }

        # Initialize base model with efficient settings
        rf_base = RandomForestClassifier(
            random_state=42,
            n_jobs=-1,           # Use all cores
            max_features='sqrt'  # Default efficient setting
        )

        rf_random = RandomizedSearchCV(
            estimator=rf_base,
            param_distributions=param_dist,
            n_iter=5,
            cv=3,
            random_state=42,
            scoring='roc_auc',
            n_jobs=-1
        )

        print("Performing hyperparameter tuning...")
        rf_random.fit(X_resampled, y_resampled)
        rf_model = rf_random.best_estimator_

        print("\nBest parameters found:")
        for param, value in rf_random.best_params_.items():
            print(f"{param}: {value}")

    else:
        # Manual efficient parameters
        rf_model = RandomForestClassifier(
            n_estimators=200,
            max_depth=15,
            min_samples_split=5,
            min_samples_leaf=2,
            max_features='sqrt',
            n_jobs=-1,
            random state=42
```

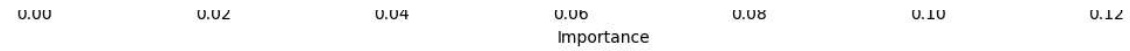
```
                    _
        )
        print("Training Random Forest with preset parameters...")
        rf_model.fit(X_resampled, y_resampled)

    # Make predictions
    y_pred = rf_model.predict(X_test)
    y_pred_proba = rf_model.predict_proba(X_test)[:, 1]

    # Calculate metrics
    rf_metrics = {
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1 Score': f1_score(y_test, y_pred),
        'ROC AUC': roc_auc_score(y_test, y_pred_proba)
    }

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix - Random Forest')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

    # Feature importance
    feature_importance = pd.DataFrame({
        'Feature': X_resampled.columns,
        'Importance': rf_model.feature_importances_
    })
    feature_importance = feature_importance.sort_values('Importance', ascending=False)

    plt.figure(figsize=(12, 6))
    sns.barplot(x='Importance', y='Feature', data=feature_importance.head(15))
    plt.title('Top 15 Most Important Features - Random Forest')
    plt.tight_layout()
    plt.show()

    # Print classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Print model performance
    print("\nRandom Forest Model Performance:")
    for metric, value in rf_metrics.items():
        print(f"{metric}: {value:.4f}")

    return rf_model, rf_metrics, feature_importance

model, rf_metrics, feature_importance = evaluate_random_forest(X_resampled, X_test, y_resampled, y_test)
```

```
Performing hyperparameter tuning...

Best parameters found:
n_estimators: 100
min_samples_split: 5
min_samples_leaf: 2
max_depth: 20

Random Forest Model Performance:
Accuracy: 0.8590
Precision: 0.3511
Recall: 0.2529
F1 Score: 0.2940
ROC AUC: 0.7334
```




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

         0.0       0.91      0.94      0.92     45139
         1.0       0.35      0.25      0.29      5931

    accuracy                           0.86     51070
   macro avg       0.63      0.60      0.61     51070
weighted avg       0.84      0.86      0.85     51070
```





- **Non-defaults (0.0):**

  - Precision: 0.92 (92% of predicted non-defaults are actual non-defaults)
  - Recall: 0.86 (86% of actual non-defaults are correctly identified)
  - F1-score: 0.89 (very strong/balanced performace)

- **Defaults (1.0):**

  - Precision: 0.31 (31% of predicted defaults are actual defaults)
  - Recall: 0.46 (catches 46% of actual defaults)
  - F1-score: 0.37 (decent balance)

- **Most Important Feature:** Loan Purpose
- **Overall:** the model performs with 82% accuracy. It is still better at predicting non defaults over defaults, however even with balanceing the training data, the test data is still very imbalanced, so 46% recall is very good (and the best performing model so far).

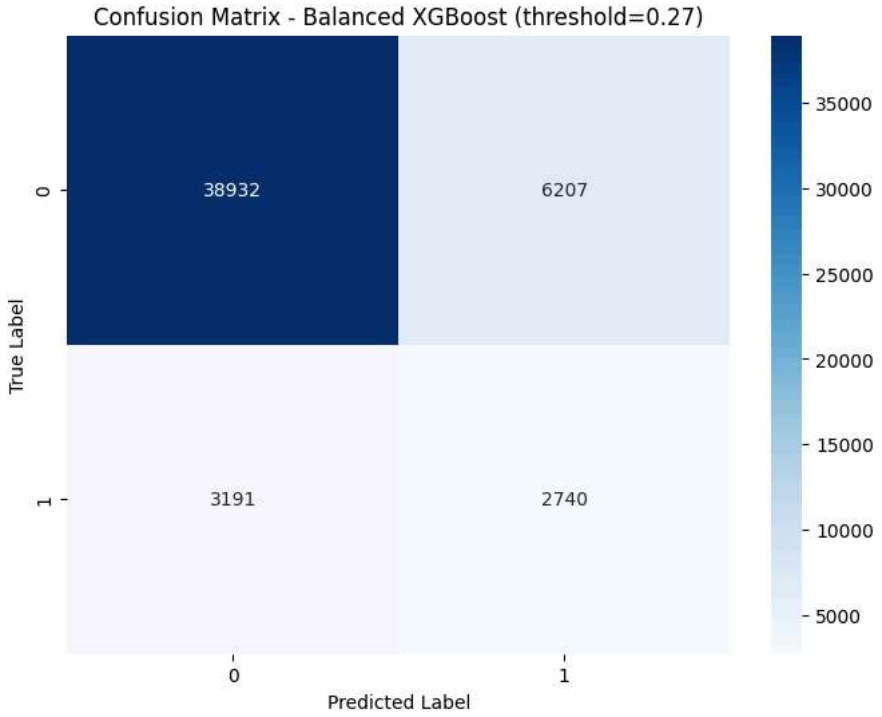
XGBoost

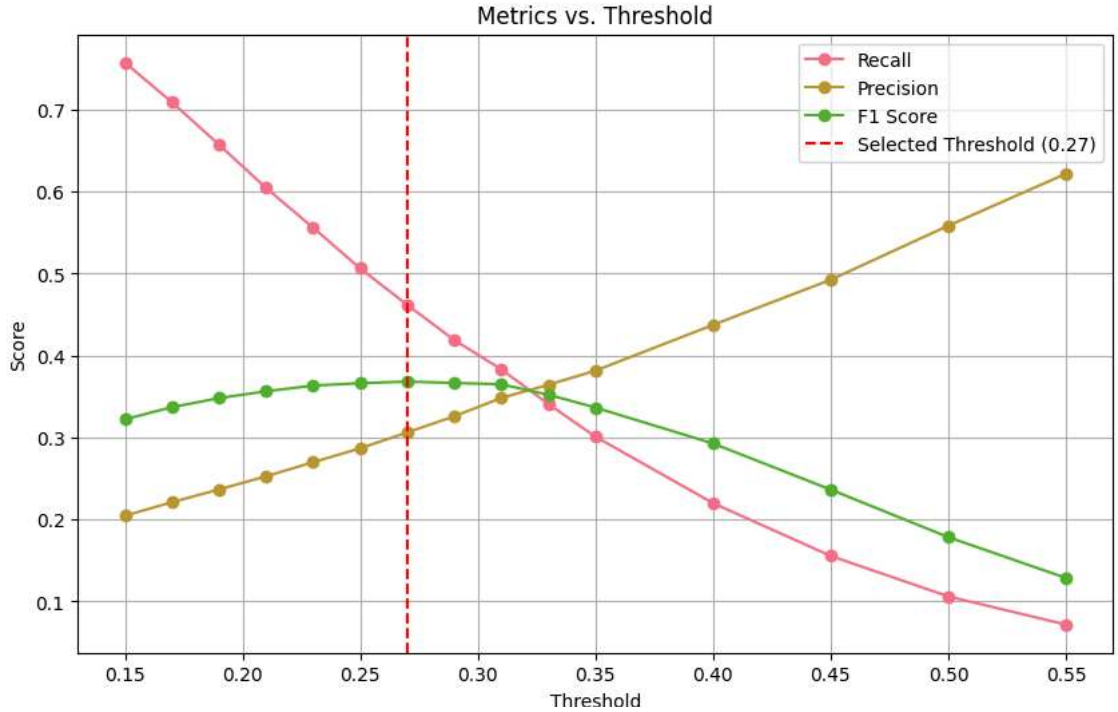
```
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import roc_auc_score, confusion_matrix, classification_report
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

def evaluate_xgboost_balanced(X_resampled, X_test, y_resampled, y_test):
    """
    Evaluate XGBoost with balanced threshold optimization
    """
    # Initialize XGBoost with balanced parameters
    xgb_model = XGBClassifier(
        # Moderate parameter settings
        max_depth=5,
        min_child_weight=2,
        learning_rate=0.05,
        n_estimators=400,
        subsample=0.8,
        colsample_bytree=0.8,

        # Moderate regularization
        reg_alpha=0.1,
        reg_lambda=1.0,

        # Class balancing
        scale_pos_weight=1.2,

        random_state=42,
        use_label_encoder=False,
        objective='binary:logistic'
    )

    # Train the model
    print("Training balanced XGBoost model...")
    eval_set = [(X_test, y_test)]
    xgb_model.fit(X_resampled, y_resampled,
                  eval_set=eval_set,
                  verbose=False)

    def evaluate_threshold(threshold):
        y_pred_proba = xgb_model.predict_proba(X_test)[:, 1]
        y_pred = (y_pred_proba >= threshold).astype(int)

        recall = recall_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
```

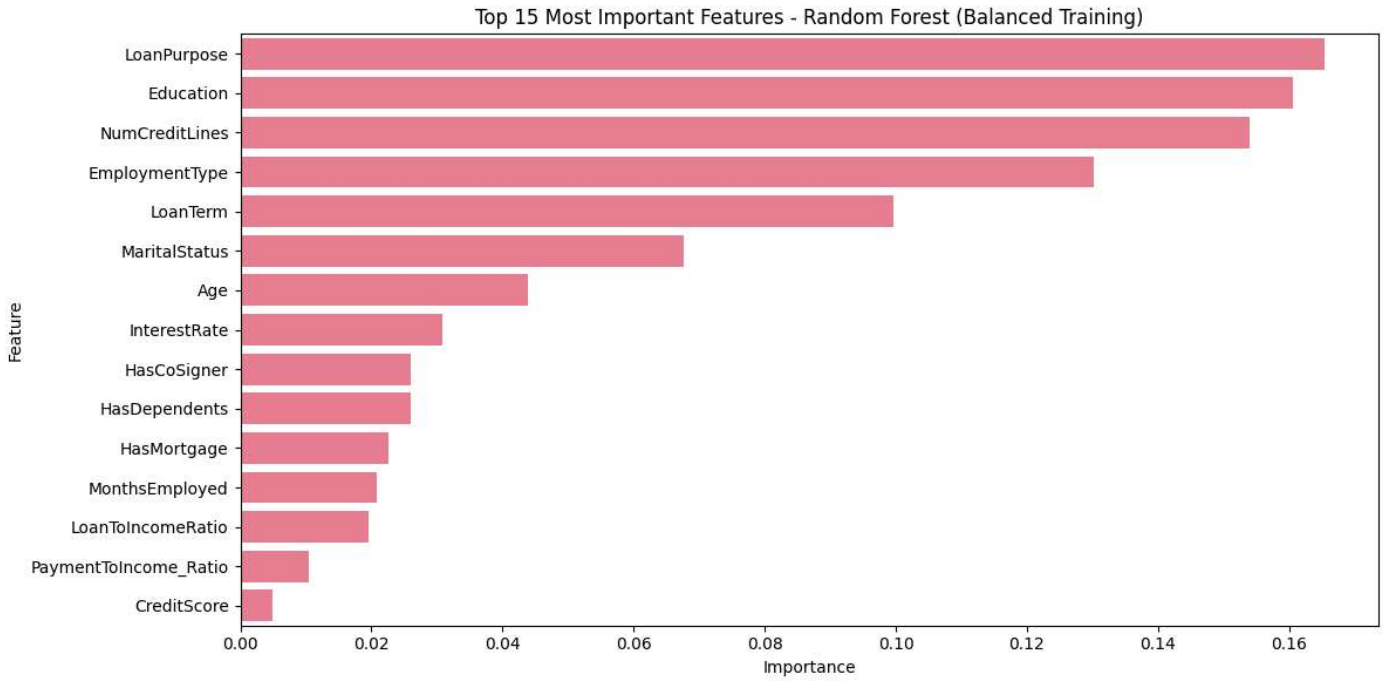
```
        f1 = f1_score(y_test, y_pred)

        # Calculate balanced accuracy (average of recall for both classes)
        spec = specificity_score(y_test, y_pred)
        balanced_acc = (recall + spec) / 2

        return {
            'threshold': threshold,
            'recall': recall,
            'precision': precision,
            'f1': f1,
            'balanced_acc': balanced_acc
        }

    def specificity_score(y_true, y_pred):
        tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
        return tn / (tn + fp)

    # Try different thresholds with finer granularity around middle range
    thresholds = np.concatenate([
        np.arange(0.15, 0.35, 0.02),  # More granular in middle range
        np.arange(0.35, 0.55, 0.05)   # Less granular outside
    ])

    threshold_results = [evaluate_threshold(t) for t in thresholds]

    # Find threshold that maximizes F1 score (balanced metric)
    best_threshold = max(threshold_results, key=lambda x: x['f1'])['threshold']

    print(f"\nTesting different prediction thresholds:")
    for result in threshold_results:
        print(f"Threshold {result['threshold']:.2f}: "
              f"Recall = {result['recall']:.3f}, "
              f"Precision = {result['precision']:.3f}, "
              f"F1 = {result['f1']:.3f}, "
              f"Balanced Acc = {result['balanced_acc']:.3f}")

    # Make predictions with best threshold
    y_pred_proba = xgb_model.predict_proba(X_test)[:, 1]
    y_pred = (y_pred_proba >= best_threshold).astype(int)

    # Calculate metrics
    xgb_metrics = {
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1 Score': f1_score(y_test, y_pred),
        'ROC AUC': roc_auc_score(y_test, y_pred_proba),
        'Balanced Accuracy': (recall_score(y_test, y_pred) +
                              specificity_score(y_test, y_pred)) / 2
    }

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'Confusion Matrix - Balanced XGBoost (threshold={best_threshold:.2f})')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

    # Plot Precision-Recall vs Threshold
    plt.figure(figsize=(10, 6))
    threshold_df = pd.DataFrame(threshold_results)
    plt.plot(threshold_df['threshold'], threshold_df['recall'],
             label='Recall', marker='o')
    plt.plot(threshold_df['threshold'], threshold_df['precision'],
             label='Precision', marker='o')
    plt.plot(threshold_df['threshold'], threshold_df['f1'],
             label='F1 Score', marker='o')
    plt.axvline(x=best_threshold, color='r', linestyle='--',
                label=f'Selected Threshold ({best_threshold:.2f})')
    plt.title('Metrics vs. Threshold')
    plt.xlabel('Threshold')
    plt.ylabel('Score')
    plt.legend()
    plt.grid(True)
```

```
    plt.show()

    # Calculate and plot feature importance
    feature_importance = pd.DataFrame({
        'Feature': X_resampled.columns,
        'Importance': xgb_model.feature_importances_
    })
    feature_importance = feature_importance.sort_values('Importance', ascending=False)

    plt.figure(figsize=(12, 6))
    sns.barplot(x='Importance', y='Feature', data=feature_importance.head(15))
    plt.title('Top 15 Most Important Features - Random Forest (Balanced Training)')
    plt.tight_layout()
    plt.show()

    print(f"\nUsing threshold {best_threshold:.2f} for final predictions")
    print("\nBalanced XGBoost Performance:")
    for metric, value in xgb_metrics.items():
        print(f"{metric}: {value:.4f}")

    # Print classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    return xgb_model, xgb_metrics, best_threshold

model, xgb_metrics, feature_importance = evaluate_xgboost_balanced(X_resampled, X_test, y_resampled, y_test)
```

```
Training balanced XGBoost model...
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarning: [23:39:53] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)

Testing different prediction thresholds:
Threshold 0.15: Recall = 0.758, Precision = 0.205, F1 = 0.322, Balanced Acc = 0.685
Threshold 0.17: Recall = 0.709, Precision = 0.221, F1 = 0.337, Balanced Acc = 0.690
Threshold 0.19: Recall = 0.657, Precision = 0.237, F1 = 0.348, Balanced Acc = 0.689
Threshold 0.21: Recall = 0.605, Precision = 0.253, F1 = 0.356, Balanced Acc = 0.685
Threshold 0.23: Recall = 0.556, Precision = 0.270, F1 = 0.363, Balanced Acc = 0.679
Threshold 0.25: Recall = 0.506, Precision = 0.287, F1 = 0.366, Balanced Acc = 0.671
Threshold 0.27: Recall = 0.462, Precision = 0.306, F1 = 0.368, Balanced Acc = 0.662
Threshold 0.29: Recall = 0.419, Precision = 0.326, F1 = 0.367, Balanced Acc = 0.653
Threshold 0.31: Recall = 0.383, Precision = 0.348, F1 = 0.365, Balanced Acc = 0.645
Threshold 0.33: Recall = 0.341, Precision = 0.364, F1 = 0.352, Balanced Acc = 0.631
Threshold 0.35: Recall = 0.301, Precision = 0.382, F1 = 0.337, Balanced Acc = 0.619
Threshold 0.40: Recall = 0.220, Precision = 0.437, F1 = 0.293, Balanced Acc = 0.591
Threshold 0.45: Recall = 0.156, Precision = 0.493, F1 = 0.237, Balanced Acc = 0.567
Threshold 0.50: Recall = 0.106, Precision = 0.559, F1 = 0.178, Balanced Acc = 0.548
Threshold 0.55: Recall = 0.072, Precision = 0.622, F1 = 0.129, Balanced Acc = 0.533
```








```
Using threshold 0.27 for final predictions

Balanced XGBoost Performance:
Accuracy: 0.8160
Precision: 0.3062
Recall: 0.4620
F1 Score: 0.3683
ROC AUC: 0.7563
Balanced Accuracy: 0.6622

Classification Report:
              precision    recall  f1-score   support

         0.0       0.92      0.86      0.89     45139
         1.0       0.31      0.46      0.37      5931

    accuracy                           0.82     51070
   macro avg       0.62      0.66      0.63     51070
weighted avg       0.85      0.82      0.83     51070
```

- **Non-defaults (0.0):**

  - Precision: 0.92 (92% of predicted non-defaults are actual non-defaults)
  - Recall: 0.86 (86% of actual non-defaults are correctly identified)
  - F1-score: 0.89 (very strong/balanced performace)

- **Defaults (1.0):**

  - Precision: 0.31 (31% of predicted defaults are actual defaults)
  - Recall: 0.46 (catches 46% of actual defaults)
  - F1-score: 0.37 (decent balance)

- **Most Important Feature:** Loan Purpose
- **Overall:** the model performs with 82% accuracy. It is still better at predicting non defaults over defaults, however even with balanceing the training data, the test data is still very imbalanced, so 46% recall is very good (and the best performing model so far).

Gradient Boosting

```
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import roc_auc_score, confusion_matrix, classification_report
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np

def evaluate_gradboost_balanced(X_resampled, X_test, y_resampled, y_test):
    """
    Evaluate Gradient Boosting with balanced threshold optimization
    """
    # Initialize Gradient Boosting with balanced parameters
    gb_model = GradientBoostingClassifier(
        # Moderate parameter settings
        max_depth=5,
        min_samples_split=5,
        learning_rate=0.05,
        n_estimators=400,
        subsample=0.8,

        # Additional parameters
        min_samples_leaf=2,
        max_features='sqrt',

        random_state=42
    )

    # Train the model
    print("Training balanced Gradient Boosting model...")
    gb_model.fit(X_resampled, y_resampled)

    def evaluate_threshold(threshold):
        y_pred_proba = gb_model.predict_proba(X_test)[:, 1]
        y_pred = (y_pred_proba >= threshold).astype(int)

        recall = recall_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred)

        # Calculate balanced accuracy (average of recall for both classes)
        spec = specificity_score(y_test, y_pred)
        balanced_acc = (recall + spec) / 2

        return {
            'threshold': threshold,
            'recall': recall,
            'precision': precision,
            'f1': f1,
            'balanced_acc': balanced_acc
```

```
    }

def specificity_score(y_true, y_pred):
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    return tn / (tn + fp)

# Try different thresholds with finer granularity around middle range
thresholds = np.concatenate([
    np.arange(0.15, 0.35, 0.02),  # More granular in middle range
    np.arange(0.35, 0.55, 0.05)   # Less granular outside
])

threshold_results = [evaluate_threshold(t) for t in thresholds]

# Find threshold that maximizes F1 score (balanced metric)
best_threshold = max(threshold_results, key=lambda x: x['f1'])['threshold']

print(f"\nTesting different prediction thresholds:")
for result in threshold_results:
    print(f"Threshold {result['threshold']:.2f}: "
          f"Recall = {result['recall']:.3f}, "
          f"Precision = {result['precision']:.3f}, "
          f"F1 = {result['f1']:.3f}, "
          f"Balanced Acc = {result['balanced_acc']:.3f}")

# Make predictions with best threshold
y_pred_proba = gb_model.predict_proba(X_test)[:, 1]
y_pred = (y_pred_proba >= best_threshold).astype(int)

# Calculate metrics
gb_metrics = {
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred),
    'Recall': recall_score(y_test, y_pred),
    'F1 Score': f1_score(y_test, y_pred),
    'ROC AUC': roc_auc_score(y_test, y_pred_proba),
    'Balanced Accuracy': (recall_score(y_test, y_pred) +
                          specificity_score(y_test, y_pred)) / 2
}

# Plot confusion matrix
plt.figure(figsize=(8, 6))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title(f'Confusion Matrix - Balanced Gradient Boosting (threshold={best_threshold:.2f})')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# Plot Precision-Recall vs Threshold
plt.figure(figsize=(10, 6))
threshold_df = pd.DataFrame(threshold_results)
plt.plot(threshold_df['threshold'], threshold_df['recall'],
         label='Recall', marker='o')
plt.plot(threshold_df['threshold'], threshold_df['precision'],
         label='Precision', marker='o')
plt.plot(threshold_df['threshold'], threshold_df['f1'],
         label='F1 Score', marker='o')
plt.axvline(x=best_threshold, color='r', linestyle='--',
            label=f'Selected Threshold ({best_threshold:.2f})')
plt.title('Metrics vs. Threshold')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()

# Calculate and plot feature importance
feature_importance = pd.DataFrame({
    'Feature': X_resampled.columns,
    'Importance': gb_model.feature_importances_
})
feature_importance = feature_importance.sort_values('Importance', ascending=False)

plt.figure(figsize=(12, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance.head(15))
plt.title('Top 15 Most Important Features - Gradient Boosting (Balanced Training)')
```