Credit Score Classification

Table of Contents:

1. Introduction

Accurate creditworthiness assessment is crucial for today's financial landscape. With increasing reliance on digital transactions, determining individual credit scores is essential for mitigating financial risks and promoting economic stability. As Abdou and Pointon(2011) highlight, credit scoring plays a crucial role in modern financial systems, enabling the categorisation and assessment of credit risk.

Using the "Credit Score Classification" dataset from Paris(2023) on Kaggle, this study aims to develop robust predictive models for classifying individuals into credit score categories. This dataset includes influential financial attributes like payment history, credit utilisation, credit age, and total debt. This research can improve financial institutions' credit risk management and inform personalised financial advice to enhance individual financial well-being.

Employing data preprocessing, exploratory analysis, feature engineering, model development, and evaluation with advanced machine learning and statistical analysis, this study will provide a detailed understanding of key factors influencing credit scores and the effectiveness of various predictive models.

Expected outcomes include identifying significant credit score predictors, developing accurate and interpretable classification models, and providing practical recommendations for financial stakeholders. These findings will contribute to credit risk assessment and offer valuable tools for financial decision-makers.

1.1 Problem Statement

Current credit scoring methods require further refinement for effective risk management in financial institutions. This study uses the "Credit Score Classification" dataset from Paris(2023) on Kaggle to develop improved segmentation models for accurate credit score classification, leveraging advanced machine learning to provide actionable insights into key influencing factors.

1.2 Objectives

1. Develop accurate credit score segmentation models using the Kaggle dataset.
2. Identify key predictors of credit scores for improved risk assessment.
3. Enhance financial decision-making and provide actionable insights for better credit health.

1.3 Data Source

This study uses the "Credit Score Classification" dataset from Kaggle (Paris,2023). This dataset contains key financial attributes for assessing creditworthiness, including payment history, credit utilisation, credit age, and total debt. Its structure and level of preprocessing includes handling of missing values, encoding categorical variables, and normalising financial features making it suitable for developing robust credit score classification models.

1.4 Methodology

Data Collection and Preprocessing:

The dataset will be examined thoroughly to understand its structure. Initial cleaning will address missing values, outliers, and inconsistencies to prepare the data for analysis.

Exploratory Data Analysis (EDA):

EDA techniques will be used to gain insights into the dataset. This includes visualizations and statistical analyses to understand variable distributions and identify key relationships between financial attributes and credit scores.

Feature Engineering:

Based on EDA insights, relevant features will be created and transformed to enhance the predictive power of the models. This step also involves selecting the most significant predictors for accurate classification.

Data Segmentation:

The dataset will be split into training and testing sets. Cross-validation techniques will be used to ensure that the models are properly validated and to prevent overfitting.

Model Development:

Various machine learning algorithms, including Random Forest, Gradient Boosting will be used to build predictive models for credit score classification.

Model Evaluation:

Given the potential for class imbalance in credit score datasets, the F1-score will be a key evaluation metric, along with accuracy, precision, recall, and AUC-ROC. As Saito and Rehmsmeier (2015) highlight, the F1-score effectively balances precision and recall in such scenarios. This makes the F1-score particularly suitable for credit score classification, where the inherent class imbalance necessitates a balanced evaluation of false positives and false negatives.

Hyperparameter Tuning:

Optuna study hyperparameter tuning will be employed to optimize model hyperparameters. The best parameters will be selected based on its performance on the cross-validation sets.

Interpretation and Analysis:

The results will be analysed to interpret model predictions and understand the key factors influencing credit scores. This will involve examining feature importance derived from the models.

Conclusion and Recommendations:

Based on the findings, conclusions will be drawn and practical recommendations will be provided for financial institutions to improve credit scoring methodologies and enhance credit risk management.

1.5 Expected Outcomes

This study anticipates several key outcomes. It is expected to develop accurate and robust segmentation models for credit score classification, contributing to more effective creditworthiness assessment by financial institutions. As Dumitrescu et al. (2022) suggest, ensemble methods based on decision trees, such as Random Forest, can offer improved classification performance compared to traditional methods like logistic regression, which informs the selection of models for this study. Furthermore, the study aims to identify key predictors of credit scores, providing valuable insights into the financial behaviours and attributes that most significantly influence credit risk, which could support more informed decision-making and improved risk management. Additionally, the research is expected to contribute to the improvement of current credit scoring methodologies by offering practical recommendations for their enhancement. Ultimately, the study aims to provide actionable insights that may aid in the development of personalized financial advice and interventions, potentially improving individual credit health and financial stability.

2. Data Understanding

The credit score classification dataset from Paris(2023) on Kaggle was selected as the data source fro this study due to its relevance to risk assessment and the availability of key financial attributes.

2.1 Import Libraries

Import necessary libraries for data manipulation, numerical operation, and visualisation.

In [1]:
```
import re
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import cufflinks as cf
import optuna
from scipy.stats import chi2_contingency
from sklearn.impute import KNNImputer
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder
from sklearn.compose import make_column_transformer
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import f1_score, confusion_matrix, classification_report, precision_score, recall_score
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from IPython.display import FileLink, display
import plotly.express as px
import plotly.graph_objects as go
from termcolor import colored
import warnings

# General settings
warnings.filterwarnings('ignore')
pd.set_option('display.float_format', lambda x: '%.3f' % x)
```

2.2 Load Data

Load the credit score file from the CVS file

In [ ]:
```
data = pd.read_csv("credit_score_data.csv")
```

3. Data Preparation

This section gets insights from values, handles missing values cutliers and performs necessary data transformation to prepare the data for modelling.

In [20]:
```
class DataAnalysis:
    def __init__(self, df):
        self.df = df

    def general_summary(self):
        print(f'Data shape (rows, columns): {self.df.shape}')
        print(f'Number of total duplicate rows: {self.df.duplicated().sum()}')
        print(colored("                                                    First 5 records of the dataset                                                  \n", 'blue', attrs=['bold']))
        return self.df.head(5)

    def object_summary(self):
        """
        This function provides a summary of categorical features in the DataFrame.
        """
        obs = self.df.shape[0]

        object_df = self.df.select_dtypes(include='object')
        summary_df = pd.DataFrame({
            'Dtype': object_df.dtypes,
            'Counts': object_df.apply(lambda x: x.count()),
            'Nulls': object_df.apply(lambda x: x.isnull().sum()),
            'NullPercent': (object_df.isnull().sum() / obs) * 100,
            'Top': object_df.apply(lambda x: x.mode()[0] if not x.mode().empty else '-'),
            'Frequency': object_df.apply(lambda x: x.value_counts().max() if not x.value_counts().empty else '-'),
            'Uniques': object_df.apply(lambda x: x.unique().shape[0]),
            'UniqueValues': object_df.apply(lambda x: list(x.unique()) if x.nunique() <= 20 else '-')
        })

        # Format 'NullPercent' to show percentages with two decimal points and a '%' sign
        summary_df['NullPercent'] = summary_df['NullPercent'].map("{:.2f}%".format)

        print('Categorical Features Summary:')
        print('_____________________________\nData Types:')
        print(summary_df['Dtype'].value_counts())
        print('_____________________________')

        return summary_df

    def numeric_summary(self):
        """
        This function provides a summary of numerical features in the DataFrame.
        """
        obs = self.df.shape[0]

        numeric_df = self.df.select_dtypes(include='number')
        summary_df = pd.DataFrame({
            'Dtype': numeric_df.dtypes,
            'Counts': numeric_df.apply(lambda x: x.count()),
            'Nulls': numeric_df.apply(lambda x: x.isnull().sum()),
            'NullPercent': (numeric_df.isnull().sum() / obs) * 100,
            'Min': numeric_df.min(),
            'Max': numeric_df.max(),
            'Uniques': numeric_df.apply(lambda x: x.unique().shape[0]),
            'UniqueValues': numeric_df.apply(lambda x: list(x.unique()) if x.nunique() <= 20 else '-')
        } )

        # Format 'NullPercent' to show percentages with two decimal points and a '%' sign
        summary_df['NullPercent'] = summary_df['NullPercent'].map("{:.2f}%".format)

        print('Numerical Features Summary:')
        print('_____________________________\nData Types:')
        print(summary_df['Dtype'].value_counts())
        print('_____________________________')

        return summary_df

def describe_by_type(self, dtype='object'):
    if dtype == 'object':
        # Only include object type columns
        obj_data = self.df.select_dtypes(include=['object'])
        return self.describe_object(obj_data)
    elif dtype == 'numeric':
        # Only include numeric type columns
        num_data = self.df.select_dtypes(include=['number'])
        return self.describe_numeric(num_data)
    else:
        raise ValueError("dtype must be 'object' or 'numeric'")

def describe_object(self, df):
    description = {}
    for column in df.columns:
        unique_count = df[column].nunique()
        top_value = df[column].mode()[0]
        freq_top_value = df[column].value_counts().iloc[0]
        description[column] = {
            'count': df[column].count(),
            'unique': unique_count,
            'top': top_value,
            'freq': freq_top_value
        }
    return pd.DataFrame(description).T

def describe_numeric(self, df):
    description = {}
    for column in df.columns:
        description[column] = {
            'count': df[column].count(),
            'mean': df[column].mean(),
            'std': df[column].std(),
            'min': df[column].min(),
            '25%': df[column].quantile(0.25),
            '50%': df[column].median(),
            '75%': df[column].quantile(0.75),
            'max': df[column].max()
        }
    return pd.DataFrame(description).T



def calculate_skewness(self, skew_limit=1):
    """ Calculates skewness for numeric features and returns those with high skewness.
    Parameters:
    skew_limit (float, optional): The limit to identify highly skewed data. Defaults to 1.
    Returns: pd.DataFrame: A DataFrame containing the skewness values of highly skewed numeric features. """
    # Selecting numerical columns
    num_cols = self.df.select_dtypes('number').columns
    # Calculating skewness
    skew_cols = self.df[num_cols].skew().loc[lambda x: (x > skew_limit) | (x < -skew_limit)].sort_values(ascending=False).to_frame('Skew')
    return skew_cols
```

No	INPUTS	Description
1	**ID**	Unique identifier for each record.
2	**Customer_ID**	Unique identifier for each customer.
3	**Month**	Month of the transaction or record.
4	**Name**	Customer's name.
5	**Age**	The customer's age.
6	**SSN**	Customer's social security number.
7	**Occupation**	The customer's occupation.
8	**Annual_Income**	The customer's annual income.
9	**Monthly_Inhand_Salary**	The customer's monthly in-hand salary.
10	**Num_Bank_Accounts**	Number of bank accounts owned by the customer.
11	**Num_Credit_Card**	Number of credit cards owned by the customer.
12	**Interest_Rate**	The interest rate applied to loans or credit.
13	**Num_of_Loan**	Number of loans taken by the customer.
14	**Type_of_Loan**	Type of loan taken by the customer.
15	**Delay_from_due_date**	The delay in payment from the due date.
16	**Num_of_Delayed_Payment**	Number of delayed payments made by the customer.
17	**Changed_Credit_Limit**	Changes made to the customer's credit limit.
18	**Num_Credit_Inquiries**	Number of credit inquiries made.
19	**Credit_Mix**	The mix of credit types the customer uses (e.g., loans, credit cards).
20	**Outstanding_Debt**	Total outstanding debt the customer has.
21	**Credit_Utilization_Ratio**	The ratio of credit used to the total credit limit.
22	**Credit_History_Age**	The length of the customer's credit history.
23	**Payment_of_Min_Amount**	Whether the customer pays the minimum amount required each month.
24	**Total_EMI_per_month**	The total EMI (Equated Monthly Installment) the customer pays each month.
25	**Amount_invested_monthly**	The amount invested by the customer each month.
26	**Payment_Behaviour**	The payment behavior of the customer.
27	**Monthly_Balance**	The customer's remaining balance at the end of each month.
28	**Credit_Score**	The customer's credit score (target variable: "Good," "Poor," "Standard").

The dataset structure

The dataset is structured for credit score prediction (target variable: Credit_Score) and includes a variety of relevant features, such as age, annual income (numerical), payment history (categorical), credit utilisation (numerical), number of loans (numerical), and outstanding debt (numerical). These attributes provide a basis for assessing creditworthiness and developing predictive and classification models for credit score assessment.

In [ ]:
```
analyzer = DataAnalysis(data)
analyzer.general_summary()
```

In [25]:
```
analyzer.object_summary()
```

```
Categorical Features Summary:
_____________________
Data Types:
Dtype
object    20
Name: count, dtype: int64
_____________________
```

Out[25]:

	Dtype	Counts	Nulls	NullPercent	Top	Frequency	Uniques	UniqueValues
ID	object	100000	0	0.00%	0x10002	1	100000	-
Customer_ID	object	100000	0	0.00%	CUS_0x1000	8	12500	-
Month	object	100000	0	0.00%	April	12500	8	[January, February, March, April, May, June, J...
Name	object	90015	9985	9.98%	Langep	44	10140	-
Age	object	100000	0	0.00%	38	2833	1788	-
SSN	object	100000	0	0.00%	#F%$D@*&8	5572	12501	-
Occupation	object	100000	0	0.00%	_____	7062	16	[Scientist, _____, Teacher, Engineer, Entrep...
Annual_Income	object	100000	0	0.00%	17273.83	16	18940	-
Num_of_Loan	object	100000	0	0.00%	3	14386	434	-
Type_of_Loan	object	88592	11408	11.41%	Not Specified	1408	6261	-
Num_of_Delayed_Payment	object	92998	7002	7.00%	19	5327	750	-
Changed_Credit_Limit	object	100000	0	0.00%	_	2091	4384	-
Credit_Mix	object	100000	0	0.00%	Standard	36479	4	[_, Good, Standard, Bad]
Outstanding_Debt	object	100000	0	0.00%	1360.45	24	13178	-
Credit_History_Age	object	90970	9030	9.03%	15 Years and 11 Months	446	405	-
Payment_of_Min_Amount	object	100000	0	0.00%	Yes	52326	3	[No, NM, Yes]
Amount_invested_monthly	object	95521	4479	4.48%	__10000__	4305	91050	-
Payment_Behaviour	object	100000	0	0.00%	Low_spent_Small_value_payments	25513	7	[High_spent_Small_value_payments, Low_spent_La...
Monthly_Balance	object	98800	1200	1.20%	__-333333333333333333333333__	9	98793	-
Credit_Score	object	100000	0	0.00%	Standard	53174	3	[Good, Standard, Poor]

In [26]:
```
analyzer.numeric_summary()
```

```
Numerical Features Summary:
_____________________
Data Types:
Dtype
float64    4
int64      4
Name: count, dtype: int64
_____________________
```

Out[26]:

	Dtype	Counts	Nulls	NullPercent	Min	Max	Uniques	UniqueValues
Monthly_Inhand_Salary	float64	84998	15002	15.00%	303.645	15204.633	13236	-
Num_Bank_Accounts	int64	100000	0	0.00%	-1.000	1798.000	943	-
Num_Credit_Card	int64	100000	0	0.00%	0.000	1499.000	1179	-
Interest_Rate	int64	100000	0	0.00%	1.000	5797.000	1750	-
Delay_from_due_date	int64	100000	0	0.00%	-5.000	67.000	73	-
Num_Credit_Inquiries	float64	98035	1965	1.97%	0.000	2597.000	1224	-
Credit_Utilization_Ratio	float64	100000	0	0.00%	20.000	50.000	100000	-
Total_EMI_per_month	float64	100000	0	0.00%	0.000	82331.000	14950	-

3.1 Data Cleaning

The data cleaning process will address several key data quality issues to prepare the dataset for analysis. Missing values will be handled strategically depending on the column. Additionally, this section will include data type checks, imputations, and visualizations to ensure comprehensive data preparation.

In [29]:
```
class DataCleaner:
    def __init__(self, df):
        self.df = df

    def remove_duplicates(self):
        """ Removes duplicate rows from the DataFrame.
        This method checks for duplicate rows in the DataFrame. If any duplicates are found, they are removed, and the updated DataFrame is returned.
        The method also prints the number of duplicate rows found and the number of duplicate rows after removal.
        Returns: pandas.DataFrame: The DataFrame with duplicate rows removed.
        """
        duplicate_count = self.df.duplicated().sum()
        print(f'Number of total duplicate rows: {duplicate_count}')

        if duplicate_count > 0:
            print("Duplicate records found. Removing duplicates...")
            self.df.drop_duplicates(inplace=True)
            print(f'Number of total duplicate rows after removal: {self.df.duplicated().sum()}')
        else:
            print("No duplicate records found.")

        return self.df

    def replace_empty_with_nan(self, column_name):
        """ Replaces empty string values in the specified column with NaN.
        This method replaces empty string values in the specified column of the DataFrame with NaN values.
        This is useful for ensuring that missing values are properly represented as NaN for further analysis or cleaning.

        Parameters:
        column_name (str): The name of the column in which to replace empty strings. """
        self.df[column_name].replace('', np.nan, inplace=True)
        print(f"Empty strings in column '{column_name}' replaced with NaN.")

    def replace_value(self, column_name, old_value, new_value):
        """ Replaces the specified old value with the new value in the given column.
        Parameters:
        column_name (str): The name of the column in which to replace values.
        old_value (str): The value to be replaced.
        new_value (str): The new value to replace the old value.
        Returns: None: The method updates the DataFrame in place. """
        self.df[column_name].replace(old_value, new_value, inplace=True)
        print(f"Value '{old_value}' in column '{column_name}' has been replaced with '{new_value}'.")

    def get_value_count(self, column_name,dropna=True):
        """
        This function calculates and returns a DataFrame with the value counts and
        their corresponding percentages for a specified column in the DataFrame.
        """
        vc = self.df[column_name].value_counts(dropna)
        vc_norm = self.df[column_name].value_counts(normalize=True)

        vc = vc.rename_axis(column_name).reset_index(name='counts')
        vc_norm = vc_norm.rename_axis(column_name).reset_index(name='percent')
        vc_norm['percent'] = (vc_norm['percent'] * 100).map('{:.2f}%'.format)

        df_result = pd.concat([vc[column_name], vc['counts'], vc_norm['percent']], axis=1)

        return df_result

    def missing_values(self):
        """
        This function calculates the missing values count and their percentage in a DataFrame.
        """
        missing_count = self.df.isnull().sum()
        value_count = self.df.isnull().count()
        missing_percentage = round(missing_count / value_count * 100, 2)

        # Format the percentage as '0.00%' with % symbol
        missing_percentage_formatted = missing_percentage.map("{:.2f}%".format)
        missing_df = pd.DataFrame({"count": missing_count, "percentage": missing_percentage_formatted})

        return missing_df

    def na_ratio_plot(self):
        """
        Plots the ratio of missing values for each feature and prints the count of missing values.
        """
        sns.displot(self.df.isna().melt(value_name='Missing_data', var_name='Features'),
                    y='Features', hue='Missing_data', multiple='fill', aspect=9/8)
        print(self.df.isna().sum()[self.df.isna().sum() > 0])

    def find_non_numeric_values(self, column_name):
        """
        Finds unique non-numeric values in a specified column of the DataFrame.
        """
        pattern = r'\D+'  # Pattern to match non-numeric characters
        # Find and flatten non-numeric values, then ensure uniqueness with set
        print("Found non numeric values: ")
        return set(re.findall(pattern, ' '.join(self.df[column_name].astype(str))))

    def drop_columns(self, columns):
        """ Drops specified columns that are not useful for analysis or modeling.
        Parameters:
        columns (list): A list of column names to be dropped from the DataFrame.
        Returns: None: The method updates the DataFrame in place, removing the specified columns. """
        try:
            self.df.drop(columns=columns, errors='ignore', inplace=True)
            print(f"Columns {columns} dropped (if they existed).")
        except Exception as e:
            print(f"An error occurred while dropping columns {columns}: {e}")

    def convert_object_to_float(self, column_name):
```

```
    """ Converts a specified column from object (string) type to float type.

    This method attempts to convert the specified column in the DataFrame from object (string) type to float type.
    If the conversion fails, an error message is printed and the DataFrame remains unchanged.

    Parameters:
    column_name (str): The name of the column to be converted. """
    try:
        self.df[column_name] = pd.to_numeric(self.df[column_name], errors='coerce')
        print(f"Column '{column_name}' successfully converted to float.")
    except Exception as e:
        print(f"Error converting column '{column_name}' to float: {e}")

def convert_to_absolute(self, column_name):
    """ Converts the values in the specified column to their absolute values.
    This method converts all values in the specified column of the DataFrame to their absolute values.
    This is useful for ensuring that numerical values are non-negative.

    Parameters:
    column_name (str): The name of the column to be converted to absolute values. """
    self.df[column_name] = self.df[column_name].abs()
    print(f"Values in column '{column_name}' have been converted to their absolute values.")

def print_missing_values(self, column_name):
    """ This function prints the count of missing values in the specified column. """
    missing_count = self.df[column_name].isna().sum()
    print(f'Remaining missing values in {column_name}:', missing_count)

def count_negative_values(self, column_name):
    """ Prints the list of unique negative values in the specified column and the count of rows with negative values.
    Parameters:
    column_name (str): The name of the column to check for negative values.
    Returns: None: The method prints the unique negative values and their count. """
    negative_values = self.df[self.df[column_name] < 0][column_name].unique()
    negative_count = self.df[self.df[column_name] < 0][column_name].count()
    print(f'List of unique negative values in column "{column_name}":', negative_values)
    print(f'Number of unique negative values in column "{column_name}":', negative_count)

def print_column_dtype(self, column_name):
    """ This function prints the data type of the specified column. """
    dtype = self.df[column_name].dtypes
    print(f'dtype of {column_name}:', dtype)

def remove_non_numeric_characters(self, column_name):
    """ This function removes non-numeric characters from the specified column. """

    try:
        self.df[column_name] = self.df[column_name].apply(lambda x: re.sub(r'[^0-9.]', '', str(x)))

        print(f"Non-numeric characters removed successfully from column '{column_name}'.")
    except Exception as e:

        print(f"An error occurred while removing non-numeric characters from column '{column_name}': {e}")

def transform_loan_types(self, loan_col):
    """ This function transforms the Type_of_Loan column by separating multiple loan types into individual binary columns. Each new column indicates whether a specific loan type is present in that row. The original Type_of_Loan column is then deleted. """
    # Get unique loan types
    unique_loan_types = self.df[loan_col].dropna().str.split(',').explode().str.strip().unique()
    unique_loan_types = [loan.replace('and ', '').strip() for loan in unique_loan_types]
    # Create binary columns for each loan type
    for loan_type in unique_loan_types:
        self.df[loan_type] = self.df[loan_col].str.contains(loan_type, na=False).astype(int)
        # Delete the original loan type column
    self.df.drop(columns=[loan_col], inplace=True)
    return self.df
def convert_credit_history_age(self, column_name):
    """ Converts a credit history age string (in the format 'X Years and Y Months') into a total number of months.
    This method converts the credit history age from a string format to a total number of months.
    If the input string is not in the correct format or an error occurs, NaN is returned.
    Parameters:
    column_name (str): The name of the column to be converted.
    Returns: None: The method updates the DataFrame in place. """

    self.df[column_name] = self.df[column_name].apply(self.history_age_month)
    print(f"Column '{column_name}' has been converted to total months.")

@staticmethod
def history_age_month(age):

    try:
        # Extract the years part (before "and")
        years = int(re.findall(r'\d+', age.split("and")[0])[0])

        # Extract the months part (after "and")
        months = int(re.findall(r'\d+', age.split("and")[1])[0])

        # Convert years to months and add the months part
        return int(years * 12 + months)

    except (IndexError, ValueError, AttributeError):

        # Return NaN if the input string is not in the correct format or an error occurs
        return np.nan

    def replace_unusual_values(self, column_name, unusual_value):
        """ Replaces unusual values in the specified column with NaN.
        This method replaces a specified unusual value in the specified column of the DataFrame with NaN.
        This is useful for ensuring that erroneous values are properly handled for further analysis or cleaning.

        Parameters:
        column_name (str): The name of the column in which to replace unusual values.
        unusual_value (str): The unusual value to be replaced with NaN.
        Returns: None: The method updates the DataFrame in place. """

        self.df[column_name].replace(unusual_value, np.NaN, inplace=True)
        print(f"Unusual value '{unusual_value}' in column '{column_name}' has been replaced with NaN.")

    def convert_hex_to_int(self, column_name):
        """ Converts hexadecimal string values in the specified column to integers.
        This method converts values in the specified column from hexadecimal strings to integers, if they are recognized as hexadecimal (i.e., strings starting with '0x').
        Parameters:
        column_name (str): The name of the column to be converted. Returns: None: The method updates the DataFrame in place. """

        self.df[column_name] = self.df[column_name].apply(lambda x: int(x, 16) if isinstance(x, str) and x.startswith('0x') else x)
        print(f"Hexadecimal values in column '{column_name}' have been converted to integers.")

    def cap_age_values(self, column_name, lower_bound=18, upper_bound=120):
        """ Caps the values in the specified age column to a reasonable range by replacing values outside the specified lower and upper bounds with NaN.
        Parameters:
        column_name (str): The name of the age column to be capped.
        lower_bound (int, optional): The lower bound for age values. Defaults to 18.
        upper_bound (int, optional): The upper bound for age values. Defaults to 120.
        Returns: None: The method updates the DataFrame in place. """

        self.df.loc[(self.df[column_name] > upper_bound) | (self.df[column_name] < lower_bound), column_name] = np.nan
        print(f"Values in column '{column_name}' outside the range {lower_bound}-{upper_bound} have been replaced with NaN.")

    def save_and_link_cleaned_data(self,file_name):
        self.df.to_csv(file_name, index=False)
        cleaned_file = FileLink(file_name, result_html_prefix="Click here to download: ")
        display(cleaned_file)
```

In [30]:

```
class DataImputer:
    def __init__(self, df):
        """
        Initializes the DataImputer with a pandas DataFrame.

        Parameters:
        df (pandas.DataFrame): The DataFrame to be imputed.
        """
        self.df = df

    def fill_missing_monthly_salary(self, annual_income_col, monthly_salary_col):
        """ This function fills missing values in the specified monthly salary column by dividing the specified annual income column by 12. """
        try:
            self.df[monthly_salary_col].fillna(self.df[annual_income_col] / 12, inplace=True)
            print(f"Missing values in column '{monthly_salary_col}' filled successfully by dividing '{annual_income_col}' by 12.")
        except Exception as e:
            print(f"An error occurred while filling missing values in column '{monthly_salary_col}': {e}")



    def knn_impute_column(self, column, n_neighbors=5):
        """
        Impute missing values in the specified column using KNN.

        This method imputes missing values in the specified column using KNN
        imputation. It uses the number of neighbors specified to determine
        the imputed values.

        Parameters:
        column (str): The name of the column to be imputed.
        n_neighbors (int): The number of neighbors to use for KNN imputation (default is 5).

        Returns:
        None: The method updates the DataFrame in place.
        """
        # Apply KNN imputation to the specified column
        imputer = KNNImputer(n_neighbors=n_neighbors)
        self.df[[column]] = imputer.fit_transform(self.df[[column]])
        print(f"Missing values in column '{column}' have been imputed using KNN with {n_neighbors} neighbors.")

    def fill_payment_columns(self, payment_column, group_columns):

        """ Fills missing values in the specified payment behavior column based on the mode of grouped data.
        Parameters:
        payment_column (str): The name of the payment behavior column to be filled.
        group_columns (list): The list of columns to group by for filling missing values.
        Returns: None: The method updates the DataFrame in place. """

        self.df[payment_column] = self.df.groupby(group_columns)[payment_column].transform( lambda x: x.fillna(x.mode()[0] if not x.mode().empty else 'Unknown'))

        print(f"Missing values in column '{payment_column}' have been filled based on groups {group_columns}")


    def impute_missing_values(self, target_column, groupby_column):
```

```
        """ Imputes missing values in the target column by grouping the data based on the groupby column and filling the missing values with the mean of each group.

        Parameters:
        target_column (str): The name of the column to impute missing values for.
        groupby_column (str): The name of the column to group by for calculating the mean.
        Returns: None: The method updates the DataFrame in place. """
        self.df[target_column] = self.df.groupby(groupby_column)[target_column].transform(lambda x: x.fillna(x.mean()))
        self.df[target_column] = self.df[target_column].transform(lambda x: x.fillna(x.mean()))
        print(f"Missing values in column '{target_column}' have been imputed using the mean of groups based on '{groupby_column}'.")
```

In [31]:

```
class DataVisualizer:
    def __init__(self, df):
        self.df = df

    def plot_average_by_column(self, group_column, value_column):
        """ Plots the average values of a specified column, grouped by another column, using a bar plot.
        This function generates a bar plot to visualize the average values of a specified column, grouped by another column.
        The plot also annotates the bars with their respective values.

        Parameters:
        group_column (str): The column name to group by.
        value_column (str): The column name whose average values are to be plotted.

        Returns:
        None: The function displays the plot. """
        plt.figure(figsize=(10, 5))
        ax = sns.barplot(x=group_column, y=value_column, data=self.df, ci=None, palette='flare')

        for p in ax.patches:
            ax.annotate(format(p.get_height(), '.2f'),
                        (p.get_x() + p.get_width() / 2., p.get_height()),
                        ha='center', va='center',
                        xytext=(0, 9), textcoords='offset points')

        plt.title(f'Average {value_column.replace("_", " ").title()} by {group_column.replace("_", " ").title()}')
        plt.xlabel(group_column.replace("_", " ").title())
        plt.ylabel(f'Average {value_column.replace("_", " ").title()}')

        plt.show()


    def plot_boxplot_num_cols(self):
        """
        Plots boxplots for all numerical columns in the DataFrame.

        This method creates a boxplot for each numerical column in the DataFrame and arranges them
        in a grid layout for easy comparison.

        Returns:
        None: The method displays the boxplots.
        """
        num_columns = self.df.select_dtypes(include="number").columns
        num_columns =  [col for col in num_columns if 'Loan' not in col]
        num_plots = len(num_columns)

        # Determine the number of rows and columns for the subplot grid
        cols = 4
        rows = (num_plots // cols) + (num_plots % cols > 0)

        fig = plt.figure(figsize=(18, 14), dpi=300)
        for idx, column in enumerate(num_columns):
            ax = fig.add_subplot(rows, cols, idx + 1)
            sns.boxplot(x=self.df[column], ax=ax)
            ax.set_title(f'Boxplot of {column}')
        plt.tight_layout()
        plt.show()


    def plot_pie_chart_bin(self, column_name, bins):
        """ Plots a pie chart of the distribution of a specified column by binning its values.
        This function generates a pie chart to visualize the distribution of a specified column's values, which are grouped into bins.
        The pie chart displays the proportion of each bin as a percentage.

        Parameters:
        column_name (str): The column name whose values are to be binned and plotted.
        bins (int or list-like): The criteria to bin the column values. It can be an integer number of bins or a sequence of bin edges.

        Returns:
        None: The function displays the pie chart. """
        income_bins = pd.cut(self.df[column_name], bins=bins)
        income_distribution = income_bins.value_counts()
        income_distribution.plot.pie(autopct='%1.1f%%', colors=sns.color_palette('pastel'))
        plt.title(f'Income Distribution of {column_name.replace("_", " ").title()}')
        plt.ylabel('')
        plt.show()


    def plot_loan_distribution_by_credit_score(self):
        """
        Plots the distribution of loan types by credit score using a count plot.

        This function generates a count plot to visualize the distribution of different loan types
        across various credit score categories. The plot shows the count of each loan type for
        each credit score category, with loans indicated by a value of 1.

        Parameters:
        column_name (str): The column name for the credit score.
        value_column (str): The column name for the loan types.

        Returns:
        None: The function displays the count plot.
        """

        loan_columns = [col for col in self.df.columns if col.endswith('Loan')]
        df_melted = self.df.melt(id_vars='Credit_Score', value_vars=loan_columns,
                                 var_name='Loan Type', value_name='Has Loan')

        plt.figure(figsize=(20,8))
        sns.countplot(x='Loan Type', hue='Credit_Score', data=df_melted[df_melted['Has Loan'] == 1], palette='inferno')

        plt.title('Distribution of Loan Types by Credit Score')
        plt.xlabel('Loan Type')
        plt.ylabel('Count')
        plt.show()

    def plot_correlation_with_columns(self, target_column, columns):
        """ Plots a heatmap of correlations between the specified target column and other numerical columns in the DataFrame.
        This method calculates the correlation of the specified target column with the provided list of columns in the DataFrame and visualizes these correlations using a heatmap.

        Parameters:
        target_column (str): The name of the target column for which to plot correlations.
        columns (list): A list of column names to be correlated with the target column.
        Returns: None: The method displays a heatmap of the correlations. """

        plt.figure(figsize=(10, 8)) # Calculate the correlations of the specified column with the provided list of columns
        corr_matrix = self.df[columns + [target_column]].corr().abs()
        column_corr = corr_matrix[[target_column]].sort_values(by=target_column, ascending=False) # Plot the heatmap
        sns.heatmap(column_corr, annot=True, cmap='coolwarm', vmin=0, vmax=1)
        plt.title(f'Correlation of Other Features with {target_column.replace("_", " ").title()}')
        plt.show()

    def plot_month_distribution_by_credit_score(self, month_column, credit_score_column):
        """ Plots a stacked bar chart showing the distribution of the specified month column by credit score.

        Parameters:
        month_column (str): The name of the month column.
        credit_score_column (str): The name of the credit score column.
        Returns: None: The method displays the stacked bar chart. """
        self.df.groupby([month_column, credit_score_column]).size().unstack().plot( kind='bar', stacked=True, figsize=(18, 8), cmap='viridis')
        # Add labels and title
        plt.xlabel(month_column.replace('_', ' ').title())
        plt.ylabel('Count')
        plt.title(f'{month_column.replace("_", " ").title()}  Distribution by {credit_score_column.replace("_", " ").title()}')
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()

    def plot_age_distribution_by_credit_score(self, x_column, y_column):
        """ Plots a boxplot to visualize the distribution of the specified y_column based on x_column using Plotly Express.
        Parameters:
        x_column (str): The name of the column for the x-axis.
        y_column (str): The name of the column for the y-axis.
        Returns: None: The method displays the boxplot. """
        fig = px.box(self.df, x=x_column, y=y_column, title=f'{y_column.replace("_", " ").title()} Distribution by {x_column.replace("_", " ").title()}')
        fig.update_traces(boxmean=True) # Show mean
        for trace in fig.data:
            trace.hoverinfo = 'none' # Disable hover info to show values by default

        fig.show()

    def plot_count(self, x_column, hue_column):
        """ Plots a count plot of the specified x_column, with data grouped by the hue_column.
        Parameters:
        x_column (str): The name of the column for the x-axis.
        hue_column (str): The name of the column to group data by (for hue).
        Returns: None: The method displays the count plot. """
        plt.figure(figsize=(24, 12))
        ax = sns.countplot(x=x_column, hue=hue_column, data=self.df, palette='inferno')
        # Add labels and title
        plt.title(f'{x_column.replace("_", " ").title()} Distribution by {hue_column.replace("_", " ").title()}')
        plt.xlabel(x_column.replace('_', ' ').title())
        plt.ylabel('Count')
        plt.tight_layout()
        plt.show()

    def plot_count_single(self, column_name):
        """ Plots the distribution of the specified column name, typically Credit Score.
        Parameters:
        column_name (str): The name of the column to plot the distribution for.
        Returns: None: The method displays the count plot. """
        plt.figure(figsize=(10, 5))
        ax = sns.countplot(x=column_name, data=self.df[self.df[column_name].notnull()], palette='inferno') # Add values on top of the bars
        for p in ax.patches:
            ax.annotate(format(p.get_height(), '.2f'),
                        (p.get_x() + p.get_width() / 2., p.get_height()),
                        ha='center', va='center',
```

```
                    xytext=(0, 9), textcoords='offset points')
        plt.title(f'Distribution of {column_name.replace("_", " ").title()}: Target Feature')
        plt.xlabel(column_name.replace('_', ' ').title())
        plt.ylabel('Count')
        plt.show()
        plt.close()

    plt.close()

def plot_pie_chart(self, column_name):
    """  Plots a pie chart for the specified column name, showing labels and percentages.

    Parameters:
    column_name (str): The name of the column to plot the pie chart for.

    Returns:
    None: The method displays the pie chart. """
    data = self.df[column_name].value_counts()
    plt.figure(figsize=(8, 8))
    wedges, texts, autotexts = plt.pie(data, labels=data.index, autopct='%1.1f%%', startangle=140, colors=sns.color_palette('inferno', len(data)))

    # Set properties of the autopct texts (percentages)
    for autotext in autotexts:
        autotext.set_color('white')
        autotext.set_fontsize(10)
        autotext.set_fontweight('bold')

    plt.title(f'Pie Chart of {column_name.replace("_", " ").title()}')
    plt.show()
    plt.close()

def plot_boxplot_outliers(self, iqr_threshold=2000):
    """ Plots a boxplot for numerical columns with significant IQR, highlighting outliers.
    Parameters:
    iqr_threshold (int, optional): The threshold for IQR to filter columns with a significant range. Defaults to 2000.
    Returns: None: The method displays the boxplot. """
    cf.go_offline()
    # Selecting only numerical columns
    df_num = self.df.select_dtypes(include='number')
    # Calculate IQR for each numerical column
    iqr_values = df_num.apply(lambda x: x.quantile(0.75) - x.quantile(0.25))
    # Filter columns with a significant IQR
    selected_columns = iqr_values[iqr_values > iqr_threshold].index
    # Filter the DataFrame with the selected columns
    df_filtered = df_num[selected_columns]
    # Plotting the boxplot for the filtered numerical columns
    df_filtered.iplot(kind="box")
```

```
def plot_correlation_matrix(self, variable_type='number'):
    """
    Plots a correlation matrix heatmap for the specified variable type (number or object).

    Parameters:
    variable_type (str): The type of variables to include in the correlation matrix ('number' or 'object').

    Returns:
    None: The method displays the correlation matrix heatmap.
    """
    if variable_type == 'number':
        df_numeric = self.df.select_dtypes(include=['number'])
        corr_matrix = df_numeric.corr(method='pearson')
        plt.figure(figsize=(20, 16))
        sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm', vmin=-1, vmax=1, cbar=True)
        plt.title('Correlation Matrix for Numerical Features')
        plt.show()
    elif variable_type == 'object':
        df_object = self.df.select_dtypes(include=['object'])
        matrix = np.zeros((len(df_object.columns), len(df_object.columns)))
        for i, col1 in enumerate(df_object.columns):
            for j, col2 in enumerate(df_object.columns):
                if i == j:
                    matrix[i, j] = 1.0
                else:
                    contingency_table = pd.crosstab(df_object[col1], df_object[col2])
                    chi2 = chi2_contingency(contingency_table)[0]
                    n = contingency_table.sum().sum()
                    matrix[i, j] = np.sqrt(chi2 / (n * (min(contingency_table.shape) - 1)))

        corr_matrix = pd.DataFrame(matrix, index=df_object.columns, columns=df_object.columns)
        plt.figure(figsize=(12, 10))
        sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm', vmin=0, vmax=1, cbar=True)
        plt.title('Correlation Matrix for Categorical Features')
        plt.show()


def plot_numeric_scatter(self):
    """
    Plots scatter plots and histograms for numeric data, excluding columns containing 'Loan'.

    This function generates a pair grid to visualize scatter plots and histograms for the
    numeric columns in the DataFrame, excluding any columns that contain the word 'Loan'.
    The pair grid includes diagonal histograms and off-diagonal scatter plots for each pair of numeric columns.

    Parameters:
    None: The function operates on the DataFrame attribute of the class.

    Returns:
    None: The function displays the pair grid plots.
    """


    numeric_data = self.df.select_dtypes(include=['number'])


    filtered_columns = [col for col in numeric_data.columns if 'Loan' not in col]
    numeric_data = numeric_data[filtered_columns]


    g = sns.PairGrid(numeric_data)
    g.map_diag(sns.histplot)
    g.map_offdiag(sns.scatterplot)
    plt.show()


def plot_bar_and_count(self):
    """
    Plots count plots and bar charts for each categorical column in the DataFrame.

    This function generates count plots and bar charts for each categorical column in the DataFrame.
    It visualizes the distribution and frequency of categorical values, showing the count plot on the left
    and the bar chart on the right.

    Parameters:
    None: The function operates on the DataFrame attribute of the class.

    Returns:
    None: The function displays the count plots and bar charts for each categorical column.
    """


    categorical_data = self.df.select_dtypes(include=['object'])

    for column in categorical_data.columns:
        plt.figure(figsize=(12, 6))

        # Count plot
        plt.subplot(1, 2, 1)
        sns.countplot(data=categorical_data, x=column)
        plt.title(f'Count Plot for {column}', fontsize=14)
        plt.xticks(rotation=45, ha='right', fontsize=10)

        # Bar chart
        plt.subplot(1, 2, 2)
        categorical_data[column].value_counts().plot(kind='bar')
        plt.title(f'Bar Chart for {column}', fontsize=14)
        plt.xticks(rotation=45, ha='right', fontsize=10)

        plt.tight_layout()
        plt.show()
```

In [32]:
```
cleaner = DataCleaner(data)
imputer = DataImputer(data)
visualizer = DataVisualizer(data)
```

Removing Duplicates

Check for and remove duplicate rows to avoid biasing the analysis if exists.

In [35]:
```
cleaner.remove_duplicates()
```

```
Number of total duplicate rows: 0
No duplicate records found.
```

Out[35]:

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	...	Credit_Mix	Outstanding_Debt	Credit_Utilization_Ratio	Credit_History_Age	Payment_of_Min_Amount	Total_EMI_per_month	Amount_invested_monthly	Payment_Behaviou
0	0x1602	CUS_0xd40	January	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843	3	...	_	809.98	26.823	22 Years and 1 Months	No	49.575	80.41529543900253	High_spent_Small_value_payments
1	0x1603	CUS_0xd40	February	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	NaN	3	...	Good	809.98	31.945	NaN	No	49.575	118.28022162236736	Low_spent_Large_value_payments
2	0x1604	CUS_0xd40	March	Aaron Maashoh	-500	821-00-0265	Scientist	19114.12	NaN	3	...	Good	809.98	28.609	22 Years and 3 Months	No	49.575	81.699521264648	Low_spent_Medium_value_payments
3	0x1605	CUS_0xd40	April	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	NaN	3	...	Good	809.98	31.378	22 Years and 4 Months	No	49.575	199.4580743910713	Low_spent_Small_value_payments
4	0x1606	CUS_0xd40	May	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843	3	...	Good	809.98	24.797	22 Years and 5 Months	No	49.575	41.420153086217326	High_spent_Medium_value_payments
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	..
99995	0x25fe9	CUS_0x942c	April	Nicks	25	078-73-5990	Mechanic	39628.99	3359.416	4	...	_	502.38	34.664	31 Years and 6 Months	No	35.104	60.97133255718485	High_spent_Large_value_payments
99996	0x25fea	CUS_0x942c	May	Nicks	25	078-73-5990	Mechanic	39628.99	3359.416	4	...	_	502.38	40.566	31 Years and 7 Months	No	35.104	54.18595028760385	High_spent_Medium_value_payments
99997	0x25feb	CUS_0x942c	June	Nicks	25	078-73-5990	Mechanic	39628.99	3359.416	4	...	Good	502.38	41.256	31 Years and 8 Months	No	35.104	24.02847744864441	High_spent_Large_value_payments
99998	0x25fec	CUS_0x942c	July	Nicks	25	078-73-5990	Mechanic	39628.99	3359.416	4	...	Good	502.38	33.638	31 Years and 9 Months	No	35.104	251.67258219721603	Low_spent_Large_value_payments
99999	0x25fed	CUS_0x942c	August	Nicks	25	078-73-5990	Mechanic	39628.99_	3359.416	4	...	Good	502.38	34.192	31 Years and 10 Months	No	35.104	167.1638651610451	!@9#%8

100000 rows × 28 columns

Annual Income

In [41]:
```
cleaner.get_value_count('Annual_Income')
```

Out[41]:

	Annual_Income	counts	percent
0	36585.12	0.000	0.02%
1	20867.67	0.000	0.02%
2	17273.83	0.000	0.02%
3	9141.63	0.000	0.01%
4	33029.66	0.000	0.01%
...	...	...	...
18935	20269.93_	0.000	0.00%
18936	15157.25_	0.000	0.00%
18937	44955.64_	0.000	0.00%
18938	76650.12_	0.000	0.00%
18939	4262933.0	0.000	0.00%

18940 rows × 3 columns

The Annual_Income column represents the total income a customer earns in a year before any deductions, such as taxes or other withholdings. This value is a crucial indicator of the customer's overall financial stability and their ability to repay loans or manage other financial obligations. Currently, the Annual_Income column is of the object data type, which is inappropriate for numerical analysis. Therefore, it must be converted to a float data type to facilitate accurate numerical processing. Additionally, some values within this column contain extraneous underscores (_) either at the beginning or end. These non-numeric characters must be removed before conversion to ensure the integrity of the data and proper numerical computation.

Annual Income - Missing Values Treatment

In [45]:
```
cleaner.print_missing_values('Annual_Income')
cleaner.print_column_dtype('Annual_Income')
cleaner.find_non_numeric_values('Annual_Income')
```

```
Remaining missing values in Annual_Income: 0
dtype of Annual_Income: object
Found non numeric values:
```

Out[45]: {' ', '.', '_', '_ '}

In [47]:
```
cleaner.remove_non_numeric_characters('Annual_Income')
```

```
Non-numeric characters removed successfully from column 'Annual_Income'.
```

In [49]:
```
cleaner.convert_object_to_float('Annual_Income')
cleaner.get_value_count('Annual_Income')
```

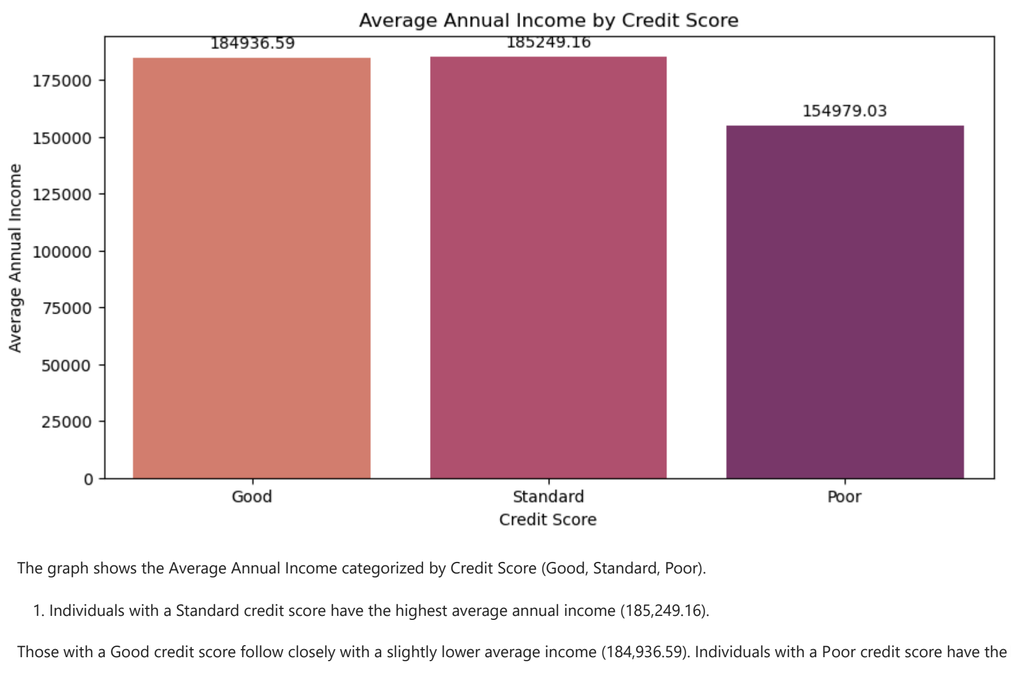
```
Column 'Annual_Income' successfully converted to float.
```

Out[49]:

	Annual_Income	counts	percent
0	17816.750	0.000	0.02%
1	22434.160	0.000	0.02%
2	40341.160	0.000	0.02%
3	17273.830	0.000	0.02%
4	109945.320	0.000	0.02%
...	...	...	...
13482	17079092.000	0.000	0.00%
13483	1910572.000	0.000	0.00%
13484	20179076.000	0.000	0.00%
13485	7980216.000	0.000	0.00%
13486	8299495.000	0.000	0.00%

13487 rows × 3 columns

In [51]:
```
visualizer.plot_average_by_column('Credit_Score', 'Annual_Income')
```



The graph shows the Average Annual Income categorized by Credit Score (Good, Standard, Poor).

1. Individuals with a Standard credit score have the highest average annual income (185,249.16).

Those with a Good credit score follow closely with a slightly lower average income (184,936.59). Individuals with a Poor credit score have the lowest average annual income (154,979.03).

```
Insights:
```

3. There is a noticeable gap in average income between individuals with a Poor credit score and those with Standard or Good credit scores.

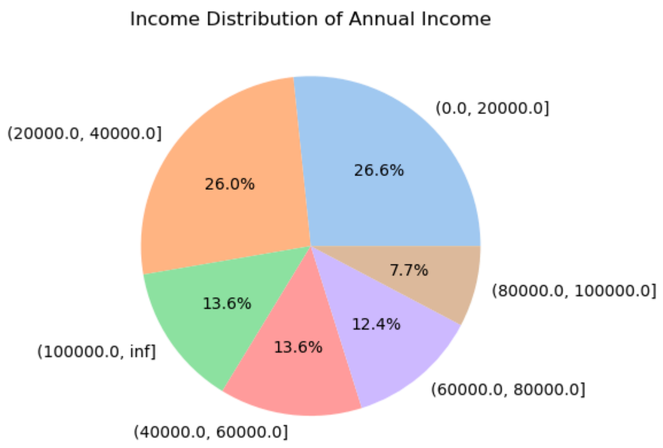
Despite having similar average incomes, individuals with Standard credit scores have a slightly higher income than those with Good scores. This may suggest that income is not the sole determinant of credit score and that other factors, such as payment behavior and credit utilization, play a significant role.

```
Implications:
```

5. Creditworthiness (as represented by the credit score) does not always correlate directly with higher income. Even individuals with substantial incomes may have poor credit scores if they fail to manage their finances effectively.

Targeting financial literacy programs for individuals with lower credit scores could help improve their financial management, regardless of income levels.

In [52]:
```
visualizer.plot_pie_chart_bin('Annual_Income', bins=[0, 20000, 40000, 60000, 80000, 100000, np.inf])
```



Key Observations:

1. The largest income group is those earning between 0 and 20,000 (26.6% of the population), indicating a significant portion of individuals at lower income levels.
2. The second-largest group is those earning between 20,000 and 40,000 (26.0%), showing a similar concentration in lower-middle income brackets.
3. The smallest proportion of individuals falls into the 80,000 to 100,000 income bracket (7.7%).
4. Higher-income groups such as 100,000 and above account for 13.6%, matching the percentage of individuals earning 40,000 to 60,000.

Insights:

1. Income Inequality: A substantial majority (around 52.6%) of individuals earn less than 40,000 annually, highlighting income inequality in the dataset.
2. Middle Bracket Stability: Middle-income brackets (40,000 to 80,000) collectively make up a significant portion of the population but are less prominent compared to the lowest income groups.
3. High-Income Concentration: The high-income category (100,000+) comprises only 13.6%, emphasizing that high earners are a minority.

Implications:

1. Financial Inclusion: Financial products should be tailored to cater to the majority low-income group, focusing on affordability and accessibility.
2. Targeted Strategies: Middle- and high-income earners may require specialized investment and credit products that align with their financial capabilities.
3. Economic Policy: The distribution suggests potential areas for policy intervention to address economic disparities.

Monthly Inhand Salary

In [57]:
```
cleaner.get_value_count('Monthly_Inhand_Salary')
```

Out[57]:

	Monthly_Inhand_Salary	counts	percent
0	6769.130	0.000	0.02%
1	6358.957	0.000	0.02%
2	2295.058	0.000	0.02%
3	6082.188	0.000	0.02%
4	3080.555	0.000	0.02%
...	...	...	...
13230	1087.546	0.000	0.00%
13231	3189.212	0.000	0.00%
13232	5640.118	0.000	0.00%
13233	7727.560	0.000	0.00%
13234	2443.654	0.000	0.00%

13235 rows × 3 columns

The Monthly_Inhand_Salary column represents the net salary a customer receives after deductions such as taxes and other withholdings. This value is crucial as it indicates the disposable income a customer has on a monthly basis, which in turn reflects their financial capacity and stability.

In the absence of specific information regarding deductions or taxes within the dataset, a pragmatic approach will be employed to impute missing values in the Monthly_Inhand_Salary column. Specifically, missing values will be calculated by dividing the Annual_Income by 12. This method ensures the consistency and completeness of the monthly salary data across all records, thereby enhancing the model's ability to make accurate predictions by working with a comprehensive and filled dataset.

Monthly Inhand Salary - Missing Values Treatment

In [59]:
```
cleaner.print_missing_values('Monthly_Inhand_Salary')
cleaner.print_column_dtype('Monthly_Inhand_Salary')
cleaner.find_non_numeric_values('Monthly_Inhand_Salary')
```

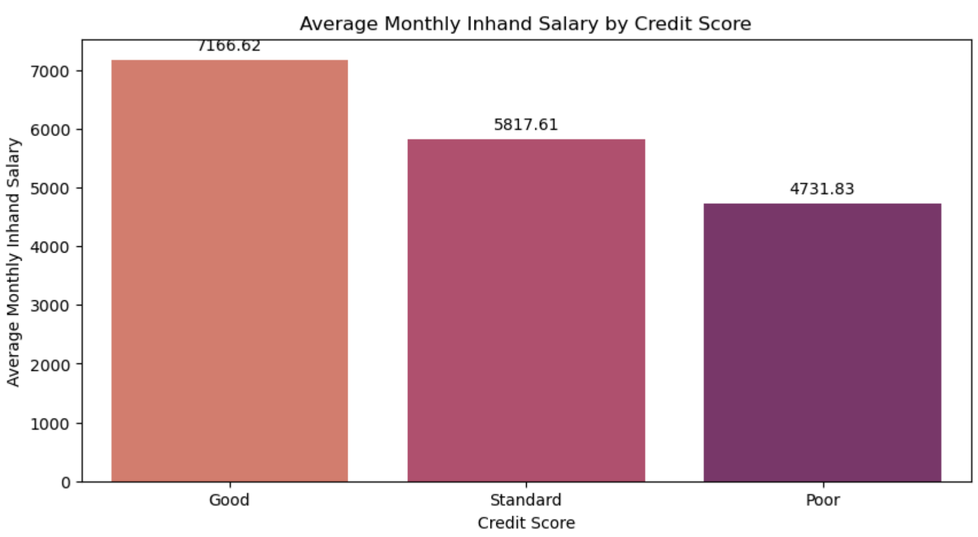
```
Remaining missing values in Monthly_Inhand_Salary: 15002
dtype of Monthly_Inhand_Salary: float64
Found non numeric values:
```

Out[59]:
```
{' ',
 ' nan ',
 ' nan nan ',
 ' nan nan nan ',
 ' nan nan nan nan ',
 ' nan nan nan nan nan ',
 '.'}
```

In [61]:
```
imputer.fill_missing_monthly_salary('Annual_Income','Monthly_Inhand_Salary')
```

```
Missing values in column 'Monthly_Inhand_Salary' filled successfully by dividing 'Annual_Income' by 12.
```

In [63]:
```
visualizer.plot_average_by_column('Credit_Score', 'Monthly_Inhand_Salary')
```



Type of Loan

The Type_of_Loan column encapsulates the specific types of loans a customer has acquired, including various categories such as Personal Loan, Auto Loan, and Home Equity Loan. This information is instrumental in categorizing the different loan products a customer may possess, thereby providing insights into their financial behavior and preferences.

In the data preprocessing phase, the Type_of_Loan column underwent a transformation to delineate multiple loan types into individual binary columns. Each new column represents the presence (1) or absence (0) of a specific loan type for a given entry. This binarization enhances the model's capability to distinguish between different loan types effectively.

Additionally, Boolean values (True/False) were converted to binary (0/1) to maintain consistency and facilitate numerical processing. The original Type_of_Loan column was subsequently removed to streamline the dataset, ensuring that the transformed data remains clear and interpretable for further analysis and modeling.

In [67]:
```
cleaner.get_value_count('Type_of_Loan')
```

Out[67]:

	Type_of_Loan	counts	percent
0	Not Specified	0.016	1.59%
1	Credit-Builder Loan	0.014	1.44%
2	Personal Loan	0.014	1.44%
3	Debt Consolidation Loan	0.014	1.43%
4	Student Loan	0.014	1.40%
...	...	...	...
6255	Not Specified, Mortgage Loan, Auto Loan, and P...	0.000	0.01%
6256	Payday Loan, Mortgage Loan, Debt Consolidation...	0.000	0.01%
6257	Debt Consolidation Loan, Auto Loan, Personal L...	0.000	0.01%
6258	Student Loan, Auto Loan, Student Loan, Credit-...	0.000	0.01%
6259	Personal Loan, Auto Loan, Mortgage Loan, Stude...	0.000	0.01%

6260 rows × 3 columns

Type of Loan - Missing Values Treatment

In [69]:
```
cleaner.print_missing_values('Type_of_Loan')
```

```
Remaining missing values in Type_of_Loan: 11408
```

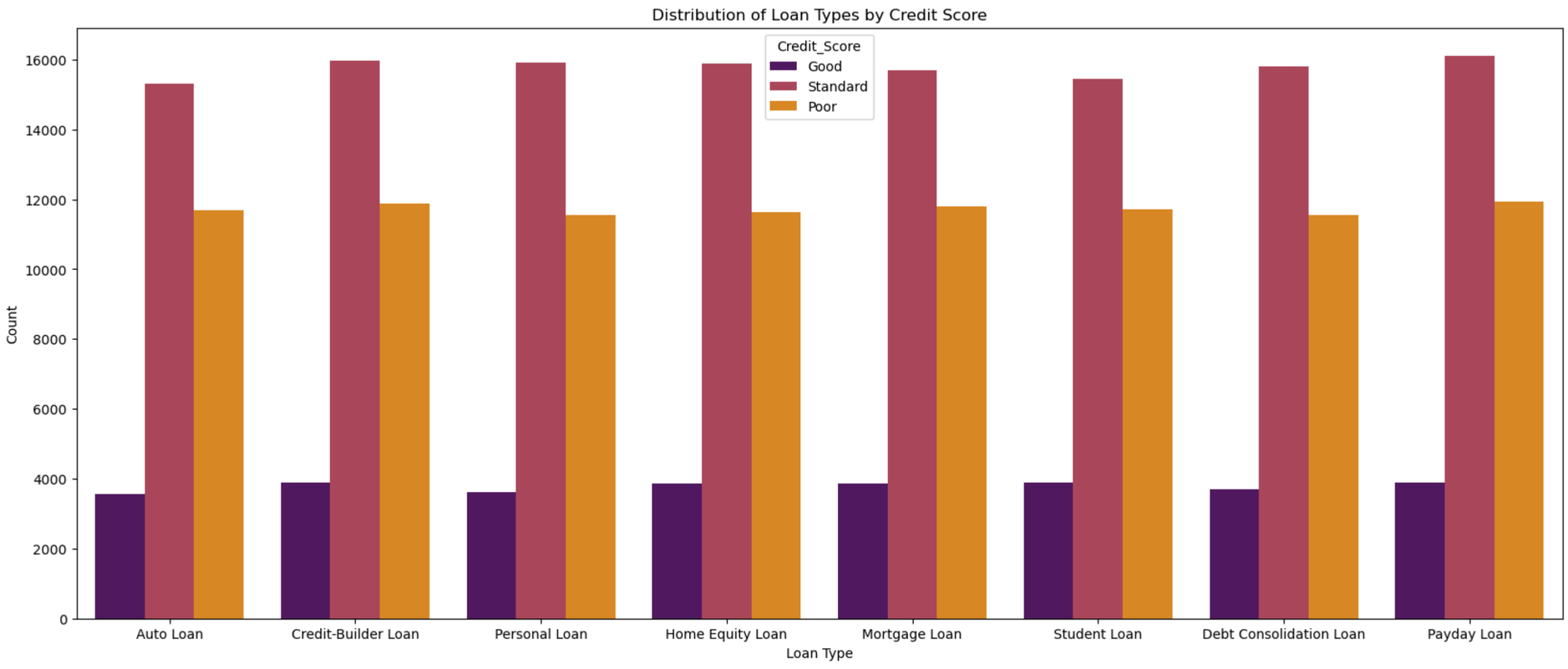
In [71]:
```
cleaner.transform_loan_types('Type_of_Loan')
```

Out[71]:

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	...	Credit_Score	Auto Loan	Credit-Builder Loan	Personal Loan	Home Equity Loan	Not Specified	Mortgage Loan	Student Loan	Debt Consolidation Loan	Payday Loan
0	0x1602	CUS_0xd40	January	Aaron Maashoh	23	821-00-0265	Scientist	19114.120	1824.843	3	...	Good	1	1	1	1	0	0	0	0	0
1	0x1603	CUS_0xd40	February	Aaron Maashoh	23	821-00-0265	Scientist	19114.120	1592.843	3	...	Good	1	1	1	1	0	0	0	0	0
2	0x1604	CUS_0xd40	March	Aaron Maashoh	-500	821-00-0265	Scientist	19114.120	1592.843	3	...	Good	1	1	1	1	0	0	0	0	0
3	0x1605	CUS_0xd40	April	Aaron Maashoh	23	821-00-0265	Scientist	19114.120	1592.843	3	...	Good	1	1	1	1	0	0	0	0	0
4	0x1606	CUS_0xd40	May	Aaron Maashoh	23	821-00-0265	Scientist	19114.120	1824.843	3	...	Good	1	1	1	1	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
99995	0x25fe9	CUS_0x942c	April	Nicks	25	078-73-5990	Mechanic	39628.990	3359.416	4	...	Poor	1	0	0	0	0	0	1	0	0
99996	0x25fea	CUS_0x942c	May	Nicks	25	078-73-5990	Mechanic	39628.990	3359.416	4	...	Poor	1	0	0	0	0	0	1	0	0
99997	0x25feb	CUS_0x942c	June	Nicks	25	078-73-5990	Mechanic	39628.990	3359.416	4	...	Poor	1	0	0	0	0	0	1	0	0
99998	0x25fec	CUS_0x942c	July	Nicks	25	078-73-5990	Mechanic	39628.990	3359.416	4	...	Standard	1	0	0	0	0	0	1	0	0
99999	0x25fed	CUS_0x942c	August	Nicks	25	078-73-5990	Mechanic	39628.990	3359.416	4	...	Poor	1	0	0	0	0	0	1	0	0

100000 rows × 36 columns

In [72]:
```
visualizer.plot_loan_distribution_by_credit_score()
```



Num of Delayed Payment

The Num_of_Delayed_Payment column quantifies the number of instances where a customer has delayed payments beyond the stipulated due date. This metric encompasses late payments on various financial obligations, including loans, credit cards, and other liabilities. It serves as a critical feature in credit scoring models, given that a higher frequency of delayed payments typically correlates with elevated risk of default and financial instability.

Anomalies:

A value of -1 is indicative of erroneous or placeholder data. Since delayed payments cannot logically assume a negative value, such entries likely represent missing or invalid data.

NaN values signify missing data points that require attention during data preprocessing. These can be addressed through imputation methods (e.g., using the median or mode) or retained as NaN if appropriate for the analysis.

Non-numeric anomalies, including characters that do not represent digits, should also be regarded as errors and cleansed accordingly.

Furthermore, the data type of this column is currently object, which is unsuitable for numerical analysis. Post-cleaning, the data type should be converted to a numerical format, such as int or float, to enable accurate computation and analysis.

In [76]:
```
cleaner.get_value_count('Num_of_Delayed_Payment')
```

Out[76]:

	Num_of_Delayed_Payment	counts	percent
0	19	0.057	5.73%
1	17	0.057	5.66%
2	16	0.056	5.56%
3	10	0.055	5.54%
4	18	0.055	5.47%
...	...	...	...
744	848_	0.000	0.00%
745	4134	0.000	0.00%
746	1530	0.000	0.00%
747	1502	0.000	0.00%
748	2047	0.000	0.00%

749 rows × 3 columns

Num of Delayed Payment - Missing Values Treatment

In [77]:
```
cleaner.print_missing_values('Num_of_Delayed_Payment')
cleaner.print_column_dtype('Num_of_Delayed_Payment')
cleaner.find_non_numeric_values('Num_of_Delayed_Payment')
```

```
Remaining missing values in Num_of_Delayed_Payment: 7002
dtype of Num_of_Delayed_Payment: object
Found non numeric values:
```

Out[77]:
```
{' ',
 ' -',
 ' nan ',
 ' nan -',
 ' nan nan ',
 ' nan nan -',
 ' nan nan nan ',
 ' nan nan nan -',
 ' nan nan nan nan ',
 ' nan nan nan nan nan ',
 '_ ',
 '_ -',
 '_ nan ',
 '_ nan -',
 '_ nan nan '}
```

In [80]:
```
cleaner.remove_non_numeric_characters('Num_of_Delayed_Payment')
```

```
Non-numeric characters removed successfully from column 'Num_of_Delayed_Payment'.
```

In [83]:
```
cleaner.replace_empty_with_nan('Num_of_Delayed_Payment')
```

```
Empty strings in column 'Num_of_Delayed_Payment' replaced with NaN.
```

In [85]:
```
cleaner.convert_object_to_float('Num_of_Delayed_Payment')
```

```
Column 'Num_of_Delayed_Payment' successfully converted to float.
```

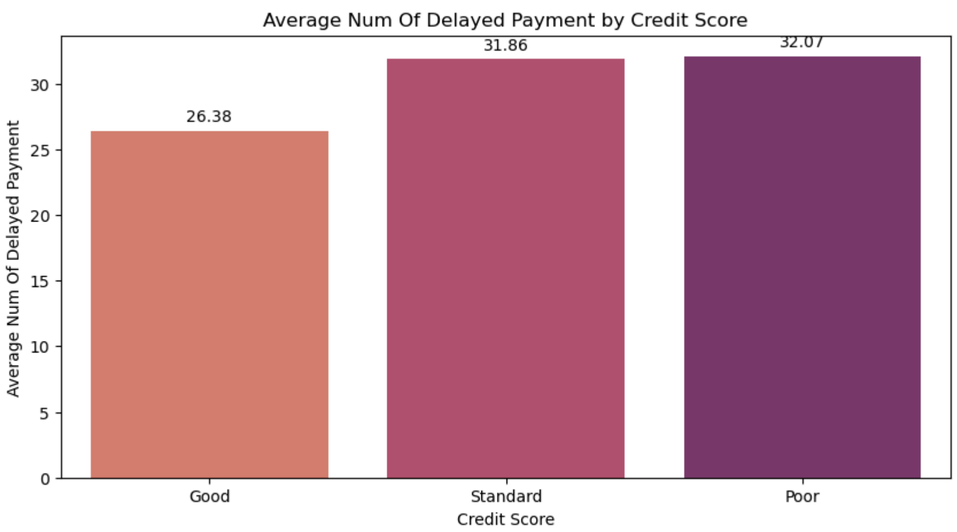
In [87]:
```
cleaner.convert_to_absolute('Num_of_Delayed_Payment')
```

```
Values in column 'Num_of_Delayed_Payment' have been converted to their absolute values.
```

In [89]:
```
imputer.knn_impute_column('Num_of_Delayed_Payment')
```

```
Missing values in column 'Num_of_Delayed_Payment' have been imputed using KNN with 5 neighbors.
```

In [90]:
```
visualizer.plot_average_by_column('Credit_Score', 'Num_of_Delayed_Payment')
```



Num Credit Inquiries

The Num_Credit_Inquiries column quantifies the number of credit inquiries recorded on a customer's credit report. Such inquiries typically transpire when a customer applies for new credit products, including loans or credit cards, prompting lenders to review the customer's credit history. A higher frequency of credit inquiries may be indicative of greater financial instability, as it often suggests that the customer is actively seeking multiple credit sources.

Initially, this column included anomalous entries such as ' ', ' nan ', ' nan nan ', and ' nan nan nan ', which were interpreted as missing data. To address these gaps, the K-Nearest Neighbors (KNN) imputation method was employed. This method imputes missing values by referencing the values of the nearest neighbors, thereby ensuring the integrity and consistency of the dataset for subsequent analysis.

In [93]:
```
cleaner.get_value_count('Num_Credit_Inquiries')
```

Out[93]:

	Num_Credit_Inquiries	counts	percent
0	4.000	0.115	11.50%
1	3.000	0.091	9.07%
2	6.000	0.083	8.27%
3	7.000	0.082	8.22%
4	2.000	0.082	8.19%
...	...	...	...
1218	1721.000	0.000	0.00%
1219	1750.000	0.000	0.00%
1220	2397.000	0.000	0.00%
1221	621.000	0.000	0.00%
1222	74.000	0.000	0.00%

1223 rows × 3 columns

Num Credit Inquiries - Missing Values Treatment

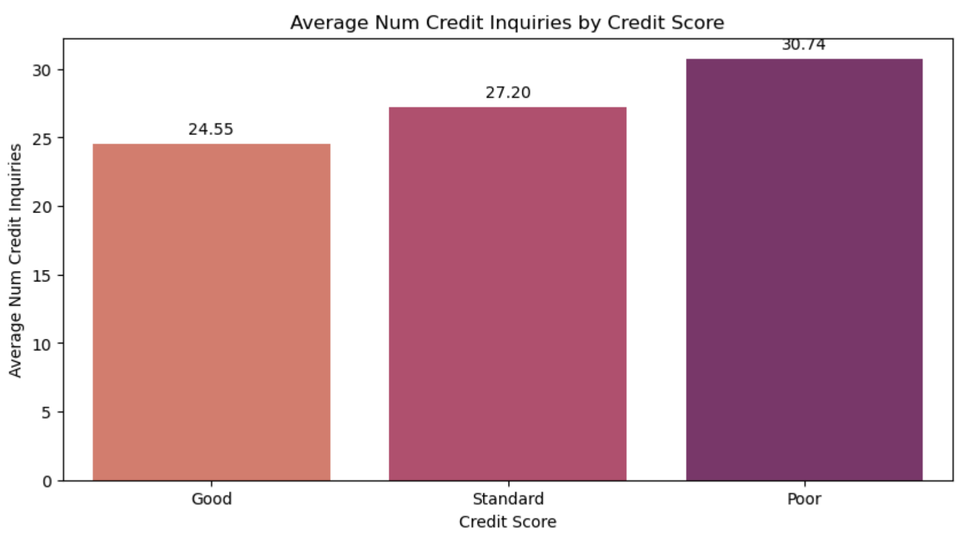
In [94]:
```
cleaner.print_missing_values('Num_Credit_Inquiries')
cleaner.print_column_dtype('Num_Credit_Inquiries')
```

```
Remaining missing values in Num_Credit_Inquiries: 1965
dtype of Num_Credit_Inquiries: float64
```

In [95]:
```
imputer.knn_impute_column('Num_Credit_Inquiries')
```

```
Missing values in column 'Num_Credit_Inquiries' have been imputed using KNN with 5 neighbors.
```

In [96]:
```
visualizer.plot_average_by_column('Credit_Score', 'Num_Credit_Inquiries')
```



An analysis of credit inquiry data reveals a notable trend: customers classified with a Poor credit score exhibit the highest average number of credit inquiries, with an average of 30.74 inquiries. Conversely, those with a Good credit score present the lowest average, at 24.55 inquiries. This pattern suggests that individuals with poorer credit scores are engaging in credit-seeking behavior more frequently than their counterparts with higher credit ratings. Such a trend may be indicative of underlying financial challenges or instability, prompting these individuals to apply for credit more often in an attempt to manage their financial obligations.

Credit History Age

In [100…
```
cleaner.get_value_count('Credit_History_Age')
```

Out[100…

	Credit_History_Age	counts	percent
0	15 Years and 11 Months	0.005	0.49%
1	19 Years and 4 Months	0.005	0.49%
2	19 Years and 5 Months	0.005	0.49%
3	17 Years and 11 Months	0.005	0.49%
4	19 Years and 3 Months	0.005	0.48%
...	...	...	...
399	0 Years and 3 Months	0.000	0.02%
400	0 Years and 2 Months	0.000	0.02%
401	33 Years and 7 Months	0.000	0.02%
402	33 Years and 8 Months	0.000	0.01%
403	0 Years and 1 Months	0.000	0.00%

404 rows × 3 columns

The Credit_History_Age column represents the overall duration of a customer's credit history, initially provided in a string format such as "X Years and Y Months." To facilitate standardization and subsequent analysis, a custom function was implemented to convert the credit history from this string format into a total number of months. This transformation enables a more streamlined and uniform analysis of the data.

No anomalous values were identified within this column aside from the presence of missing data. The missing values were addressed using the K-Nearest Neighbors (KNN) imputation method, ensuring the dataset's consistency and integrity. This technique is widely recognized for its ability to preserve data structure and relationships. As Batista and Monard (2003) state, "The KNN imputation method is particularly effective for maintaining the consistency and predictive power of datasets, as it relies on the similarity of observations to estimate missing values." This ensured the dataset remained robust for subsequent analytical procedures.

Credit History Age - Missing Values Treatment

In [101…
```
cleaner.print_missing_values('Credit_History_Age')
cleaner.print_column_dtype('Credit_History_Age')
```

```
Remaining missing values in Credit_History_Age: 9030
dtype of Credit_History_Age: object
```

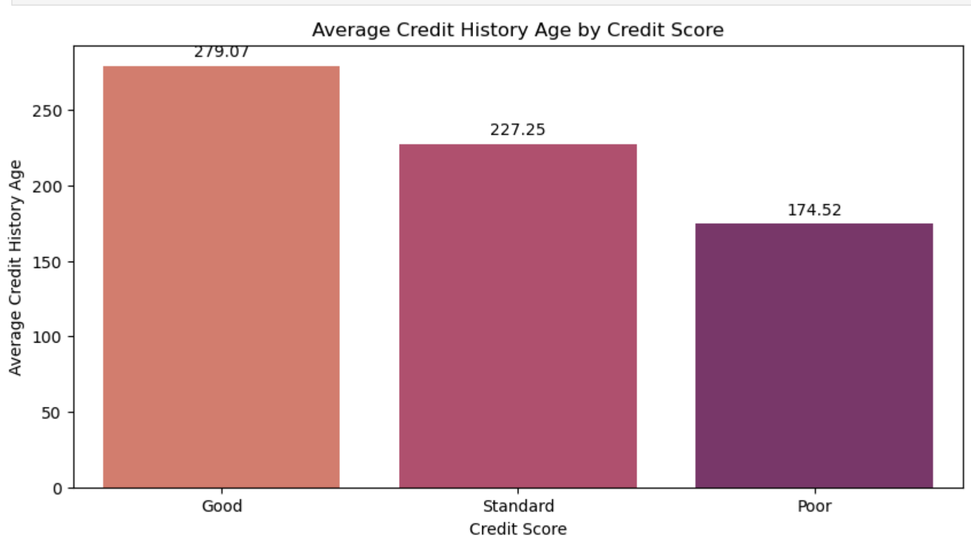
In [102…
```
cleaner.convert_credit_history_age('Credit_History_Age')
```

```
Column 'Credit_History_Age' has been converted to total months.
```

In [103…
```
imputer.knn_impute_column('Credit_History_Age')
```

```
Missing values in column 'Credit_History_Age' have been imputed using KNN with 5 neighbors.
```

In [104…
```
visualizer.plot_average_by_column('Credit_Score', 'Credit_History_Age')
```



An analysis of credit history data reveals a significant correlation between the length of credit history and credit score. Specifically, customers with a Good Credit Score have an average credit history age of 279 months, which is substantially longer compared to those with a Standard Credit Score (227 months) or a Poor Credit Score (174 months). This observation suggests that a more extended credit history is potentially associated with a higher credit score, indicating that prolonged and consistent credit activity may contribute positively to a customer's creditworthiness.

Amount Invested Monthly

The Amount_invested_monthly column, which represents the monthly investment amount, initially contained non-numeric characters and missing values. To address this, the data was first cleaned to remove non-numeric characters and subsequently converted to a float data type. Missing values within this column were then imputed using the K-Nearest Neighbors (KNN) imputation method, ensuring the data's consistency and reliability for further analysis.

In [108…
```
cleaner.get_value_count('Amount_invested_monthly')
```

Out[108…

	Amount_invested_monthly	counts	percent
0	__10000__	0.045	4.51%
1	0.0	0.002	0.18%
2	80.41529543900253	0.000	0.00%
3	36.66235139442514	0.000	0.00%
4	89.7384893604547	0.000	0.00%
...	...	...	...
91044	36.541908593249026	0.000	0.00%
91045	93.45116318631192	0.000	0.00%
91046	140.80972223052834	0.000	0.00%
91047	38.73937670100975	0.000	0.00%
91048	167.1638651610451	0.000	0.00%

91049 rows × 3 columns

In [109…
```
cleaner.print_missing_values('Amount_invested_monthly')
cleaner.print_column_dtype('Amount_invested_monthly')
cleaner.find_non_numeric_values('Amount_invested_monthly')
```

```
Remaining missing values in Amount_invested_monthly: 4479
dtype of Amount_invested_monthly: object
Found non numeric values:
```

Out[109…
```
{' ',
 ' __',
 ' nan ',
 ' nan __',
 ' nan nan ',
 ' nan nan __',
 ' nan nan nan ',
 ' nan nan nan nan ',
 '.',
 '__ ',
 '__ __',
 '__ nan ',
 '__ nan __',
 '__ nan nan ',
 '__ nan nan __'}
```

Average Amount Invested Monthly by Credit Score

In [110…
```
cleaner.remove_non_numeric_characters('Amount_invested_monthly')
```

```
Non-numeric characters removed successfully from column 'Amount_invested_monthly'.
```

In [111…
```
cleaner.replace_empty_with_nan('Amount_invested_monthly')
```

```
Empty strings in column 'Amount_invested_monthly' replaced with NaN.
```

In [112…
```
cleaner.convert_object_to_float('Amount_invested_monthly')
```

```
Column 'Amount_invested_monthly' successfully converted to float.
```

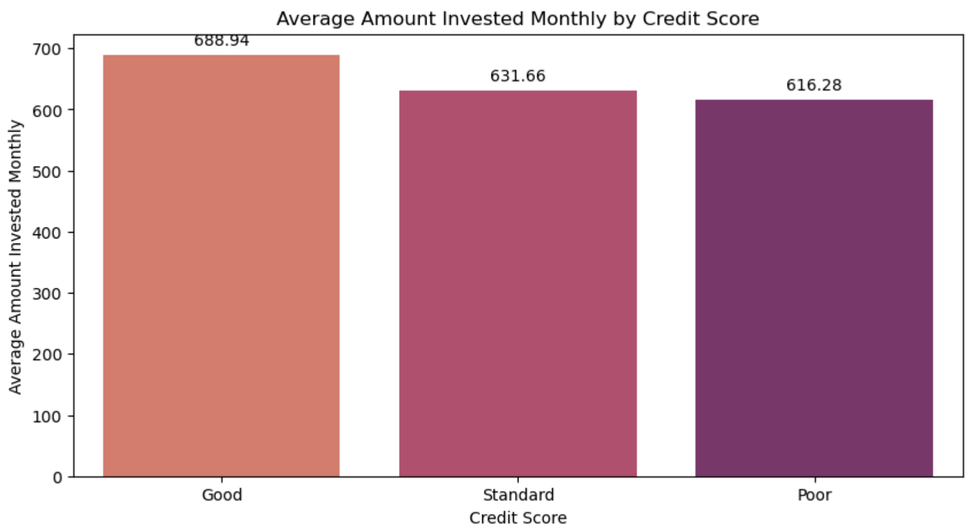
In [113…
```
cleaner.convert_to_absolute('Amount_invested_monthly')
```

```
Values in column 'Amount_invested_monthly' have been converted to their absolute values.
```

In [114…
```
imputer.knn_impute_column('Amount_invested_monthly')
```

```
Missing values in column 'Amount_invested_monthly' have been imputed using KNN with 5 neighbors.
```

In [115…
```
visualizer.plot_average_by_column('Credit_Score', 'Amount_invested_monthly')
```



The analysis of monthly investment amounts indicates a clear trend among individuals with varying credit scores. Specifically, individuals with a Good credit score exhibit the highest average monthly investment, amounting to 688.94 units. In comparison, those with Standard credit scores invest slightly less, averaging 631.66 units per month, while individuals with Poor credit scores invest the least, with an average of 616.28 units. This pattern suggests a potential positive correlation between higher monthly investment amounts and better credit scores, implying that individuals with better credit scores are more likely to invest larger sums on a monthly basis.

Monthly Balance

In [119…
```
cleaner.get_value_count('Monthly_Balance')
```

Out[119…

	Monthly_Balance	counts	percent
0	__-333333333333333333333333333__	0.000	0.01%
1	312.49408867943663	0.000	0.00%
2	415.32532309844316	0.000	0.00%
3	252.08489793906085	0.000	0.00%
4	254.9709216273975	0.000	0.00%
...	...	...	...
98787	366.2890379762706	0.000	0.00%
98788	151.1882696261166	0.000	0.00%
98789	306.75027851710234	0.000	0.00%
98790	278.8720257394474	0.000	0.00%
98791	393.674	0.000	0.00%

98792 rows × 3 columns

The Monthly_Balance column represents the remaining balance in a customer's account at the end of each month after all expenses and income have been accounted for. This metric provides valuable insights into a customer's financial stability and management practices.

The column was initially cleaned by removing anomalous values, such as excessively large or erroneous entries, and these were replaced with NaN to facilitate appropriate handling. Missing values were subsequently addressed by grouping the data based on relevant financial factors, such as the Delay_from_due_date, and imputing the missing values using the group's average balance. This approach ensured that the Monthly_Balance data remained consistent and accurately reflective of actual customer financial behavior.

Correcting Outlier Values - Replaced with NaN To facilitate Subsequent Data Processing

The value 333333333333333.0 observed in the Monthly_Balance column is anomalously large and evidently erroneous when compared to other entries in the dataset. To facilitate proper handling in subsequent data processing steps, this value has been replaced with NaN. This ensures that further imputation or analysis techniques can appropriately address and correct these outlier values, thereby maintaining the integrity and consistency of the dataset.

In [121…
```
cleaner.replace_unusual_values('Monthly_Balance','__-333333333333333333333333333__')
```

```
Unusual value '__-333333333333333333333333333__' in column 'Monthly_Balance' has been replaced with NaN.
```

In [122…
```
cleaner.print_missing_values('Monthly_Balance')
cleaner.print_column_dtype('Monthly_Balance')
cleaner.find_non_numeric_values('Monthly_Balance')
```

```
Remaining missing values in Monthly_Balance: 1209
dtype of Monthly_Balance: object
Found non numeric values:
```

Out[122…
```
{' ', ' nan ', ' nan nan ', ' nan nan nan ', ' nan nan nan nan ', '.'}
```

In [123…
```
cleaner.remove_non_numeric_characters('Monthly_Balance')
cleaner.convert_hex_to_int('Monthly_Balance')
```

```
Non-numeric characters removed successfully from column 'Monthly_Balance'.
Hexadecimal values in column 'Monthly_Balance' have been converted to integers.
```

In [124…
```
cleaner.replace_empty_with_nan('Monthly_Balance')
```

```
Empty strings in column 'Monthly_Balance' replaced with NaN.
```

In [125…
```
cleaner.convert_object_to_float('Monthly_Balance')
```

```
Column 'Monthly_Balance' successfully converted to float.
```

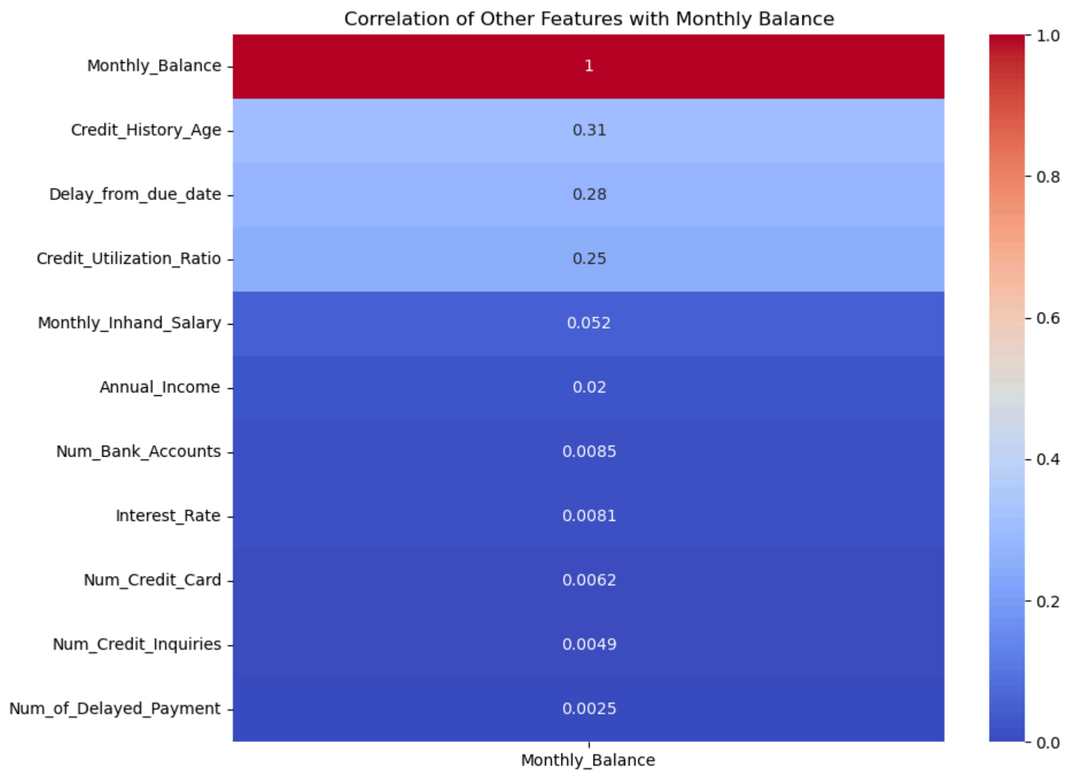
The Monthly_Balance column reflects the residual funds in a customer's account at the end of each month after accounting for all income and expenses. This metric serves as an indicator of a customer's financial stability and cash flow management.

The Delay_from_due_date column denotes the number of days a customer delayed payment beyond the due date. There is a plausible relationship between Delay_from_due_date and Monthly_Balance, as habitual delays in payment may indicate financial distress or suboptimal cash flow management, potentially leading to a lower monthly balance.

Given this potential correlation, it is reasonable to hypothesize that customers who consistently delay payments may have lower end-of-month balances. To explore this relationship and enhance the accuracy of our dataset, we can group the data by Delay_from_due_date to identify patterns. This grouping can then be leveraged to impute missing values in the Monthly_Balance column using the average balance within each group. This method ensures that the imputation process maintains the logical consistency and integrity of the dataset, providing a more accurate reflection of customer financial behavior.

Correlation Of Other features with Monthly Ballance

In [127…
```
visualizer.plot_correlation_with_columns('Monthly_Balance', data.select_dtypes(include=['number']).columns.to_list()[:10])
```



Observations:

- Highest Correlation: Credit_History_Age (0.31): There is a moderate positive correlation between the length of credit history and monthly balance. This suggests that individuals with a longer credit history tend to have higher monthly balances, likely due to more experience managing credit and finances effectively. Delay_from_due_date (0.28): A moderate positive correlation indicates that delays in payments are somewhat linked to higher monthly balances. This may imply that individuals who delay payments could be retaining more liquidity or avoiding financial obligations.
- Moderate Correlation: Credit_Utilization_Ratio (0.25): The correlation shows that higher credit utilization is moderately associated with higher monthly balances. This could indicate that individuals with higher spending tend to leave more liquidity available.
- Low to Negligible Correlation: Monthly_Inhand_Salary (0.052): The correlation between in-hand salary and monthly balance is weak, suggesting that salary levels have limited direct influence on monthly balances.
- Annual_Income (0.02): Annual income shows almost no correlation with monthly balance, indicating that income alone does not determine how much balance an individual retains monthly.
- Other Features such as Num_Bank_Accounts, Interest_Rate, Num_Credit_Card, Num_Credit_Inquiries, Num_of_Delayed_Payment, have correlations near zero, implying little to no linear relationship with monthly balance.

Insights:

- Behavioral and Historical Factors Matter Most:

Credit history age and delayed payments have the strongest impact on monthly balance compared to salary or income, highlighting the importance of behavioral factors over financial capability.

- Wealth Does Not Always Equal Balance:

Features like annual income and in-hand salary, which reflect financial capability, show weak correlations with monthly balance. This indicates that how individuals manage their finances (e.g., credit utilization, payment behavior) is more relevant than how much they earn.

- Low Influence of Financial Products:

Features like the number of credit cards, credit inquiries, and bank accounts have negligible impact on monthly balances, indicating they are not primary drivers of retained balances.

Implications:

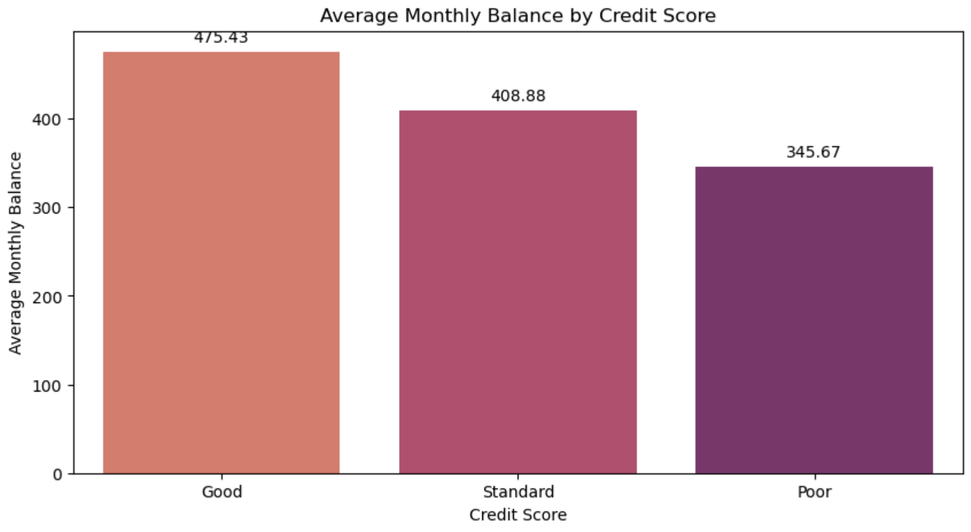
- Focus on Financial Behavior: Strategies to improve monthly balance should focus on improving credit utilization and reducing delayed payments rather than solely targeting income growth.
- Credit History Management: Encouraging users to build a strong credit history can positively influence their financial stability.
- Low Impact Features: Features with very low correlations may not significantly impact predictions related to monthly balances and could potentially be deprioritized in modeling efforts.

Average Monthly Balance by Credit Score

In [128…
```
imputer.impute_missing_values('Monthly_Balance','Delay_from_due_date')
```

```
Missing values in column 'Monthly_Balance' have been imputed using the mean of groups based on 'Delay_from_due_date'.
```

In [129…
```
visualizer.plot_average_by_column('Credit_Score', 'Monthly_Balance')
```



The analysis indicates a notable trend in financial behavior based on credit scores. Customers with a Good credit score exhibit a significantly higher average monthly balance, maintaining an average of approximately 475 units. In contrast, customers with Standard or Poor credit scores exhibit lower average balances, with those having poor credit averaging around 345 units. This pattern suggests a potential correlation between higher credit scores and superior financial management practices, resulting in higher retained balances. The findings imply that individuals with better credit scores are likely to demonstrate more effective financial management, leading to greater financial stability.

Month Distribution by Credit Score

In [132…
```
cleaner.get_value_count('Month')
```

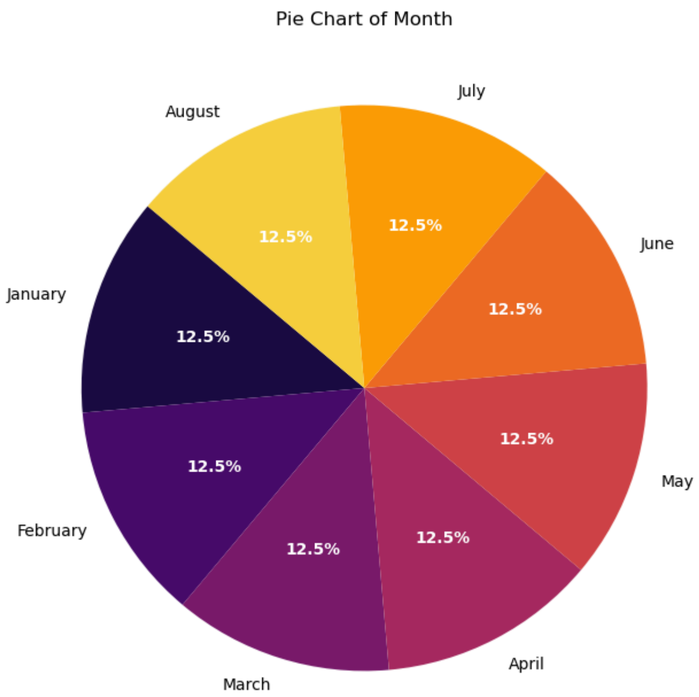
Out[132…

	Month	counts	percent
0	January	0.125	12.50%
1	February	0.125	12.50%
2	March	0.125	12.50%
3	April	0.125	12.50%
4	May	0.125	12.50%
5	June	0.125	12.50%
6	July	0.125	12.50%
7	August	0.125	12.50%

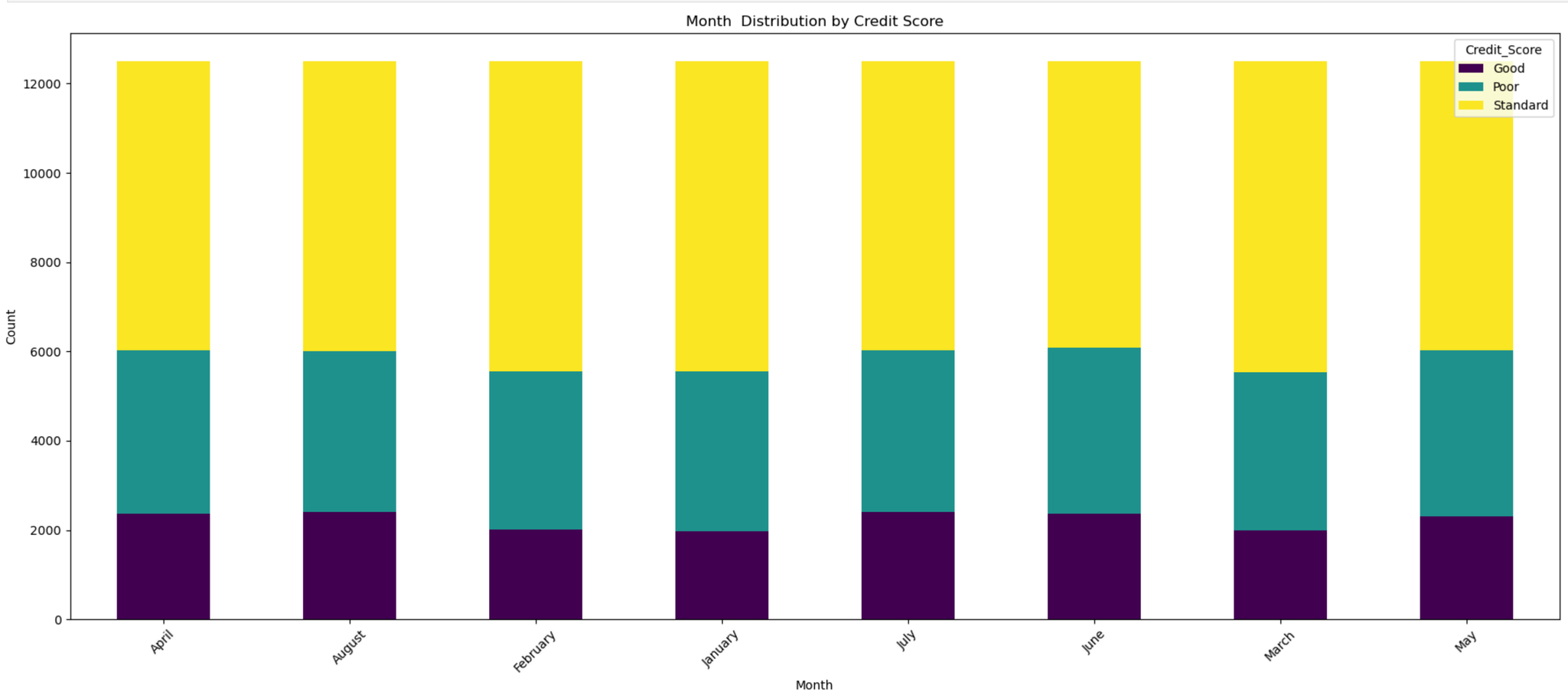
In [133…
```
cleaner.print_missing_values('Month')
cleaner.print_column_dtype('Month')
```

```
Remaining missing values in Month: 0
dtype of Month: object
```

In [134…
```
visualizer.plot_pie_chart('Month')
```



In [135…
```
visualizer.plot_month_distribution_by_credit_score('Month','Credit_Score')
```



Age Distribution

The Age column provides critical demographic information, which can significantly influence financial behavior and creditworthiness.

Although there is no universally fixed age cutoff, lending institutions generally become more stringent with borrowers aged 65-70, especially for long-term loans.

For this dataset, which focuses on financial activities such as loans, credit, and banking services, it is reasonable to set an age limit of 65-70 years. This aligns with industry practices where financial institutions tend to exercise increased caution with individuals approaching or exceeding typical retirement age.

Additionally, age values below 18 are considered unrealistic for most loan types, as borrowing typically begins after reaching legal adulthood.

To preserve data integrity, age values outside these reasonable limits were replaced with NaN rather than being discarded. Subsequently, the data was grouped by Monthly_Balance, Credit_History_Age, and Delay_from_due_date, and the missing age values were imputed using the average age within each group. These columns were chosen based on the rationale that individuals with similar income, credit history, and payment behavior are likely to exhibit comparable age characteristcs.

In [138…
```
cleaner.get_value_count('Age')
```

Out[138…

	Age	counts	percent
0	38	0.028	2.83%
1	28	0.028	2.83%
2	31	0.028	2.81%
3	26	0.028	2.79%
4	32	0.027	2.75%
...	...	...	...
1783	471	0.000	0.00%
1784	1520	0.000	0.00%
1785	8663	0.000	0.00%
1786	3363	0.000	0.00%
1787	1342	0.000	0.00%

1788 rows × 3 columns

Age Distribution - Missing Values treatment

In [139…
```
cleaner.print_missing_values('Age')
cleaner.print_column_dtype('Age')
cleaner.find_non_numeric_values('Age')
```

```
Remaining missing values in Age: 0
dtype of Age: object
Found non numeric values:
```

Out[139… {' ', ' -', '_ ', '_ -'}

Conversion of Age to Numeric Character

In [141…
```
cleaner.remove_non_numeric_characters('Age')
```

```
Non-numeric characters removed successfully from column 'Age'.
```

In [143…
```
cleaner.convert_object_to_float('Age')
```

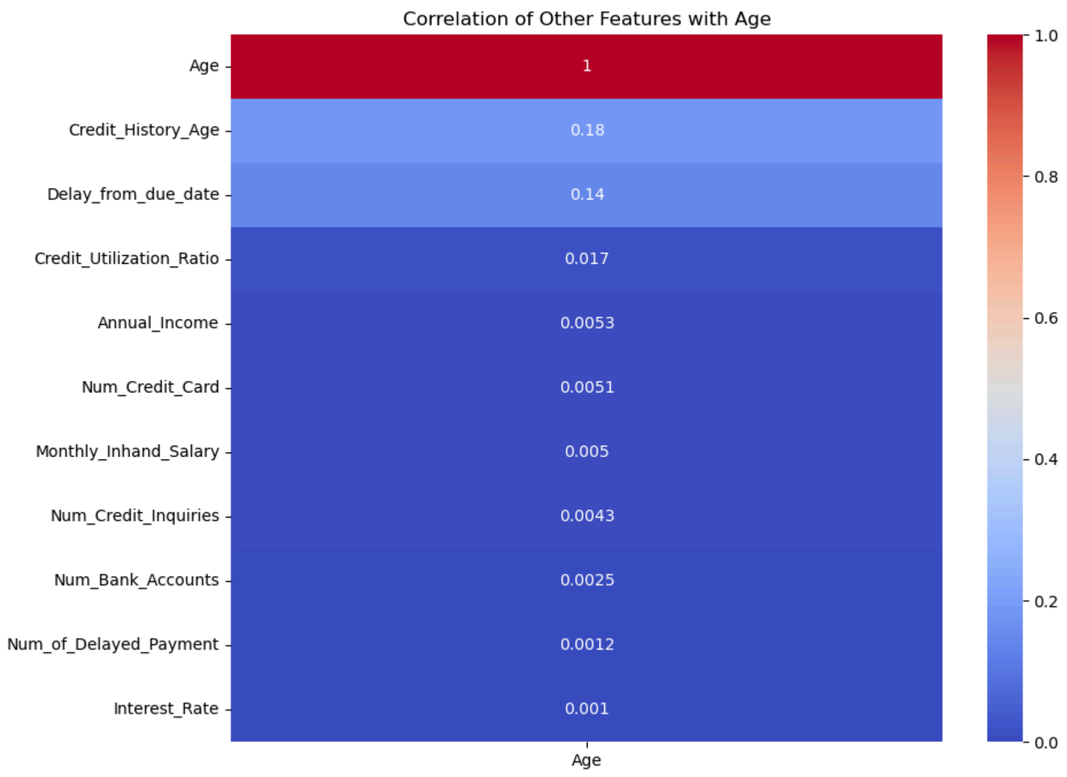
```
Column 'Age' successfully converted to float.
```

In [146…
```
cleaner.cap_age_values('Age')
```

```
Values in column 'Age' outside the range 18-120 have been replaced with NaN.
```

Correlation of Other Feature with Age

In [147…
```
visualizer.plot_correlation_with_columns('Age', data.select_dtypes(include=['number']).columns.to_list()[1:11])
```



Key Observations:

- Strongest Correlation:

Credit_History_Age (0.18): There is a weak positive correlation between Age and Credit History Age, indicating that older individuals are slightly more likely to have a longer credit history. This is expected, as age and credit history tend to grow together over time.

- Moderate Correlation:

Delay_from_due_date (0.14): There is a weak positive correlation, suggesting that older individuals may have a slightly higher tendency to delay payments. This might reflect behavior or financial conditions that require further investigation.

- Low to Negligible Correlation:

Credit_Utilization_Ratio (0.017): Minimal correlation shows that age does not strongly affect credit utilization. Annual_Income (0.0053): Negligible correlation implies that age does not directly determine annual income within this dataset. Other Features (e.g., Num_Credit_Card, Monthly_Inhand_Salary, Interest_Rate, etc.): Correlations close to zero indicate little to no relationship with age.

Insights:

- Age and Credit History Connection: The slight positive correlation between Age and Credit History Age is intuitive, as older individuals tend to have more time to establish and maintain a credit history.
- Delay_from_due_date Behavior: The weak positive correlation with payment delays might suggest that some older individuals manage payments differently, potentially due to factors like financial burdens or priorities.
- No Influence of Age on Financial Products: Features such as the number of credit cards, bank accounts, and credit inquiries show no meaningful relationship with Age, indicating that these are independent of an individual's age.

Implications:

- Modeling Considerations:

Age might have limited predictive value for most features except Credit History Age and potentially Delay_from_due_date.

- Behavioral Analysis:

The relationship between Age and payment delays could be further explored to identify patterns or trends for specific age groups.

- Focus on Other Factors:

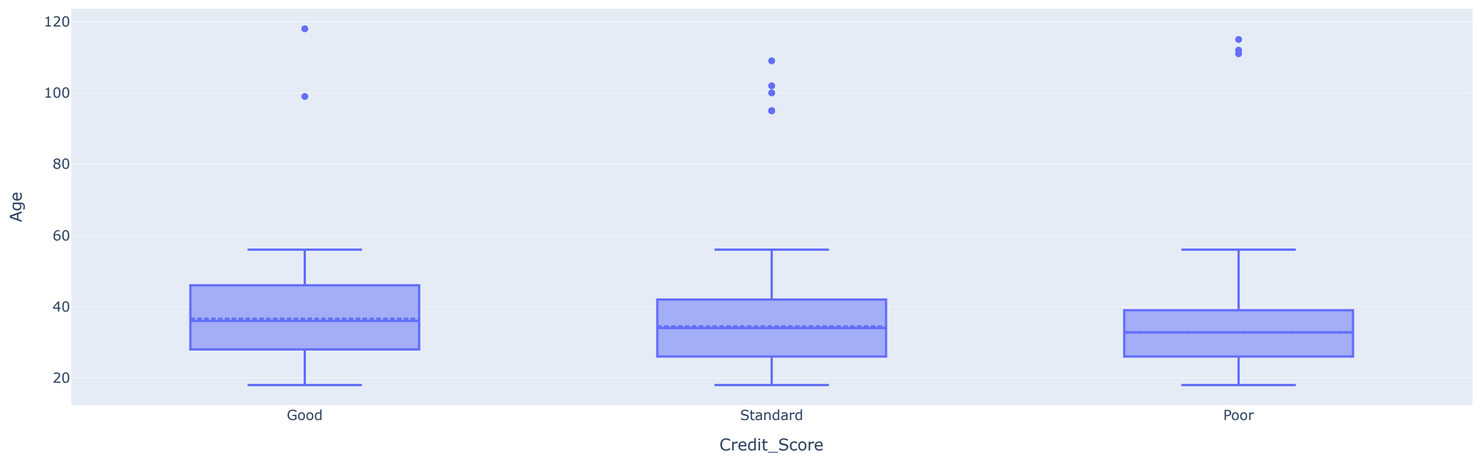
As Age has minimal correlation with most financial variables, other factors like income, spending behavior, or credit utilization are likely more influential for analysis or predictions.

Age Distribution by Credit Score

In [148…
```
imputer.impute_missing_values('Age',['Credit_History_Age', 'Delay_from_due_date'])
```

```
Missing values in column 'Age' have been imputed using the mean of groups based on '['Credit_History_Age', 'Delay_from_due_date']'.
```

In [149…
```
visualizer.plot_age_distribution_by_credit_score('Credit_Score', 'Age')
```



Occupation

The Occupation column delineates the type of employment or professional engagement of the customer. This column is integral to understanding the demographic and financial contexts of the dataset, as various occupations are associated with differing levels of income, expenditure patterns, and risk profiles. These distinctions are pivotal for credit scoring and other financial analyses, where the type of occupation can significantly influence a customer's financial behavior and creditworthiness.

In [152…
```
cleaner.get_value_count('Occupation')
```

Out[152…

	Occupation	counts	percent
0	_____	0.071	7.06%
1	Lawyer	0.066	6.58%
2	Architect	0.064	6.35%
3	Engineer	0.064	6.35%
4	Scientist	0.063	6.30%
5	Mechanic	0.063	6.29%
6	Accountant	0.063	6.27%
7	Developer	0.062	6.24%
8	Media_Manager	0.062	6.23%
9	Teacher	0.062	6.21%
10	Entrepreneur	0.062	6.17%
11	Doctor	0.061	6.09%
12	Journalist	0.061	6.08%
13	Manager	0.060	5.97%
14	Musician	0.059	5.91%
15	Writer	0.059	5.88%

Occupation - Missing Values Treatment

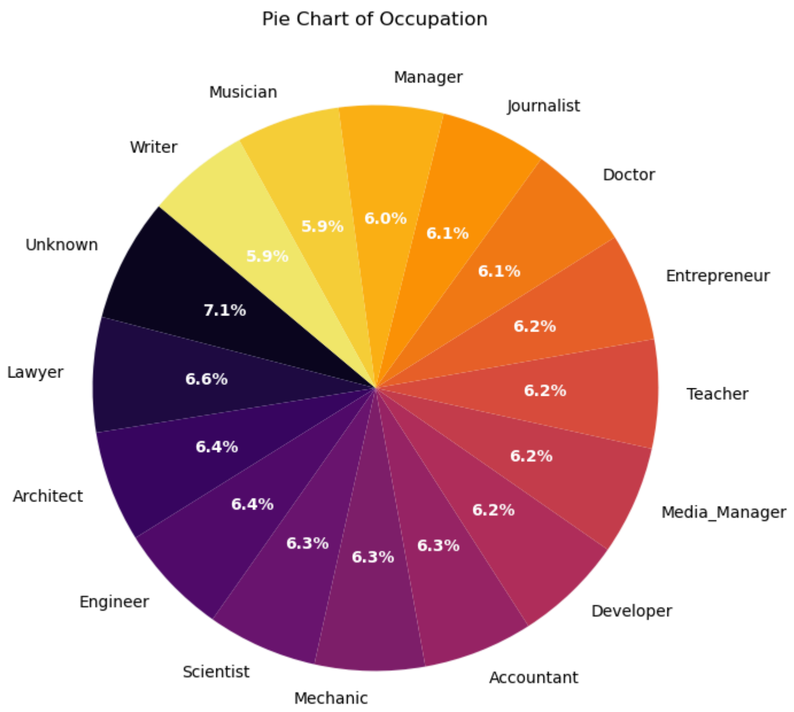
In [153…
```
cleaner.print_missing_values('Occupation')
cleaner.print_column_dtype('Occupation')
```

```
Remaining missing values in Occupation: 0
dtype of Occupation: object
```

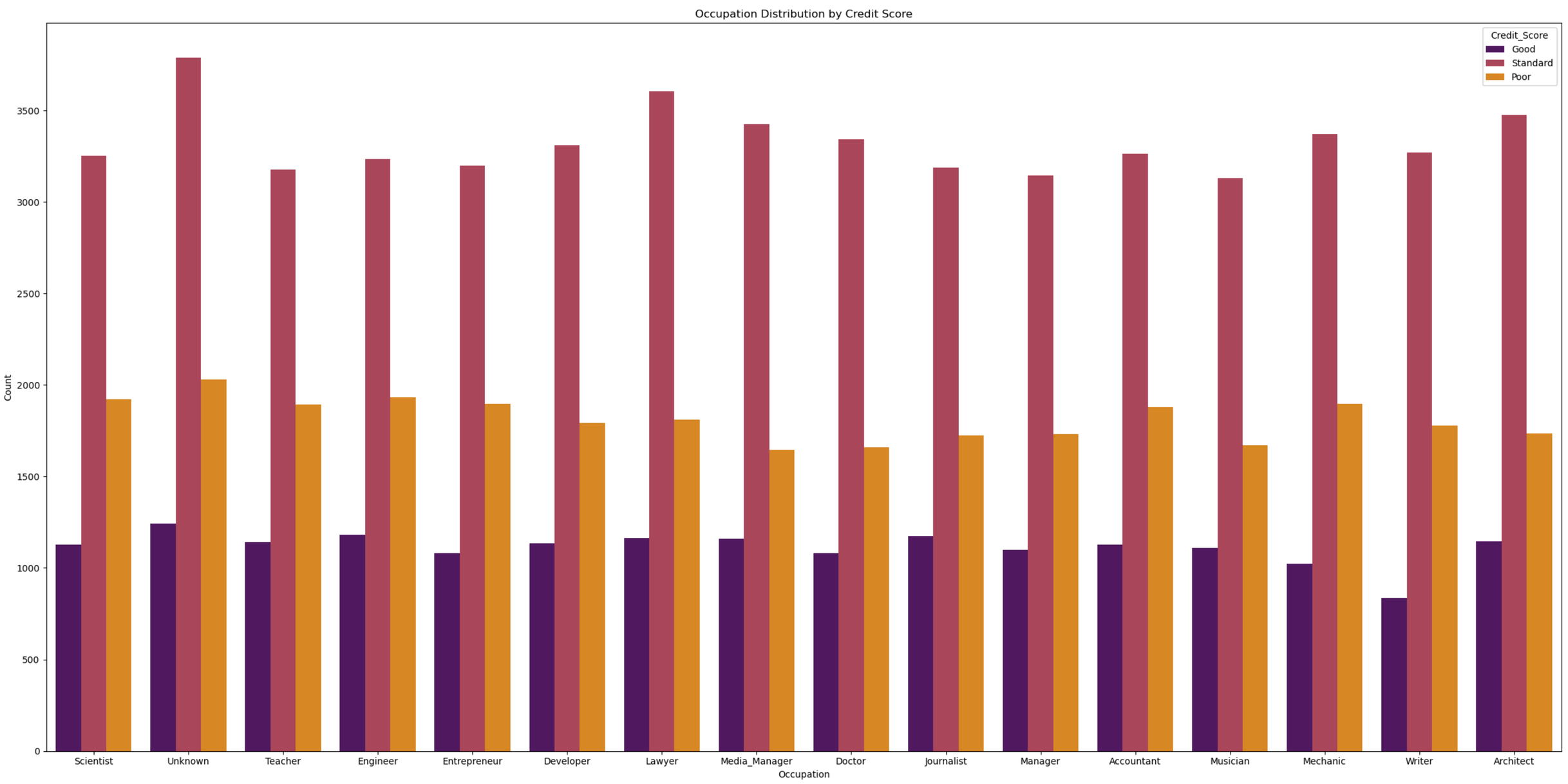
In [154…
```
cleaner.replace_value('Occupation','_______', 'Unknown')
```

```
Value '_______' in column 'Occupation' has been replaced with 'Unknown'.
```

In [155…
```
visualizer.plot_pie_chart('Occupation')
```



In [156…
```
visualizer.plot_count('Occupation', 'Credit_Score')
```



The graph illustrates that, across most occupations, the majority of individuals belong to the Standard credit score category. Specifically, occupations such as Media Manager, Lawyer, and Unknown exhibit the highest counts in this category, suggesting a consistent trend within these roles.

Conversely, the number of individuals with a Good credit score is notably lower across all occupational groups, reflecting a less common distribution pattern. In contrast, the Poor credit score category remains relatively small for most professions. However, certain roles, such as Mechanic and Musician, demonstrate a higher proportion of individuals within the poor credit group, standing out as exceptions to the overall trend.

This distribution provides insights into the variation of credit scores among occupations, highlighting the dominance of the Standard category and the nuanced disparities in Poor credit representation among specific professions.

Num Bank Accounts

In [160…
```
cleaner.get_value_count('Num_Bank_Accounts')
```

Out[160…

	Num_Bank_Accounts	counts	percent
0	6	0.130	13.00%
1	7	0.128	12.82%
2	8	0.128	12.77%
3	4	0.122	12.19%
4	5	0.121	12.12%
…	…	…	…
938	1626	0.000	0.00%
939	1470	0.000	0.00%
940	887	0.000	0.00%
941	211	0.000	0.00%
942	697	0.000	0.00%

943 rows × 3 columns

The Num_Bank_Accounts column represents the number of bank accounts held by each customer. Upon reviewing the dataset, an anomalous value of -1 was identified, which is logically inconsistent as the number of bank accounts cannot be negative. It is reasonable to interpret this value as representing missing or unknown data rather than an actual numerical count.

To address this issue, the value -1 has been replaced with 0. This replacement assumes that these customers likely do not hold any bank accounts, a plausible assumption that maintains the dataset's integrity. Replacing -1 with 0 ensures that the dataset avoids introducing missing values (NaN), which could complicate further analysis.

This preprocessing step enhances the dataset's consistency and reliability, enabling a more accurate representation of the customers' financial profiles. By addressing the anomaly in this way, the dataset remains suitable for statistical modeling and analysis without the potential distortions caused by negative or missing values.

Number of Bank Accounts - Missing Values and Negative Values Treatment

In [161…
```
cleaner.print_missing_values('Num_Bank_Accounts')
cleaner.print_column_dtype('Num_Bank_Accounts')
```

```
Remaining missing values in Num_Bank_Accounts: 0
dtype of Num_Bank_Accounts: int64
```

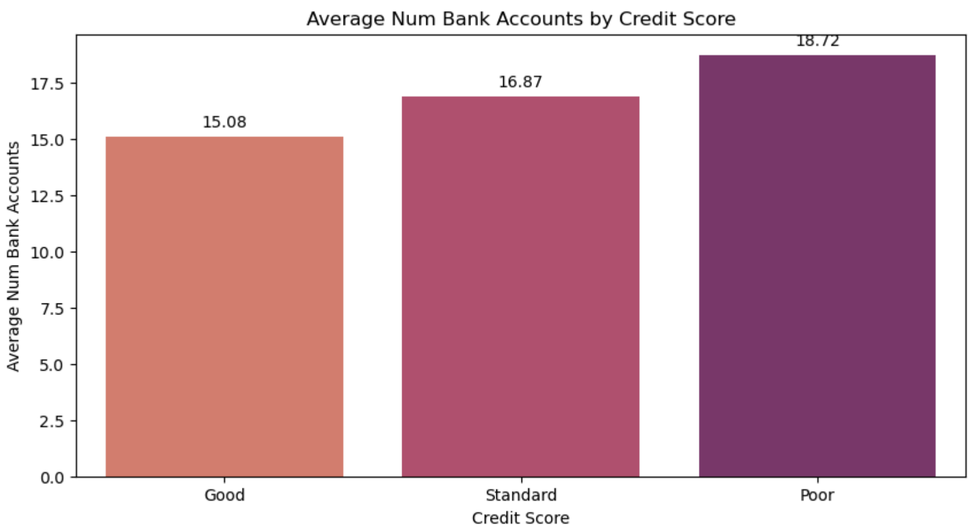
In [162…
```
cleaner.count_negative_values('Num_Bank_Accounts')
```

```
List of unique negative values in column "Num_Bank_Accounts": [-1]
Number of unique negative values in column "Num_Bank_Accounts": 21
```

In [163…
```
cleaner.replace_value('Num_Bank_Accounts',-1,0)
```

```
Value '-1' in column 'Num_Bank_Accounts' has been replaced with '0'.
```

In [164…
```
visualizer.plot_average_by_column('Credit_Score', 'Num_Bank_Accounts')
```



The analysis reveals that customers with a Poor credit score exhibit a higher average number of bank accounts (18.72) in comparison to those with Standard (16.87) and Good (15.08) credit scores. This observation suggests a potential correlation between the number of bank accounts held and credit score status, indicating that individuals with more bank accounts may tend to have poorer credit scores. Such findings underscore the importance of considering the number of bank accounts as a variable in assessing creditworthiness and financial behavior.

Number of Credit Cards

In [167…
```
cleaner.get_value_count('Num_Credit_Card')
```

Out[167…

	Num_Credit_Card	counts	percent
0	5	0.185	18.46%
1	7	0.166	16.61%
2	6	0.166	16.56%
3	4	0.140	14.03%
4	3	0.133	13.28%
...	...	...	...
1174	791	0.000	0.00%
1175	1118	0.000	0.00%
1176	657	0.000	0.00%
1177	640	0.000	0.00%
1178	679	0.000	0.00%

1179 rows × 3 columns

Number of Credit Cards - Missing Values and Negative Values Treatment

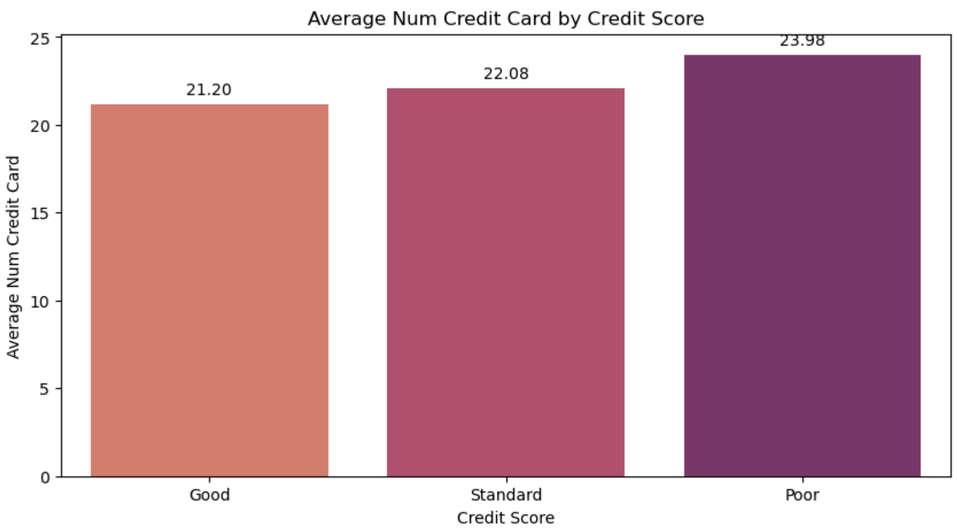
In [168…
```
cleaner.print_missing_values('Num_Credit_Card')
cleaner.print_column_dtype('Num_Credit_Card')
```

```
Remaining missing values in Num_Credit_Card: 0
dtype of Num_Credit_Card: int64
```

In [169…
```
cleaner.count_negative_values('Num_Credit_Card')
```

```
List of unique negative values in column "Num_Credit_Card": []
Number of unique negative values in column "Num_Credit_Card": 0
```

In [170…
```
visualizer.plot_average_by_column('Credit_Score', 'Num_Credit_Card')
```



The analysis suggests a potential correlation between the number of credit cards held and credit score status, similar to the observed behavior regarding the number of bank accounts. Specifically, individuals possessing a greater number of credit cards tend to exhibit poorer credit scores. This relationship underscores the necessity of including the number of credit cards as a significant variable in creditworthiness assessments and financial behavior analyses. The findings highlight the intricate dynamics between credit card ownership and credit score, warranting further examination within financial and demographic studies.

Num of Loan

In [174…
```
cleaner.get_value_count('Num_of_Loan')
```

Out[174…

	Num_of_Loan	counts	percent
0	3	0.144	14.39%
1	2	0.142	14.25%
2	4	0.140	14.02%
3	0	0.104	10.38%
4	1	0.101	10.08%
...	...	...	...
429	1320_	0.000	0.00%
430	103	0.000	0.00%
431	1444	0.000	0.00%
432	392	0.000	0.00%
433	966	0.000	0.00%

434 rows × 3 columns

Num of Loan - Missing Values and Non-numeric characters Treatment

In [178…
```
cleaner.print_missing_values('Num_of_Loan')
cleaner.print_column_dtype('Num_of_Loan')
cleaner.find_non_numeric_values('Num_of_Loan')
```

```
Remaining missing values in Num_of_Loan: 0
dtype of Num_of_Loan: object
Found non numeric values:
```

Out[178… {' ', ' -', '_ ', '_ -'}

In [180…
```
cleaner.remove_non_numeric_characters('Num_of_Loan')
```

```
Non-numeric characters removed successfully from column 'Num_of_Loan'.
```

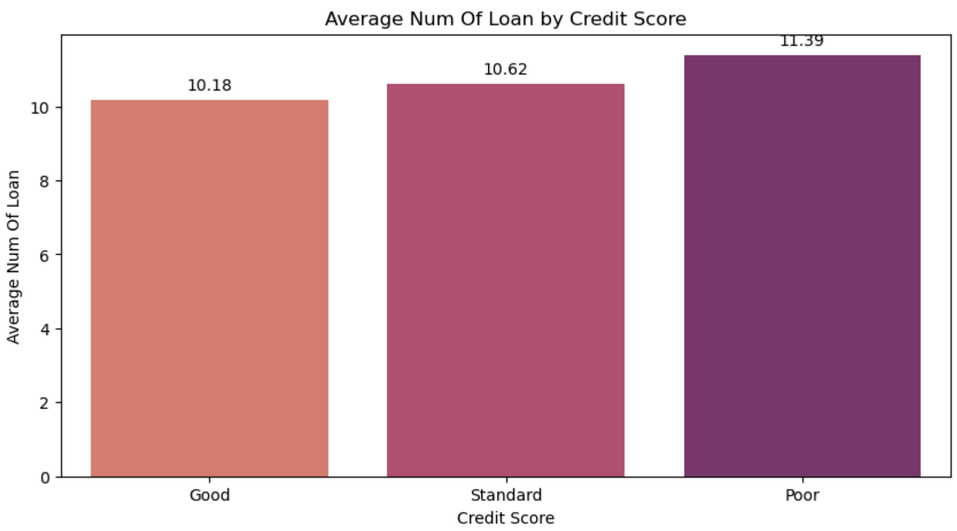
In [181…
```
cleaner.replace_empty_with_nan('Num_of_Loan')
```

```
Empty strings in column 'Num_of_Loan' replaced with NaN.
```

In [182…
```
cleaner.convert_object_to_float('Num_of_Loan')
```

```
Column 'Num_of_Loan' successfully converted to float.
```

In [183…
```
visualizer.plot_average_by_column('Credit_Score', 'Num_of_Loan')
```



The analysis indicates a potential correlation between the number of loans held and credit score status, analogous to the patterns observed with the number of bank accounts and credit cards. Specifically, individuals with a greater number of loans tend to exhibit poorer credit scores. This relationship underscores the importance of incorporating the number of loans as a critical variable in assessments of creditworthiness and financial behavior. The findings emphasize the complex dynamics between loan ownership and credit score, highlighting the need for further investigation within the context of financial and demographic studies.

Interest Rate

The Interest_Rate column denotes the interest rate applied to loans or credit obtained by the customer. This column is crucial as it provides insight into the cost of borrowing for each customer. A thorough review of the data reveals that there are no unusual or missing values in this column, confirming that all entries are both valid and complete. Consequently, the interest rate data can be utilized reliably in subsequent analyses, ensuring accuracy and integrity in financial assessments.

In [187…
```
cleaner.get_value_count('Interest_Rate')
```

Out[187…

	Interest_Rate	counts	percent
0	8	0.050	5.01%
1	5	0.050	4.98%
2	6	0.047	4.72%
3	12	0.045	4.54%
4	10	0.045	4.54%
...	...	...	...
1745	4995	0.000	0.00%
1746	1899	0.000	0.00%
1747	2120	0.000	0.00%
1748	5762	0.000	0.00%
1749	5729	0.000	0.00%

1750 rows × 3 columns

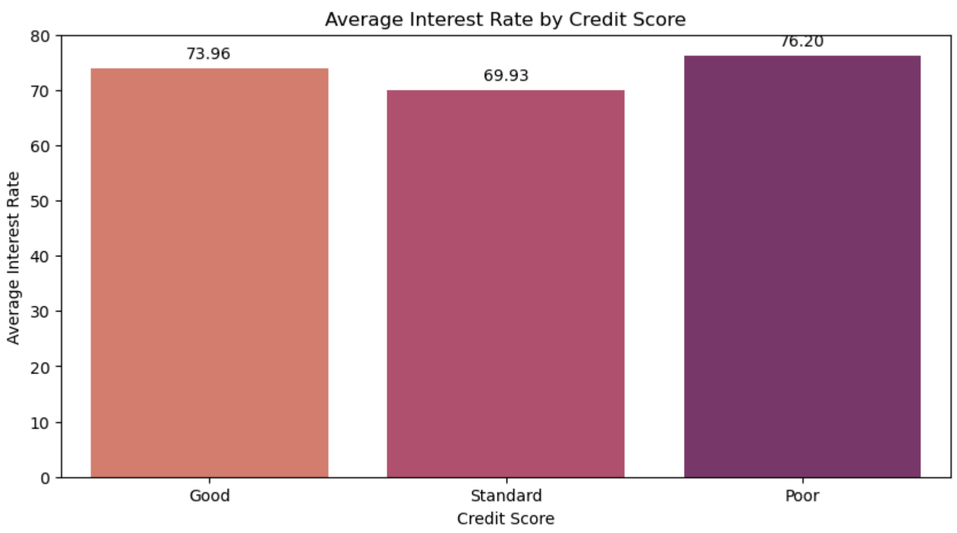
Interest Rate - Missing Values and Non-numeric characters Treatment

In [188…
```
cleaner.print_missing_values('Interest_Rate')
cleaner.print_column_dtype('Interest_Rate')
cleaner.find_non_numeric_values('Interest_Rate')
```

```
Remaining missing values in Interest_Rate: 0
dtype of Interest_Rate: int64
Found non numeric values:
```

Out[188… {' '}

In [189…
```
visualizer.plot_average_by_column('Credit_Score', 'Interest_Rate')
```



The analysis reveals that individuals with a Poor credit score bear the highest average interest rate at 76.20, whereas those with a Standard credit score benefit from the lowest average rate of 69.93. Notably, customers with a Good credit score incur a relatively high average interest rate of 73.96. This disparity suggests potential inconsistencies in the assignment of interest rates across different credit score categories, warranting further investigation into the criteria and mechanisms that govern these financial decisions.

Delay from due date

The Delay_from_due_date column represents the number of days a payment is overdue, serving as a critical indicator of a customer's payment behavior and financial responsibility. This feature is pivotal for analysis and modeling as it directly correlates with the risk of default, making it highly relevant for credit scoring and risk assessment models.

Upon reviewing the data, only unusual negative values are present, and there are no missing entries. The presence of negative values suggests they may have been entered erroneously, particularly since the Delay_from_due_date column is intended to indicate the number of days a payment is overdue, which should inherently be a positive or zero value. The diversity of negative values (e.g., -1, -2, -3, etc.) further implies that these entries were likely meant to be positive but were incorrectly recorded as negative.

This insight underscores the need for data validation and correction to ensure the accuracy and reliability of subsequent analyses. By addressing these anomalies, we can maintain the integrity of the dataset and enhance the effectiveness of credit scoring and risk assessment processes.

In [193…
```
cleaner.get_value_count('Delay_from_due_date')
```

Out[193…

	Delay_from_due_date	counts	percent
0	15	0.036	3.60%
1	13	0.034	3.42%
2	8	0.033	3.32%
3	14	0.033	3.31%
4	10	0.033	3.28%
...	...	...	...
68	-4	0.001	0.06%
69	65	0.001	0.06%
70	-5	0.000	0.03%
71	66	0.000	0.03%
72	67	0.000	0.02%

73 rows × 3 columns

Delay from due date - Missing Values and Negative Values Treatment

In [194…
```
cleaner.print_missing_values('Delay_from_due_date')
cleaner.print_column_dtype('Delay_from_due_date')
```

```
Remaining missing values in Delay_from_due_date: 0
dtype of Delay_from_due_date: int64
```

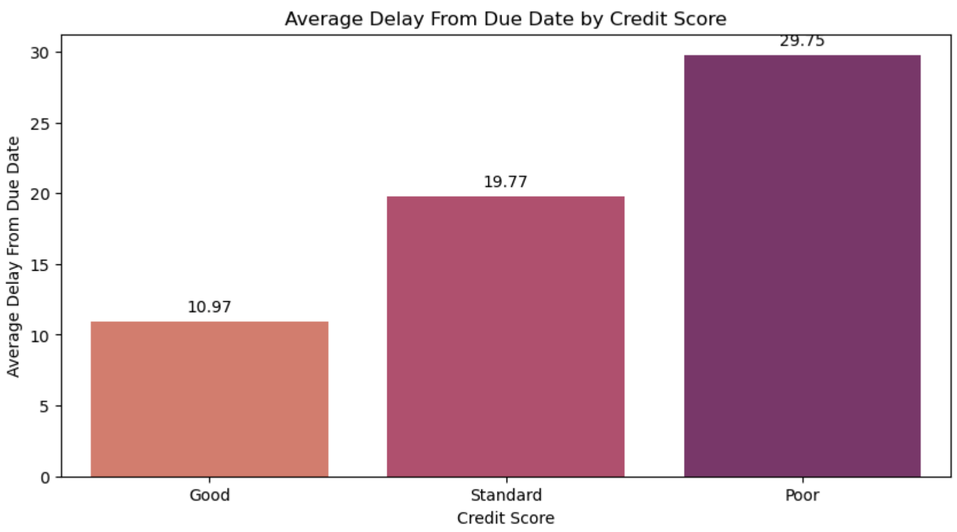
In [195…
```
cleaner.count_negative_values('Delay_from_due_date')
```

```
List of unique negative values in column "Delay_from_due_date": [-1 -2 -3 -5 -4]
Number of unique negative values in column "Delay_from_due_date": 591
```

In [196…
```
cleaner.convert_to_absolute('Delay_from_due_date')
```

```
Values in column 'Delay_from_due_date' have been converted to their absolute values.
```

In [197…
```
visualizer.plot_average_by_column('Credit_Score', 'Delay_from_due_date')
```



The analysis indicates that customers with a Poor credit score experience the longest average delay in payments, with a mean delay of 29.75 days. Conversely, those with a Good credit score exhibit the shortest average delay, at 10.97 days. This observation suggests a strong correlation between delayed payments and poorer credit scores, highlighting the importance of timely payments in maintaining a favorable credit rating. The findings underscore the critical role that payment punctuality plays in creditworthiness assessments and financial stability.

Changed Credit Limit

The Changed_Credit_Limit column records variations in a customer's credit limit, with positive values indicating an increase and negative values indicating a decrease. During data preprocessing, anomalous characters such as ' ', '-', ' _ ', ' _ -', and ' _ _ ' were identified and rectified. The _ characters were replaced with 0, under the assumption that they signify no change in the credit limit. This ensures that the data remains numeric and clean for analysis. Negative values were retained, as they are valid indicators of a reduction in the credit limit, which is essential for understanding customer risk. This approach maintains the integrity and accuracy of the dataset, thereby enhancing its suitability for subsequent analyses related to creditworthiness and financial risk assessments.

In [201…
```
cleaner.get_value_count('Changed_Credit_Limit')
```

Out[201…

	Changed_Credit_Limit	counts	percent
0	_	0.021	2.09%
1	8.22	0.001	0.13%
2	11.5	0.001	0.13%
3	11.32	0.001	0.13%
4	7.35	0.001	0.12%
...	...	...	...
4379	-1.84	0.000	0.00%
4380	0.8899999999999999	0.000	0.00%
4381	28.06	0.000	0.00%
4382	1.5599999999999996	0.000	0.00%
4383	21.17	0.000	0.00%

4384 rows × 3 columns

Changed Credit Limit - Missing Values and Non-numeric characters Treatment

In [203…
```
cleaner.remove_non_numeric_characters('Changed_Credit_Limit')
```

```
Non-numeric characters removed successfully from column 'Changed_Credit_Limit'.
```

In [202…
```
cleaner.print_missing_values('Changed_Credit_Limit')
cleaner.print_column_dtype('Changed_Credit_Limit')
cleaner.find_non_numeric_values('Changed_Credit_Limit')
```

```
Remaining missing values in Changed_Credit_Limit: 0
dtype of Changed_Credit_Limit: object
Found non numeric values:
```

Out[202…
```
{' ', ' -', ' _ ', ' _ -', ' _ _ ', '.'}
```

In [204…
```
cleaner.replace_value('Changed_Credit_Limit','', 0)
```

```
Value '' in column 'Changed_Credit_Limit' has been replaced with '0'.
```

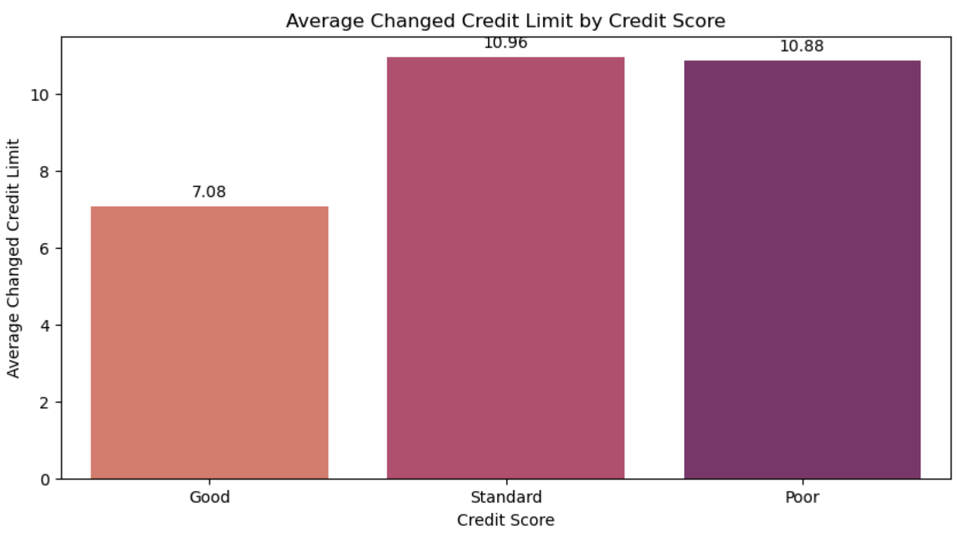
In [205…
```
cleaner.convert_object_to_float('Changed_Credit_Limit')
```

```
Column 'Changed_Credit_Limit' successfully converted to float.
```

In [206…
```
cleaner.count_negative_values('Changed_Credit_Limit')
```

```
List of unique negative values in column "Changed_Credit_Limit": []
Number of unique negative values in column "Changed_Credit_Limit": 0
```

In [207…
```
visualizer.plot_average_by_column('Credit_Score', 'Changed_Credit_Limit')
```



The analysis demonstrates that customers with Standard and Poor credit scores have experienced similar and higher average changes to their credit limits, with values around 10.96 and 10.88, respectively. In contrast, those with a Good credit score exhibit a lower average change of 7.08. This trend suggests that credit limit adjustments are more frequent or substantial among customers with lower credit scores. Such observations highlight the potential relationship between credit score levels and the propensity for credit limit modifications, which could be a vital factor in understanding credit management practices and customer risk profiles.

Credit_Mix

The Credit_Mix column delineates the variety of credit types utilized by a customer, encompassing instruments such as credit cards, loans, and mortgages. A well-balanced credit mix is known to positively influence a customer's credit score by demonstrating their competency in managing diverse credit forms.

In the dataset, values marked with _ are presumed to represent missing or unknown data. To address this issue, these _ values were systematically replaced with 'Unknown', thus indicating the absence of provided information. This methodological approach ensures that the dataset remains intact and comprehensive, facilitating subsequent analyses without the complication of missing values. This strategy enables accurate interpretation and enhances the reliability of creditworthiness and financial behavior assessments.

In [211…
```
cleaner.get_value_count('Credit_Mix')
```

Out[211…

	Credit_Mix	counts	percent
0	Standard	0.365	36.48%
1	Good	0.243	24.34%
2	_	0.202	20.20%
3	Bad	0.190	18.99%

Credit_Mix - Missing Values Treatment

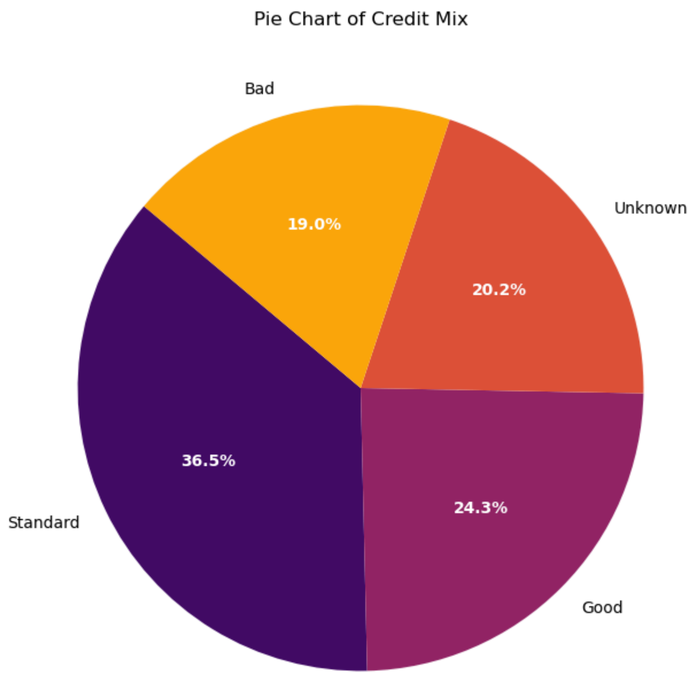
In [212…
```
cleaner.print_missing_values('Credit_Mix')
cleaner.print_column_dtype('Credit_Mix')
```

```
Remaining missing values in Credit_Mix: 0
dtype of Credit_Mix: object
```

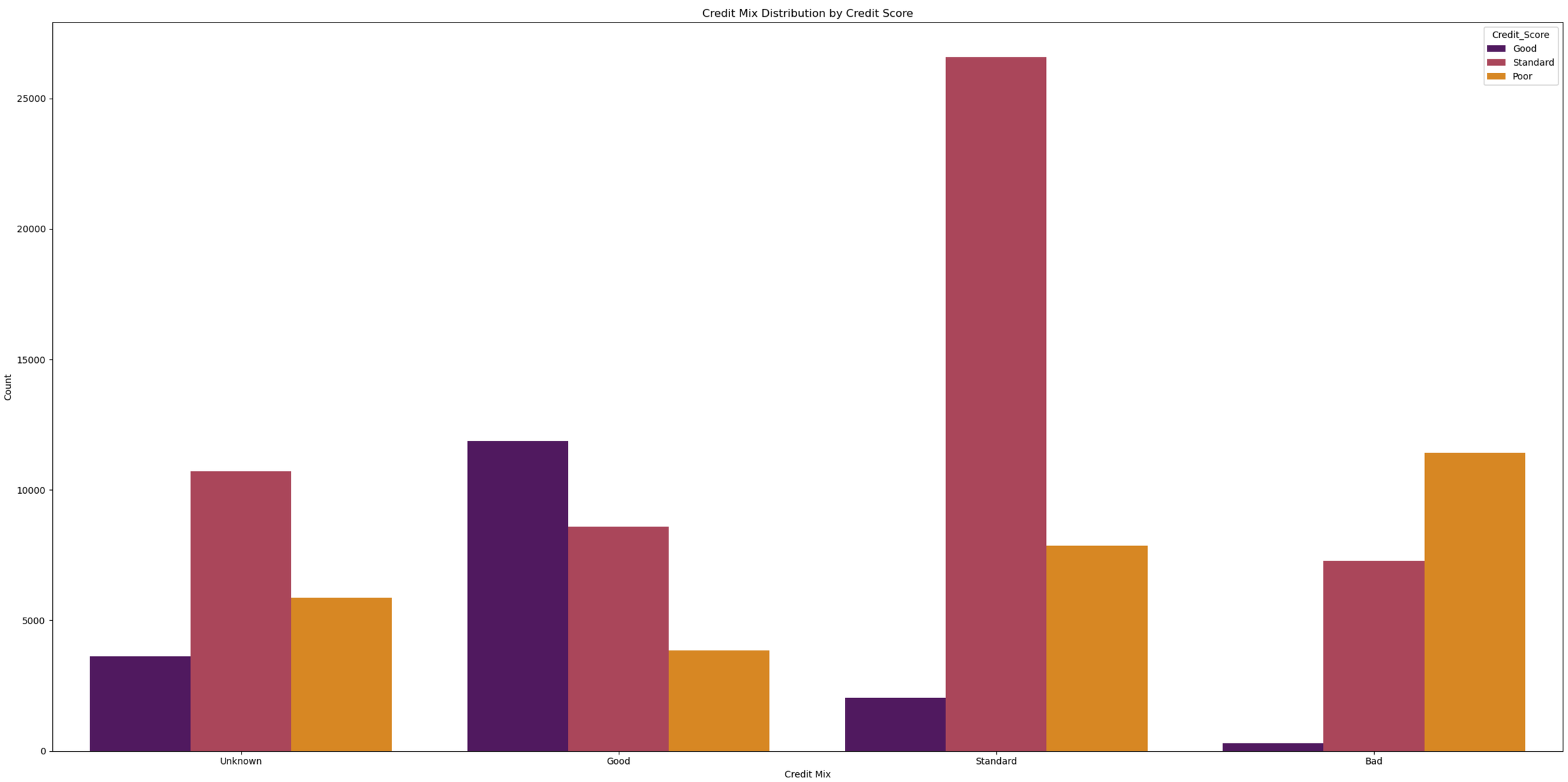
In [213…
```
cleaner.replace_value('Credit_Mix','_', 'Unknown')
```

```
Value '_' in column 'Credit_Mix' has been replaced with 'Unknown'.
```

In [214…
```
visualizer.plot_pie_chart('Credit_Mix')
```



In [215…
```
visualizer.plot_count('Credit_Mix','Credit_Score')
```



The analysis indicates that a Good Credit Mix is strongly associated with Good Credit Scores. Conversely, a Bad Credit Mix is more prevalent among individuals with Poor Credit Scores. The Standard Credit Mix appears most frequently within both Standard and Poor Credit Score categories, indicating a balanced yet less optimized credit type. Additionally, an Unknown Credit Mix does not exhibit a strong correlation with any specific credit score. These findings underscore the significance of maintaining a diverse and well-managed credit mix in achieving higher credit scores, while also highlighting areas for further examination and improvement.

Outstanding Debt

The Outstanding_Debt column captures the total amount of unpaid debt that a customer owes, encompassing balances from loans, credit cards, and other lines of credit. This column is crucial as it offers a comprehensive view of the customer's current financial obligations, which is essential for assessing credit risk and evaluating the customer's capacity to manage additional debt.

Upon reviewing the dataset, it was found that there were no missing values; however, _ characters were present in place of numeric values. These anomalous characters were cleaned, and the data type was converted to numeric to ensure that the column is appropriately formatted for analysis and modeling purposes. This process ensures that the debt values are accurately managed and can be effectively utilized in financial calculations, thus enhancing the reliability and precision of the analysis.

In [219…
```
cleaner.get_value_count('Outstanding_Debt')
```

Out[219…

	Outstanding_Debt	counts	percent
0	1360.45	0.000	0.02%
1	460.46	0.000	0.02%
2	1151.7	0.000	0.02%
3	1109.03	0.000	0.02%
4	467.7	0.000	0.02%
...	...	...	...
13173	245.46_	0.000	0.00%
13174	645.77_	0.000	0.00%
13175	174.79_	0.000	0.00%
13176	1181.13_	0.000	0.00%
13177	1013.53_	0.000	0.00%

13178 rows × 3 columns

Outstanding Debt - Missing Values and Non-Numeric Characters Treatment

In [220…
```
cleaner.print_missing_values('Outstanding_Debt')
cleaner.print_column_dtype('Outstanding_Debt')
cleaner.find_non_numeric_values('Outstanding_Debt')
```

```
Remaining missing values in Outstanding_Debt: 0
dtype of Outstanding_Debt: object
Found non numeric values:
```

Out[220… {' ', '.', '_ '}

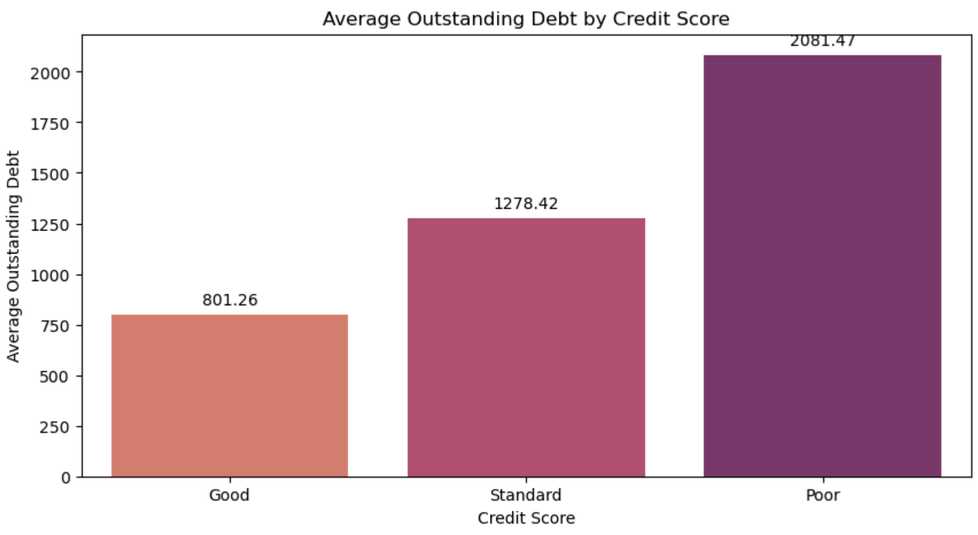
In [221…
```
cleaner.remove_non_numeric_characters('Outstanding_Debt')
```

```
Non-numeric characters removed successfully from column 'Outstanding_Debt'.
```

In [222…
```
cleaner.convert_object_to_float('Outstanding_Debt')
```

```
Column 'Outstanding_Debt' successfully converted to float.
```

In [223…
```
visualizer.plot_average_by_column('Credit_Score', 'Outstanding_Debt')
```



The analysis indicates that customers with poor credit scores exhibit significantly higher outstanding debt compared to those with standard and good credit scores. This observation suggests a clear relationship between elevated outstanding debt and poorer credit scores, highlighting that debt management is a considerable challenge for individuals with lower credit ratings. Conversely, customers with good credit scores tend to have much lower outstanding debt, which reflects a higher level of financial responsibility and effective debt management. These findings underscore the importance of outstanding debt as a critical factor in credit score assessment and financial health evaluation.

Credit Utilization Ratio

The Credit_Utilization_Ratio column quantifies the proportion of a customer's current outstanding debt relative to their total available credit. This ratio, typically expressed as a percentage, assesses the extent to which the customer utilizes their available credit.

Upon reviewing the dataset, it was found that there were no missing or unusual values within this column, ensuring its reliability for analysis.

Higher Ratios: Elevated credit utilization ratios (e.g., above 30%) may suggest that a customer is heavily dependent on credit and could face challenges in repaying future debts, thereby negatively impacting their credit score.

Lower Ratios: Conversely, lower credit utilization ratios indicate that the customer is managing their credit effectively by not excessively utilizing available credit limits. This responsible credit management is viewed positively in credit scoring.

These insights underline the critical role of the credit utilization ratio in evaluating credit risk and financial behavior.

In [227…
```
cleaner.get_value_count('Credit_Utilization_Ratio')
```

Out[227…

	Credit_Utilization_Ratio	counts	percent
0	26.823	0.000	0.00%
1	28.328	0.000	0.00%
2	30.017	0.000	0.00%
3	25.479	0.000	0.00%
4	33.934	0.000	0.00%
...	...	...	...
99995	30.687	0.000	0.00%
99996	38.730	0.000	0.00%
99997	30.018	0.000	0.00%
99998	27.280	0.000	0.00%
99999	34.192	0.000	0.00%

100000 rows × 3 columns

Credit Utilization Ratio - Missing Values and Non-Numeric Characters Treatment

In [228…
```
cleaner.print_missing_values('Credit_Utilization_Ratio')
cleaner.print_column_dtype('Credit_Utilization_Ratio')
cleaner.find_non_numeric_values('Credit_Utilization_Ratio')
```

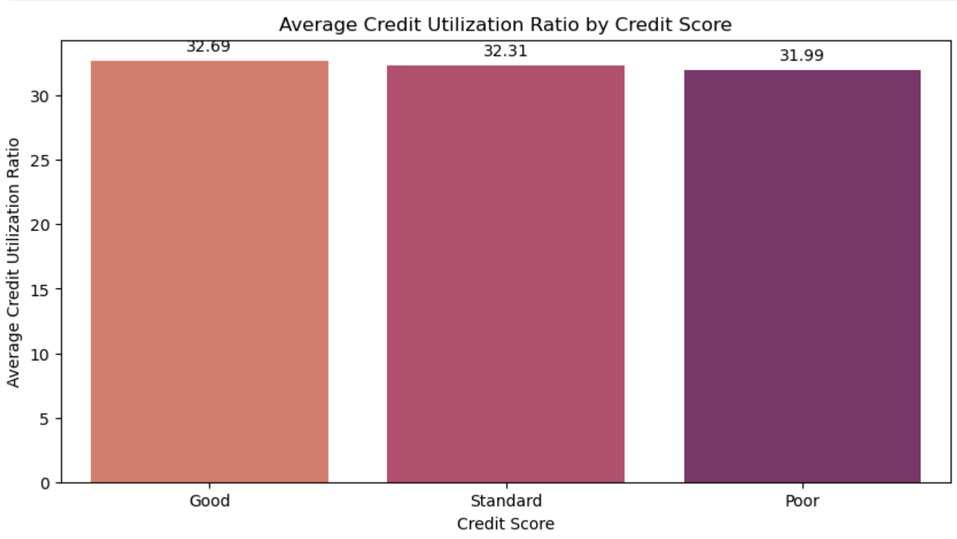
```
Remaining missing values in Credit_Utilization_Ratio: 0
dtype of Credit_Utilization_Ratio: float64
Found non numeric values:
```

Out[228… {' ', '.'}

In [229…
```
cleaner.count_negative_values('Credit_Utilization_Ratio')
```

```
List of unique negative values in column "Credit_Utilization_Ratio": []
Number of unique negative values in column "Credit_Utilization_Ratio": 0
```

In [230…
```
visualizer.plot_average_by_column('Credit_Score', 'Credit_Utilization_Ratio')
```



The plot reveals that the average Credit Utilization Ratio is fairly consistent across all credit score categories. Notably, individuals with a Good Credit Score have a slightly higher ratio (32.69%) compared to those with Standard (32.31%) and Poor (31.99%) credit scores. This finding suggests that credit utilization does not significantly vary among these groups, indicating a relatively uniform pattern of credit usage across different credit score categories.

Payment Behaviour

The Payment_Behaviour column delineates the spending and payment patterns of customers, categorizing their transactions into small, medium, or large amounts. This feature is instrumental in comprehensively understanding a customer's financial management practices, particularly in terms of expenditure and repayments. By analyzing this column, one can gain valuable insights into the customer's financial behavior, which is pivotal for assessing their creditworthiness and overall financial health.

In [234…
```
cleaner.get_value_count('Payment_Behaviour')
```

Out[234…

	Payment_Behaviour	counts	percent
0	Low_spent_Small_value_payments	0.255	25.51%
1	High_spent_Medium_value_payments	0.175	17.54%
2	Low_spent_Medium_value_payments	0.139	13.86%
3	High_spent_Large_value_payments	0.137	13.72%
4	High_spent_Small_value_payments	0.113	11.34%
5	Low_spent_Large_value_payments	0.104	10.42%
6	!@9#%8	0.076	7.60%

Payment Behaviour - Missing Values and Non-Numeric Characters Treatment

In [235…
```
cleaner.print_missing_values('Payment_Behaviour')
cleaner.print_column_dtype('Payment_Behaviour')
```

```
Remaining missing values in Payment_Behaviour: 0
dtype of Payment_Behaviour: object
```

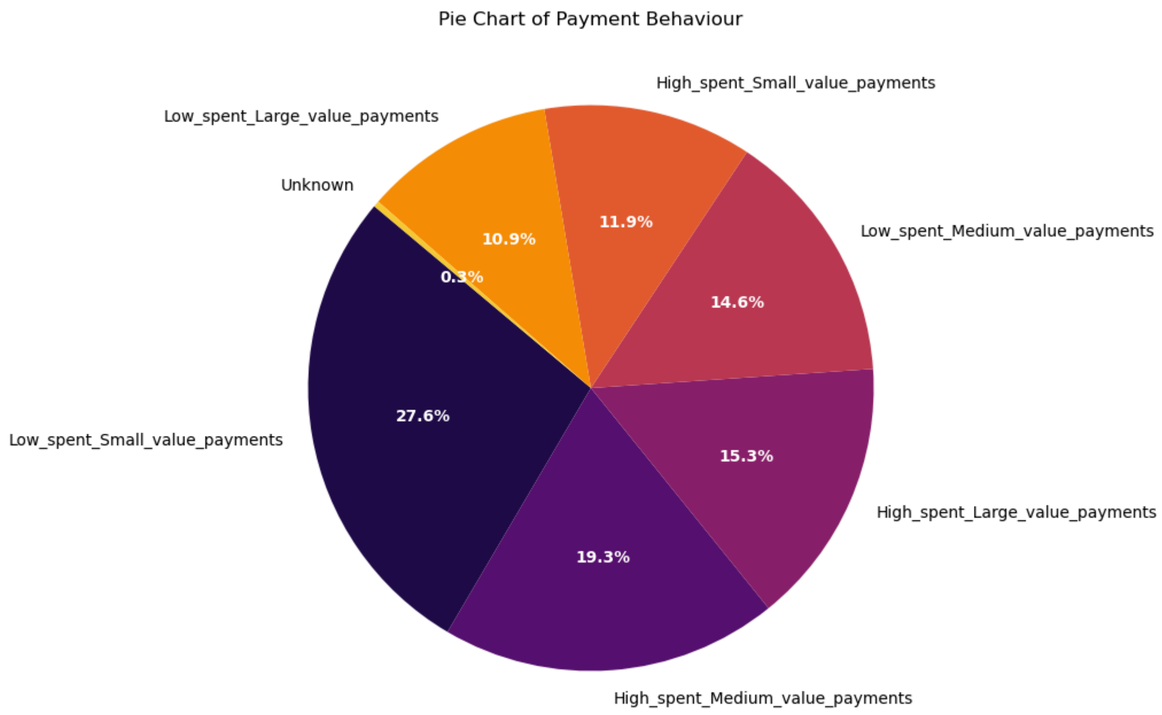
In [236…
```
cleaner.replace_value('Payment_Behaviour','!@9#%8', np.nan)
```

```
Value '!@9#%8' in column 'Payment_Behaviour' has been replaced with 'nan'.
```

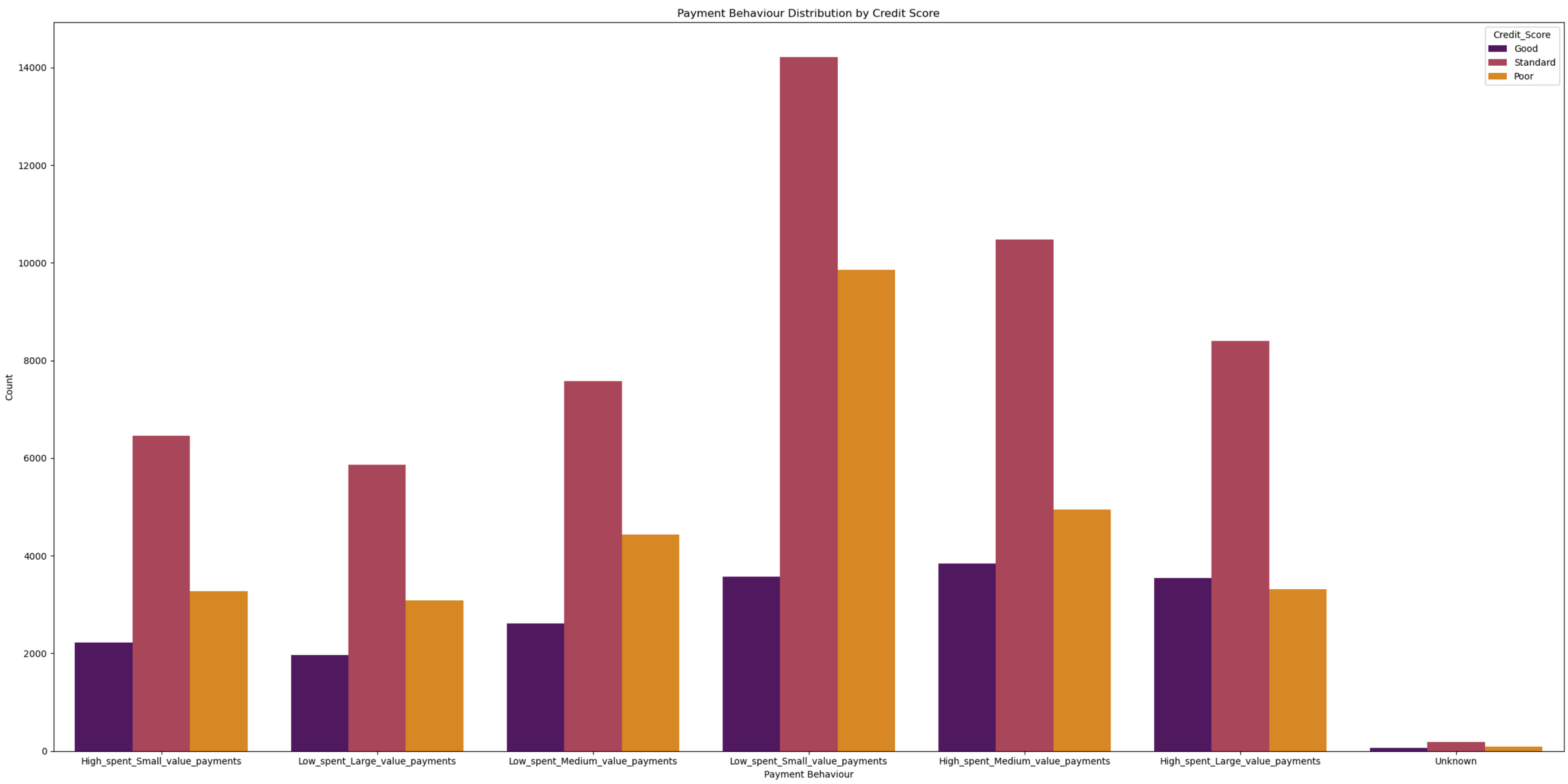
In [237…
```
imputer.fill_payment_columns('Payment_Behaviour', ['Outstanding_Debt', 'Num_of_Loan', 'Annual_Income'])
```

```
Missing values in column 'Payment_Behaviour' have been filled based on groups ['Outstanding_Debt', 'Num_of_Loan', 'Annual_Income']
```

In [238…
```
visualizer.plot_pie_chart('Payment_Behaviour')
```



In [239…
```
visualizer.plot_count('Payment_Behaviour','Credit_Score')
```



The analysis indicates that customers with a Standard credit score exhibit a higher proportion of both low-spent small value payments and high-spent large value payments compared to other credit score categories. Conversely, customers with a Poor credit score demonstrate more low-spent small value payments, suggesting more frequent small transactions, yet reflecting lower overall financial responsibility. In contrast, customers with a Good credit score generally show fewer high-spent large value payments, which implies more disciplined or stable financial behavior.

This observed pattern underscores the relationship between payment behavior and credit score, where higher credit scores are associated with more responsible and consistent payment habits. Understanding this relationship is crucial for comprehending customer financial behaviors and predicting their creditworthiness, making it an essential factor in credit risk models.

Payment of Min Amount

The Payment_of_Min_Amount column indicates whether a customer has fulfilled the minimum payment requirements for their credit or loan obligations. This column includes the following values: Yes signifies that the customer has paid the minimum required amount, No indicates that the customer has not paid the minimum required amount, and NM likely denotes missing or unreported data regarding the payment of the minimum amount. This column is instrumental in assessing a customer's payment behavior and financial responsibility, serving as a crucial factor in credit risk evaluations and lending decisions.

In [243…
```
cleaner.get_value_count('Payment_of_Min_Amount')
```

Out[243…

	Payment_of_Min_Amount	counts	percent
0	Yes	0.523	52.33%
1	No	0.357	35.67%
2	NM	0.120	12.01%

Payment of Min Amount - Missing Values Treatment

In [244…
```
cleaner.print_missing_values('Payment_of_Min_Amount')
cleaner.print_column_dtype('Payment_of_Min_Amount')
```

```
Remaining missing values in Payment_of_Min_Amount: 0
dtype of Payment_of_Min_Amount: object
```

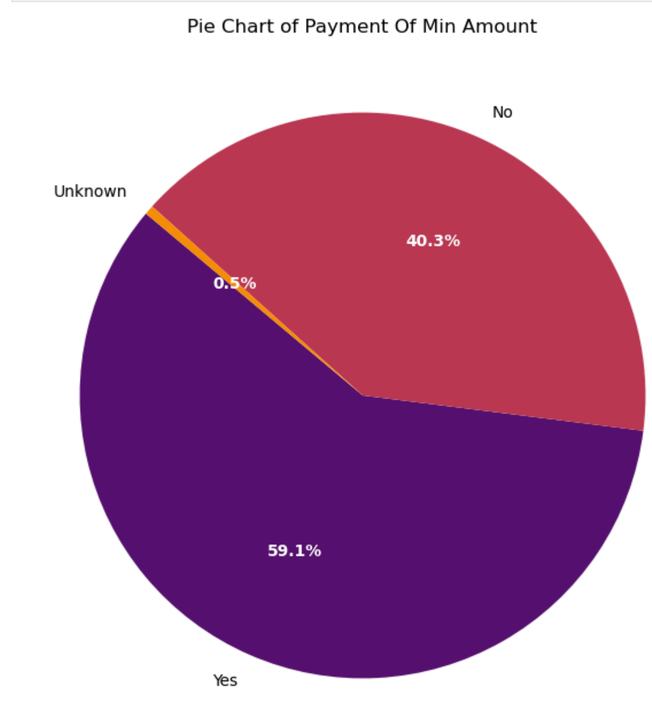
In [245…
```
cleaner.replace_value('Payment_of_Min_Amount','NM', np.nan)
```

```
Value 'NM' in column 'Payment_of_Min_Amount' has been replaced with 'nan'.
```

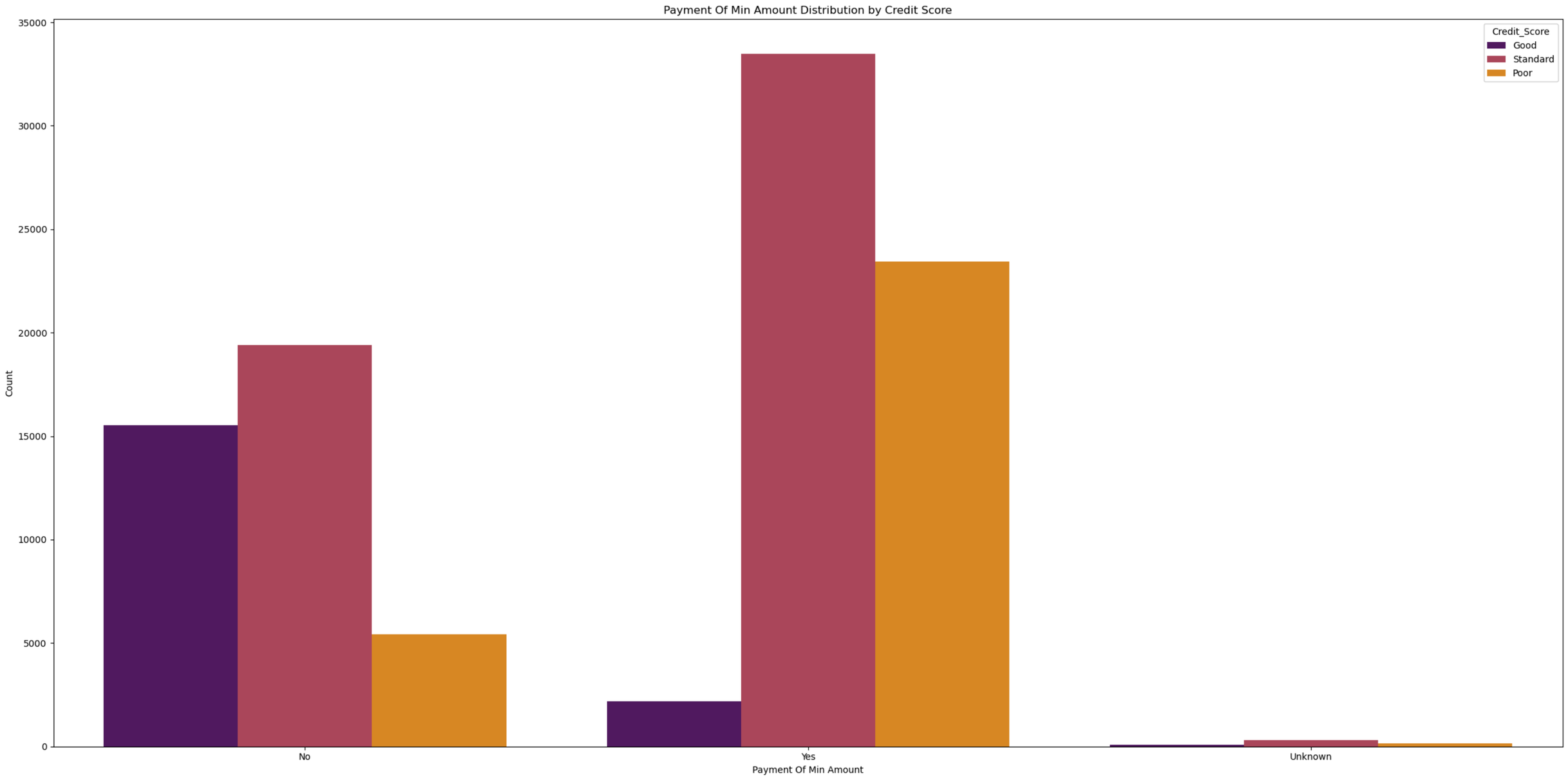
In [246…
```
imputer.fill_payment_columns('Payment_of_Min_Amount', ['Outstanding_Debt', 'Num_of_Loan', 'Annual_Income'])
```

```
Missing values in column 'Payment_of_Min_Amount' have been filled based on groups ['Outstanding_Debt', 'Num_of_Loan', 'Annual_Income']
```

In [247…
```
visualizer.plot_pie_chart('Payment_of_Min_Amount')
```



In [248…
```
visualizer.plot_count('Payment_of_Min_Amount','Credit_Score')
```



The analysis indicates that customers with standard and poor credit scores are more inclined to make the minimum payment ('Yes'). In contrast, customers with good credit scores display a more balanced distribution between making and not making the minimum payment. Additionally, the 'Unknown' category is minimal across all credit scores, suggesting that most customers exhibit clear payment behaviors. This pattern highlights that making the minimum payment is more prevalent among customers with lower credit scores. These findings are crucial for understanding customer payment practices and their implications for credit risk assessment.

Total EMI per month

The Total_EMI_per_month column represents the monthly loan installment amounts paid by a customer, encompassing both principal and interest components. This metric is crucial as it reflects the customer's financial commitment to their loans. Elevated EMIs indicate a higher debt burden, which could affect the customer's capacity to undertake additional loans or manage other financial obligations. Notably, there are no missing or unusual values in this column, confirming that it is well-prepared for subsequent analysis.

In [252…
```
cleaner.get_value_count('Total_EMI_per_month')
```

Out[252…

	Total_EMI_per_month	counts	percent
0	0.000	0.106	10.61%
1	49.575	0.000	0.01%
2	73.533	0.000	0.01%
3	22.961	0.000	0.01%
4	38.661	0.000	0.01%
...	...	...	...
14945	36408.000	0.000	0.00%
14946	23760.000	0.000	0.00%
14947	24612.000	0.000	0.00%
14948	24325.000	0.000	0.00%
14949	58638.000	0.000	0.00%

14950 rows × 3 columns

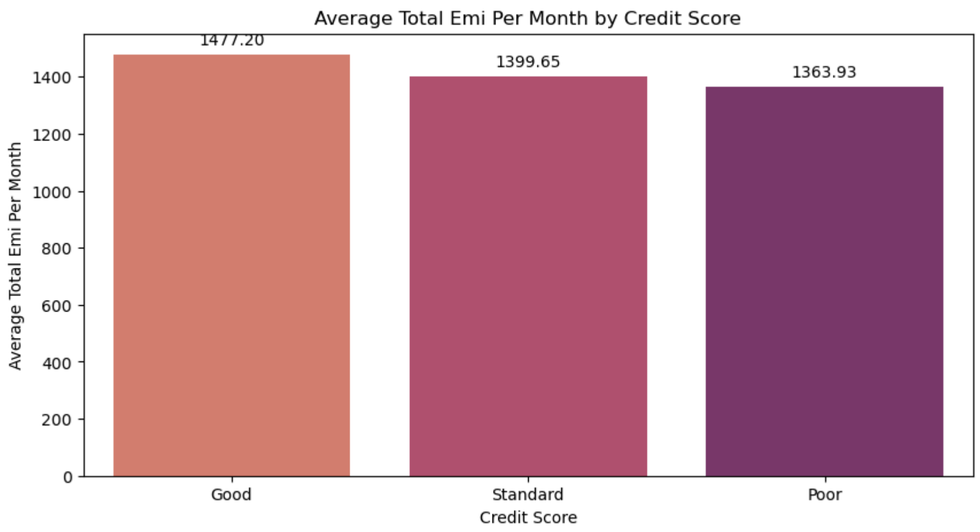
Total EMI per month - Missing Values and Non-Numeric Characters Treatment

In [253…
```
cleaner.print_missing_values('Total_EMI_per_month')
cleaner.print_column_dtype('Total_EMI_per_month')
cleaner.find_non_numeric_values('Total_EMI_per_month')
```

```
Remaining missing values in Total_EMI_per_month: 0
dtype of Total_EMI_per_month: float64
Found non numeric values:
```

Out[253… {' ', '.'}

In [256…
```
visualizer.plot_average_by_column('Credit_Score', 'Total_EMI_per_month')
```



The analysis reveals that customers with a good credit score typically have slightly higher Total EMI per month compared to those with standard and poor credit scores. This observation suggests that individuals with higher credit scores are likely managing larger loans or have more substantial financial commitments. However, their ability to handle debt more effectively contributes to maintaining a higher credit score. These findings underscore the relationship between debt management and creditworthiness, highlighting the importance of responsible financial behavior in achieving and sustaining a good credit rating.

Dropped Columns not Contributing to the Predictive power of the model

The Name column represents the full name of the customer. While it provides identification, it is typically not useful for analysis or modeling purposes and may be removed or anonymized to protect privacy during data preprocessing. Similarly, identification columns such as ID, Customer_ID, Name, and Social Security Number (SSN) are not directly beneficial in classification tasks. These columns do not contribute to the predictive power of the model and may increase the complexity rather than enhancing performance. Therefore, these columns will be excluded from the dataset during the data cleaning process to streamline the analysis and protect customer privacy.

The Month column, in which all values are uniform, fails to provide any variation or predictive power for the model. Given its lack of meaningful behavioral representation, this column does not contribute to the analysis. Additionally, the dataset lacks valid seasonality, further diminishing the relevance of the Month column. Consequently, its inclusion does not reflect any significant patterns or trends that could enhance the model's predictive capabilities.

In [266…
```
cleaner.drop_columns(['ID', 'Customer_ID', 'Name', 'SSN','Month'])
```

```
Columns ['ID', 'Customer_ID', 'Name', 'SSN', 'Month'] dropped (if they existed).
```

Target Variable: Credit Score

The Credit_Score feature categorizes customers into Good, Standard, and Poor based on their creditworthiness. In real-world finance, this categorization assists banks in assessing risk, approving loans, and setting interest rates. Higher credit scores indicate lower risk, which typically results in more favorable loan terms, while lower scores suggest higher risk.

In the context of the dataset, the Credit_Score serves as the target variable to be predicted. By analyzing various features such as income, debt, and payment behavior, it becomes possible to classify customers into these credit score categories. This classification facilitates better risk assessment and decision-making, enabling financial institutions to make informed credit-related decisions.

In [269…
```
cleaner.get_value_count('Credit_Score')
```

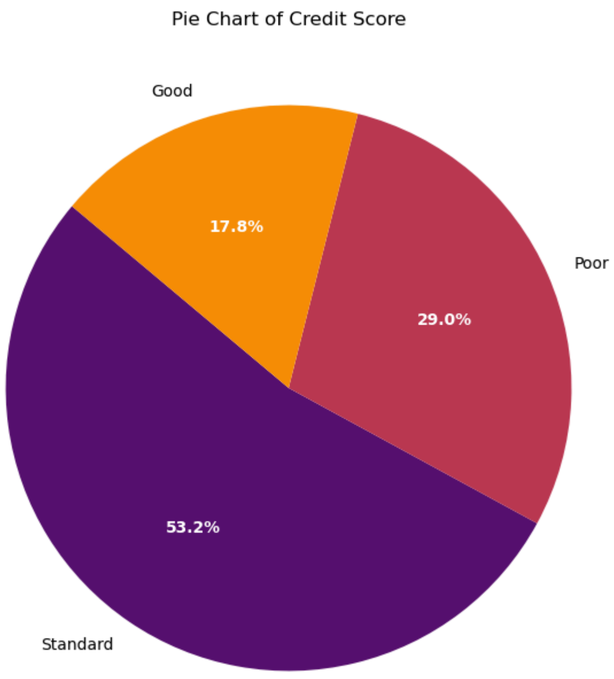
Out[269…

	Credit_Score	counts	percent
0	Standard	0.532	53.17%
1	Poor	0.290	29.00%
2	Good	0.178	17.83%

In [270…
```
cleaner.print_missing_values('Credit_Score')
```

```
Remaining missing values in Credit_Score: 0
```

In [271…
```
visualizer.plot_pie_chart('Credit_Score')
```



The analysis reveals that the majority of customers are categorized under the Standard credit score group, comprising approximately 53.2% of the dataset. The Poor credit score category follows with about 29%, while the Good credit score group has the least representation, totaling around 17.8%. This distribution suggests that most customers possess an average credit score, and there is a relatively smaller proportion of customers with excellent credit scores within this dataset.

The Credit_Score target feature exhibits an imbalanced distribution, with a significantly higher percentage of instances in the Standard category compared to the Good and Poor categories. This imbalance poses a challenge for the predictive model, as it may become biased towards the majority class, resulting in less accurate predictions for the minority classes.

Outliers Investigation

In [274…
```
analyzer.describe_by_type(dtype='numeric')
```

Out[274…

	count	mean	std	min	25%	50%	75%	max
Age	100000.000	34.343	9.812	18.000	26.000	34.000	42.000	118.000
Annual_Income	100000.000	176415.701	1429618.051	7005.930	19457.500	37578.610	72790.920	24198062.000
Monthly_Inhand_Salary	100000.000	5743.259	45814.695	303.645	1625.793	3101.372	5971.780	1990379.583
Num_Bank_Accounts	100000.000	17.091	117.405	0.000	3.000	6.000	7.000	1798.000
Num_Credit_Card	100000.000	22.474	129.057	0.000	4.000	5.000	7.000	1499.000
Interest_Rate	100000.000	72.466	466.423	1.000	8.000	13.000	20.000	5797.000
Num_of_Loan	100000.000	10.762	61.790	0.000	2.000	3.000	6.000	1496.000
Delay_from_due_date	100000.000	21.095	14.823	0.000	10.000	18.000	28.000	67.000
Num_of_Delayed_Payment	100000.000	30.946	217.972	0.000	9.000	15.000	19.000	4397.000
Changed_Credit_Limit	100000.000	10.247	6.768	0.000	4.990	9.250	14.660	36.970
Num_Credit_Inquiries	100000.000	27.754	191.270	0.000	3.000	6.000	9.000	2597.000
Outstanding_Debt	100000.000	1426.220	1155.129	0.230	566.072	1166.155	1945.963	4998.070
Credit_Utilization_Ratio	100000.000	32.285	5.117	20.000	28.053	32.306	36.497	50.000
Credit_History_Age	100000.000	221.195	95.131	1.000	154.000	221.195	292.000	404.000
Total_EMI_per_month	100000.000	1403.118	8306.041	0.000	30.307	69.249	161.224	82331.000
Amount_invested_monthly	100000.000	637.413	1997.035	0.000	77.017	143.128	304.766	10000.000
Monthly_Balance	100000.000	402.413	212.763	0.008	270.914	337.660	468.390	1602.041
Auto Loan	100000.000	0.306	0.461	0.000	0.000	0.000	1.000	1.000
Credit-Builder Loan	100000.000	0.317	0.465	0.000	0.000	0.000	1.000	1.000
Personal Loan	100000.000	0.311	0.463	0.000	0.000	0.000	1.000	1.000
Home Equity Loan	100000.000	0.314	0.464	0.000	0.000	0.000	1.000	1.000
Not Specified	100000.000	0.317	0.465	0.000	0.000	0.000	1.000	1.000
Mortgage Loan	100000.000	0.314	0.464	0.000	0.000	0.000	1.000	1.000
Student Loan	100000.000	0.310	0.463	0.000	0.000	0.000	1.000	1.000
Debt Consolidation Loan	100000.000	0.310	0.463	0.000	0.000	0.000	1.000	1.000
Payday Loan	100000.000	0.319	0.466	0.000	0.000	0.000	1.000	1.000

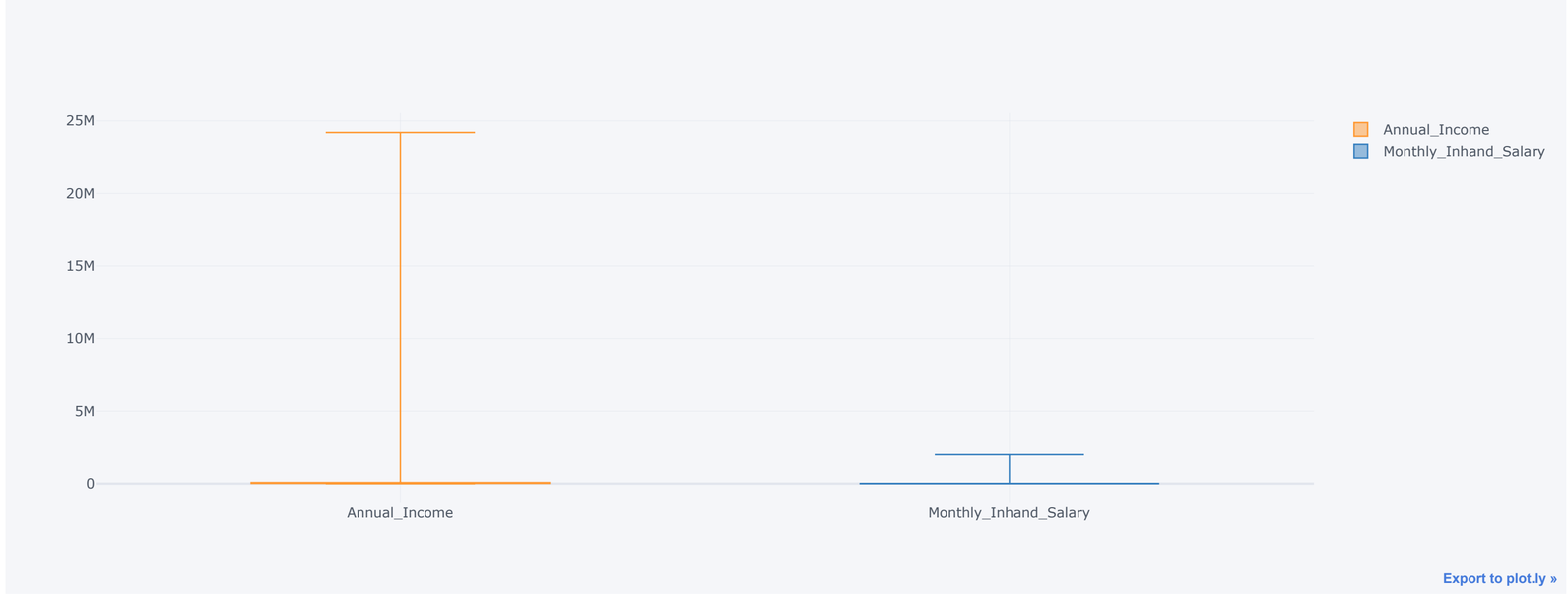
In [275…
```
analyzer.numeric_summary()
```

```
Numerical Features Summary:
_____________________________
Data Types:
Dtype
float64    12
int32       9
int64       5
Name: count, dtype: int64
_____________________________
```

Out[275…

	Dtype	Counts	Nulls	NullPercent	Min	Max	Uniques	UniqueValues
Age	float64	100000	0	0.00%	18.000	118.000	994	-
Annual_Income	float64	100000	0	0.00%	7005.930	24198062.000	13487	-
Monthly_Inhand_Salary	float64	100000	0	0.00%	303.645	1990379.583	22405	-
Num_Bank_Accounts	int64	100000	0	0.00%	0.000	1798.000	942	-
Num_Credit_Card	int64	100000	0	0.00%	0.000	1499.000	1179	-
Interest_Rate	int64	100000	0	0.00%	1.000	5797.000	1750	-
Num_of_Loan	int64	100000	0	0.00%	0.000	1496.000	413	-
Delay_from_due_date	int64	100000	0	0.00%	0.000	67.000	68	-
Num_of_Delayed_Payment	float64	100000	0	0.00%	0.000	4397.000	709	-
Changed_Credit_Limit	float64	100000	0	0.00%	0.000	36.970	3754	-
Num_Credit_Inquiries	float64	100000	0	0.00%	0.000	2597.000	1224	-
Outstanding_Debt	float64	100000	0	0.00%	0.230	4998.070	12203	-
Credit_Utilization_Ratio	float64	100000	0	0.00%	20.000	50.000	100000	-
Credit_History_Age	float64	100000	0	0.00%	1.000	404.000	405	-
Total_EMI_per_month	float64	100000	0	0.00%	0.000	82331.000	14950	-
Amount_invested_monthly	float64	100000	0	0.00%	0.000	10000.000	91050	-
Monthly_Balance	float64	100000	0	0.00%	0.008	1602.041	98859	-
Auto Loan	int32	100000	0	0.00%	0.000	1.000	2	[1, 0]
Credit-Builder Loan	int32	100000	0	0.00%	0.000	1.000	2	[1, 0]
Personal Loan	int32	100000	0	0.00%	0.000	1.000	2	[1, 0]
Home Equity Loan	int32	100000	0	0.00%	0.000	1.000	2	[1, 0]
Not Specified	int32	100000	0	0.00%	0.000	1.000	2	[0, 1]
Mortgage Loan	int32	100000	0	0.00%	0.000	1.000	2	[0, 1]
Student Loan	int32	100000	0	0.00%	0.000	1.000	2	[0, 1]
Debt Consolidation Loan	int32	100000	0	0.00%	0.000	1.000	2	[0, 1]
Payday Loan	int32	100000	0	0.00%	0.000	1.000	2	[0, 1]

In [276…
```
visualizer.plot_boxplot_outliers()
```



In [277…
```
analyzer.describe_by_type(dtype='object')
```

Out[277…

	count	unique	top	freq
Occupation	100000	16	Unknown	7062
Credit_Mix	100000	4	Standard	36479
Payment_of_Min_Amount	100000	3	Yes	59118
Payment_Behaviour	100000	7	Low_spent_Small_value_payments	27649
Credit_Score	100000	3	Standard	53174

In [278…
```
analyzer.object_summary()
```

```
Categorical Features Summary:
_____________________________
Data Types:
Dtype
object    5
Name: count, dtype: int64
_____________________________
```

Out[278…

	Dtype	Counts	Nulls	NullPercent	Top	Frequency	Uniques	UniqueValues
Occupation	object	100000	0	0.00%	Unknown	7062	16	[Scientist, Unknown, Teacher, Engineer, Entrep...
Credit_Mix	object	100000	0	0.00%	Standard	36479	4	[Unknown, Good, Standard, Bad]
Payment_of_Min_Amount	object	100000	0	0.00%	Yes	59118	3	[No, Yes, Unknown]
Payment_Behaviour	object	100000	0	0.00%	Low_spent_Small_value_payments	27649	7	[High_spent_Small_value_payments, Low_spent_La...
Credit_Score	object	100000	0	0.00%	Standard	53174	3	[Good, Standard, Poor]

In [279…
```
analyzer.calculate_skewness(skew_limit=1.5)
```

Out[279…

	Skew
Monthly_Inhand_Salary	32.421
Num_of_Loan	16.606
Num_of_Delayed_Payment	14.842
Annual_Income	12.512
Num_Bank_Accounts	11.202
Num_Credit_Inquiries	9.884
Interest_Rate	9.006
Num_Credit_Card	8.458
Total_EMI_per_month	7.103
Amount_invested_monthly	4.423
Monthly_Balance	1.605

Outliers Observation

The dataset reveals noticeable outliers in features such as Annual Income and Monthly Inhand Salary. Given the financial context of this data, these outliers likely reflect genuine variations in the financial profiles of customers. Therefore, it has been determined that no immediate action will be taken to address these outliers at this stage. Should the model's performance warrant it, subsequent adjustments may be considered to further enhance the outcomes.

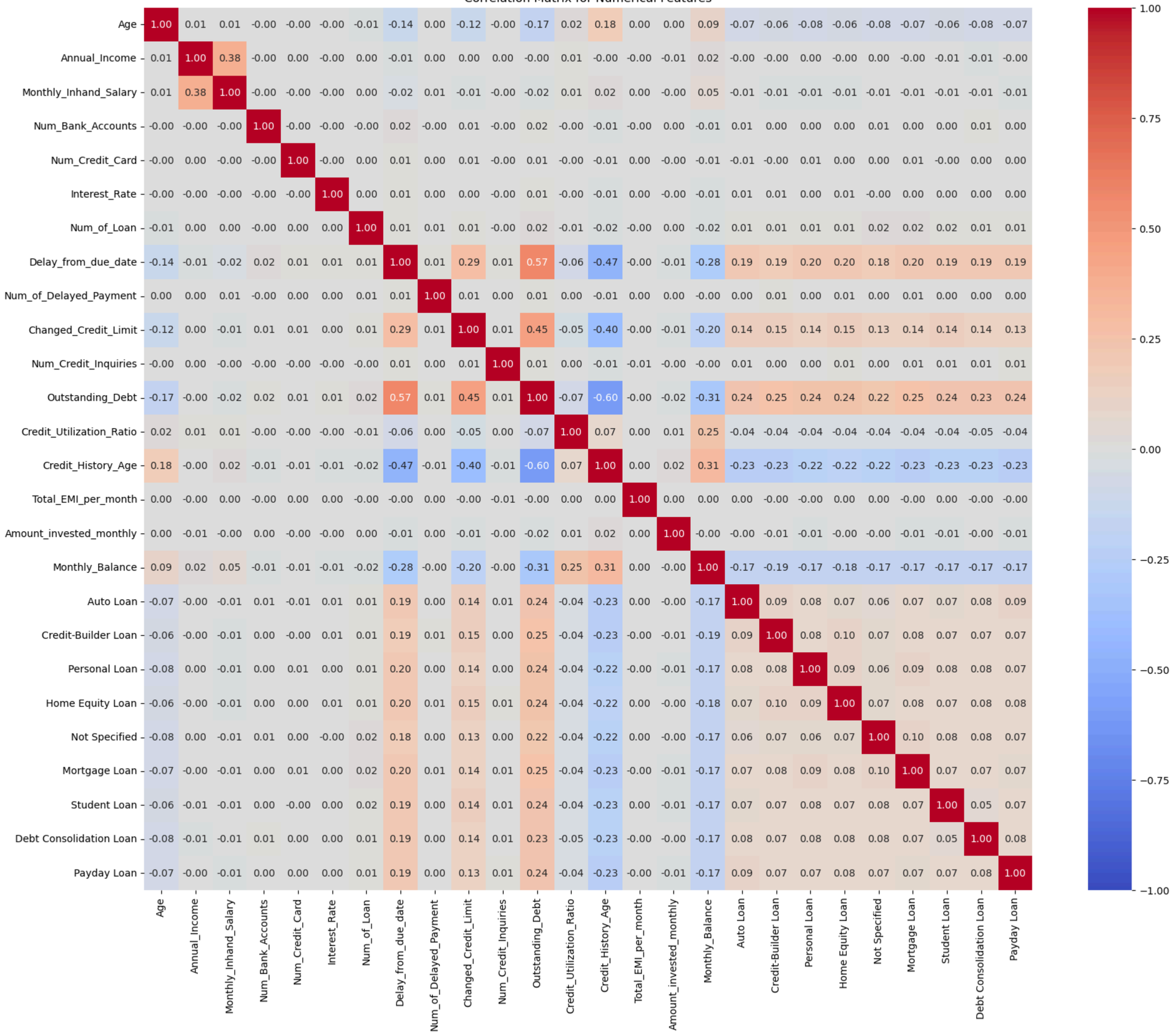
Additionally, the dataset exhibits significant skewness in features like Num_Bank_Accounts, Num_Credit_Inquiries, and Interest_Rate. These skewed distributions indicate that the data is not uniformly distributed, with many values concentrated on one side. The current decision to maintain these outliers and skewed features intact is rooted in a strategic approach to preserve the integrity of the financial profiles and avoid potential loss of critical information at this juncture.

In [281…
```
cleaner.save_and_link_cleaned_data('Credit_Data_Cleaned.csv')
```

Click here to download: Credit_Data_Cleaned.csv

3.2 Correlation Matrix

In [283…
```
visualizer.plot_correlation_matrix(variable_type='number')
```



Correlation Matrix Insights for Numeric Variables

The correlation matrix provides a comprehensive overview of the relationships between numerical features within the dataset, with correlation coefficients ranging from -1 to 1. These coefficients measure the strength and direction of linear relationships between pairs of variables, offering valuable insights into potential associations.

Positive Correlation (0 to 1)

A positive correlation indicates that as one variable increases, the other tends to increase as well. For instance, the correlation coefficient of 0.57 between the "Delay_from_due_date" and "Outstanding_Dept" columns suggests a moderate positive relationship. This implies that as delays from the due date increase, the outstanding debt also tends to increase. Additionally, a positive correlation of 0.45 is observed between "Outstanding_Dept" and "Changed_Credit_Limit," indicating that an increase in outstanding debt is associated with changes in credit limit.

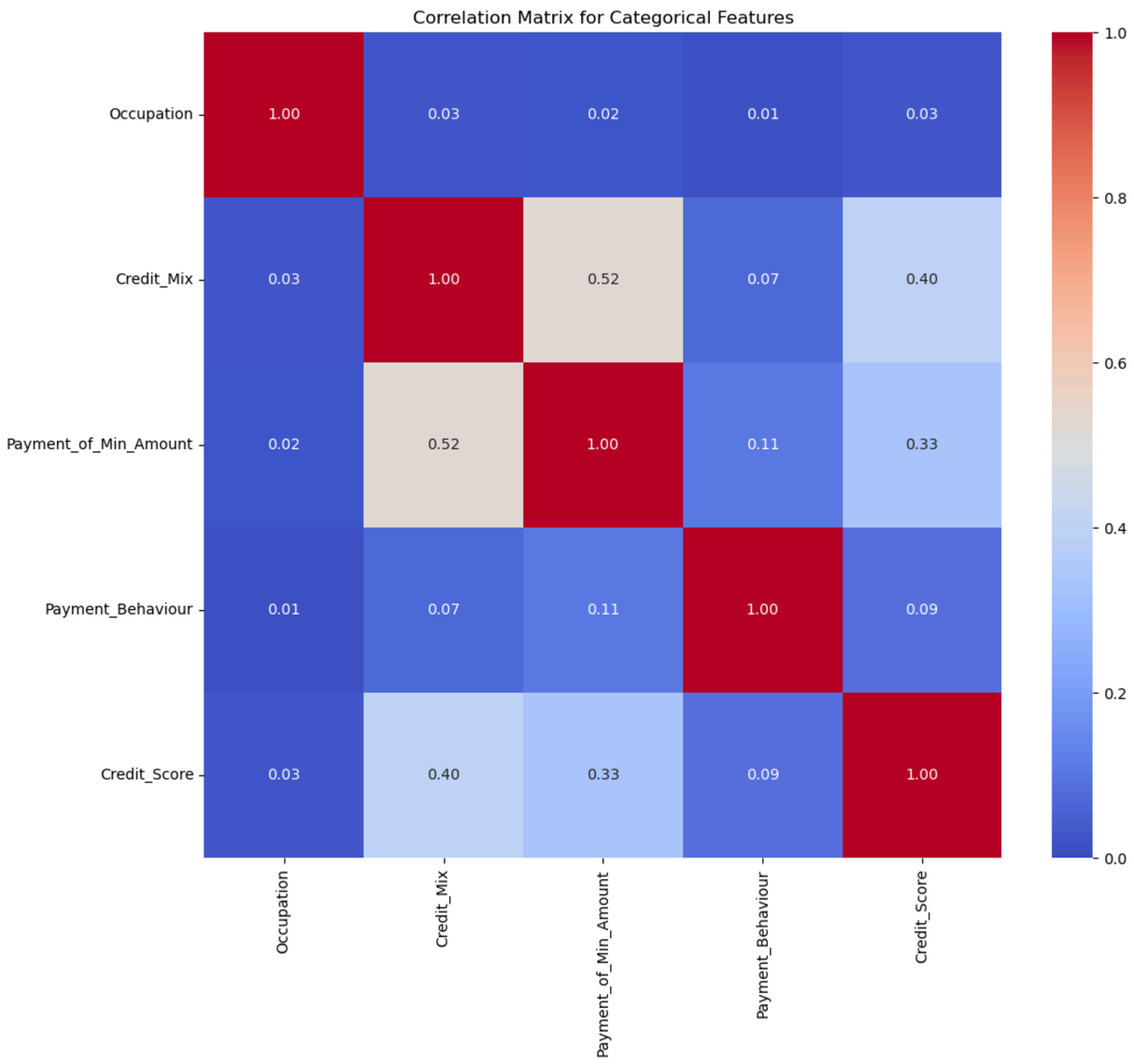
Negative Correlation (0 to -1)

A negative correlation signifies that as one variable increases, the other tends to decrease. For example, the "Delay_from_due_date" and "Credit_History_Age" columns have a correlation coefficient of -0.47. This indicates that as the age of credit history increases, the delays from the due date decrease. Similarly, a negative correlation of -0.31 exists between "Monthly_Balance" and "Outstanding_Dept," suggesting that higher monthly balances are associated with lower outstanding debt.

Weak or No Correlation (around 0)

A coefficient close to 0 indicates a weak or no linear relationship between the variables. The correlation coefficient between "Age" and "Annual_Income" is 0.01, which implies a very weak relationship, indicating that age has minimal or no linear effect on annual income.

In [563…
```
visualizer.plot_correlation_matrix(variable_type='object')
```



Correlation Matrix Insights for Cateogorical Variables

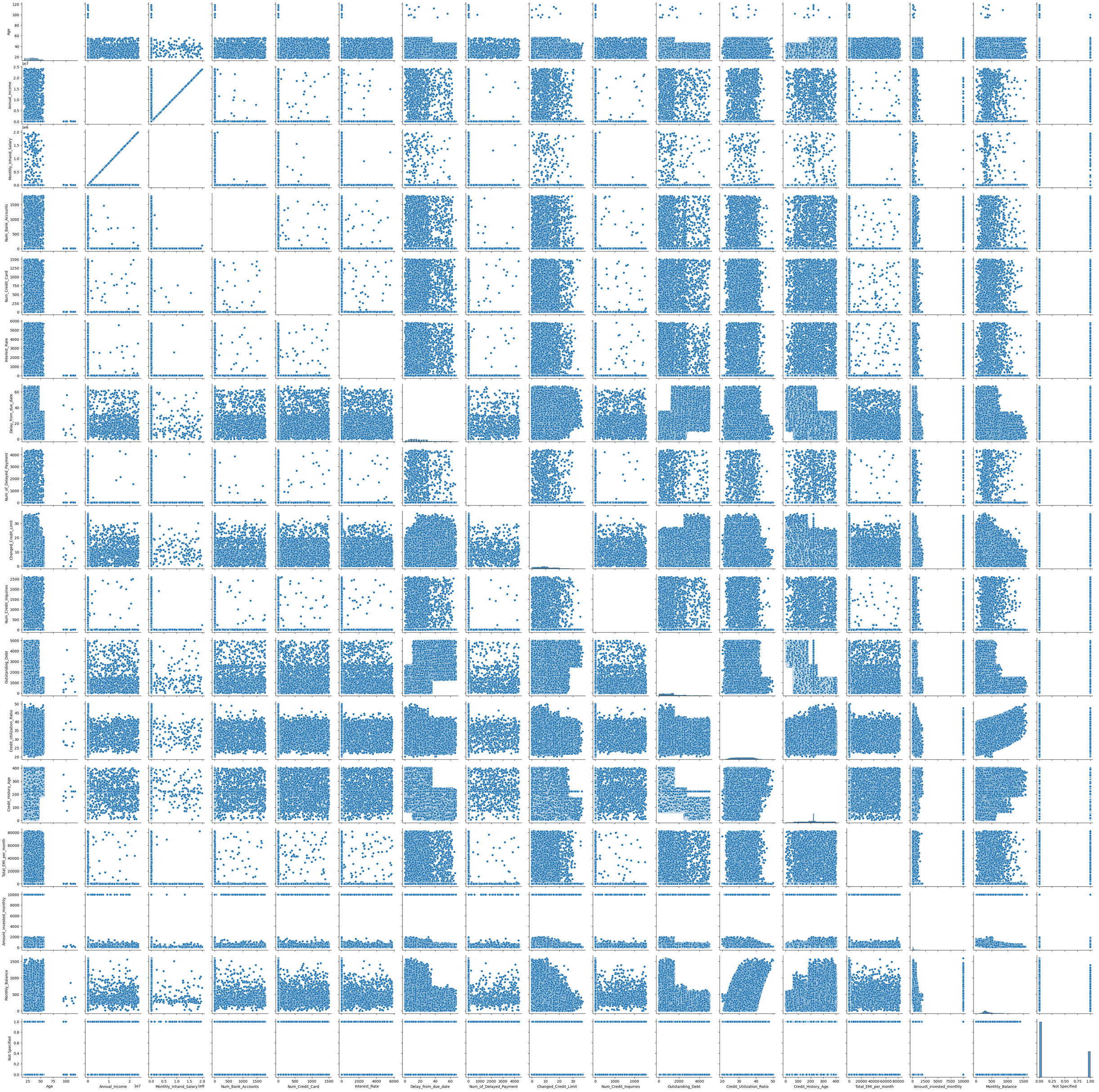
Positive Correlations: There is a moderate positive correlation between Credit_Mix and Payment_of_Min_Amount (0.52), indicating that customers with a diverse credit mix are more likely to make minimum payments.

Negative Correlations: The correlation between Credit_History_Age and Num_of_Delayed_Payment is -0.47, suggesting that individuals with a longer credit history tend to have fewer delayed payments.

Weak Correlations: Some features, such as Interest_Rate and Num_Bank_Accounts, have very weak correlations with other features, indicating minimal linear relationships.

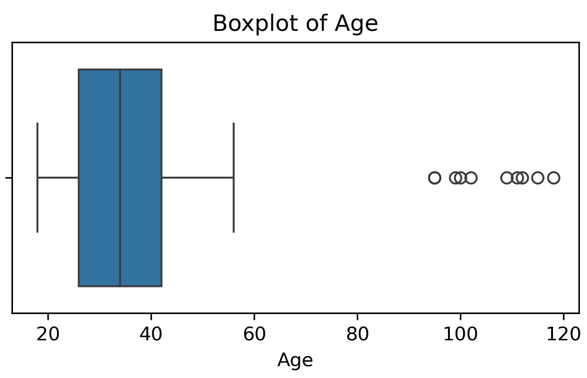
3.3 Data Visualization

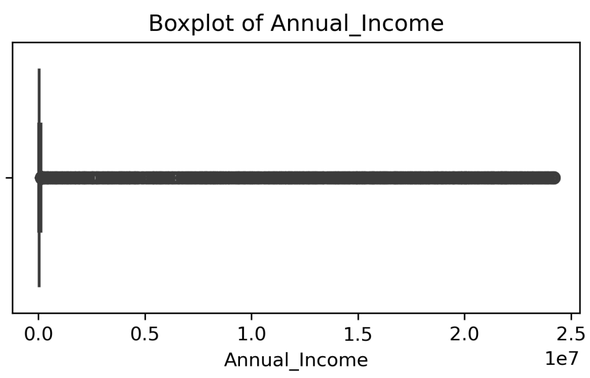
In [363…
```
visualizer.plot_numeric_scatter()
```

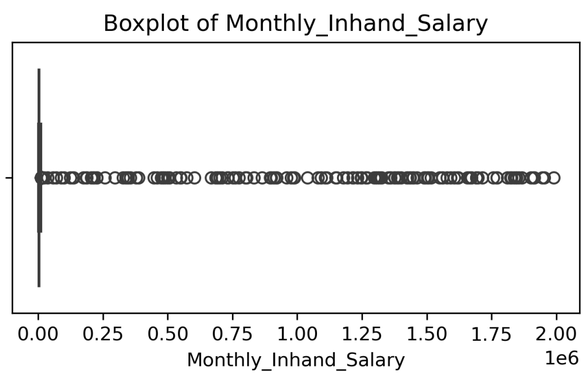


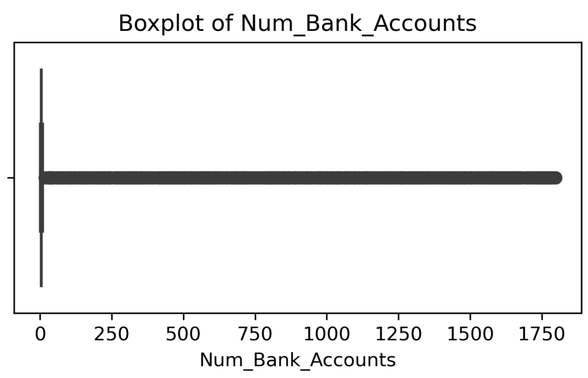
In [364…

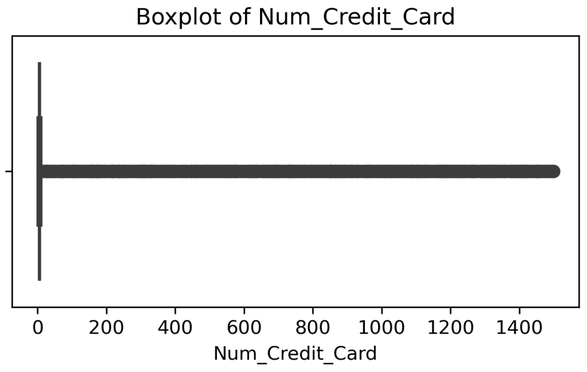
```
visualizer.plot_boxplot_num_cols()
```

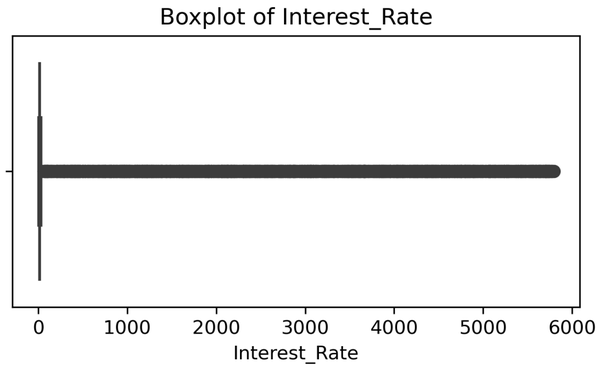


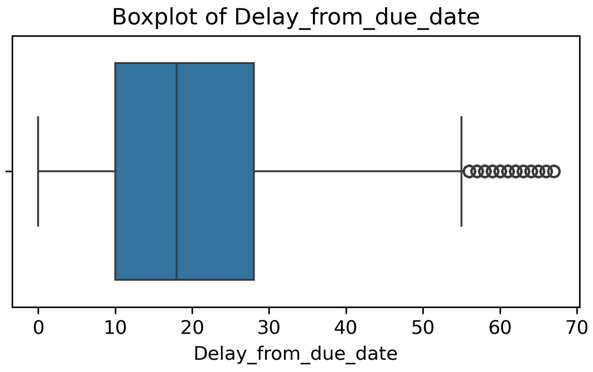


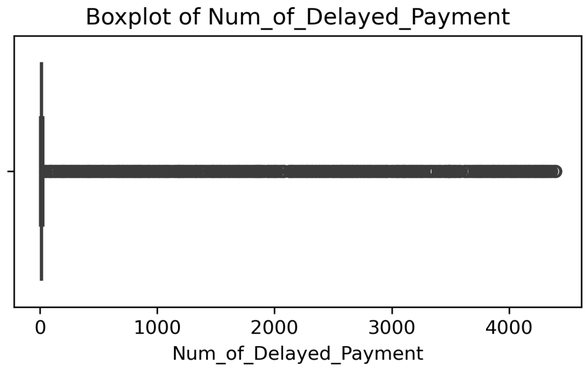


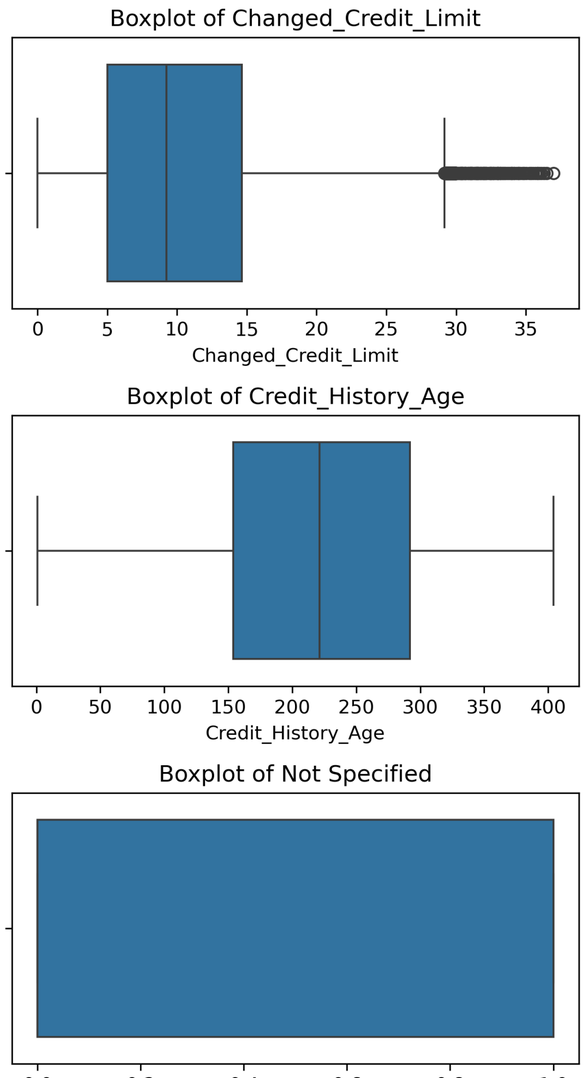


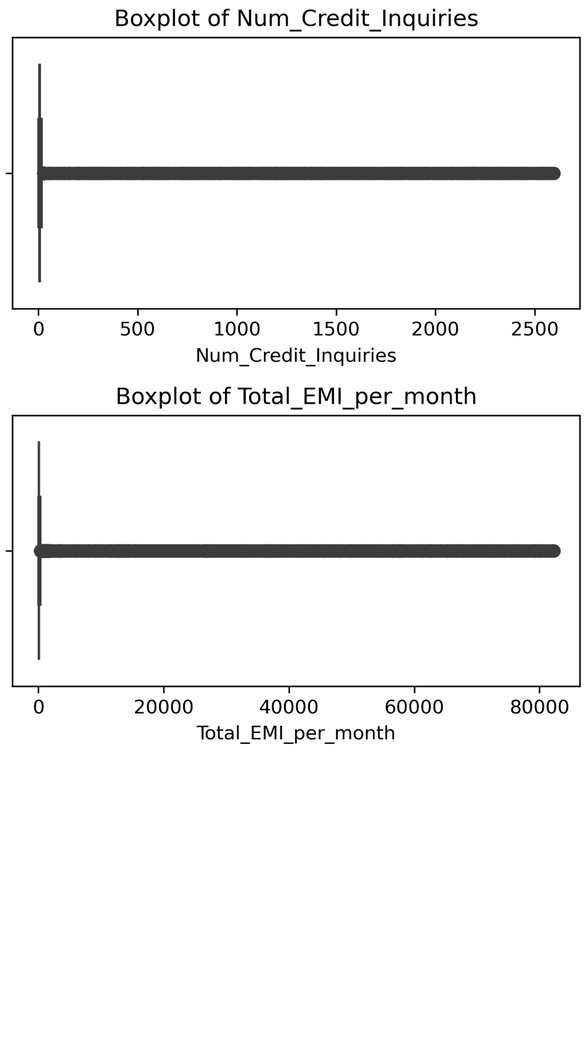


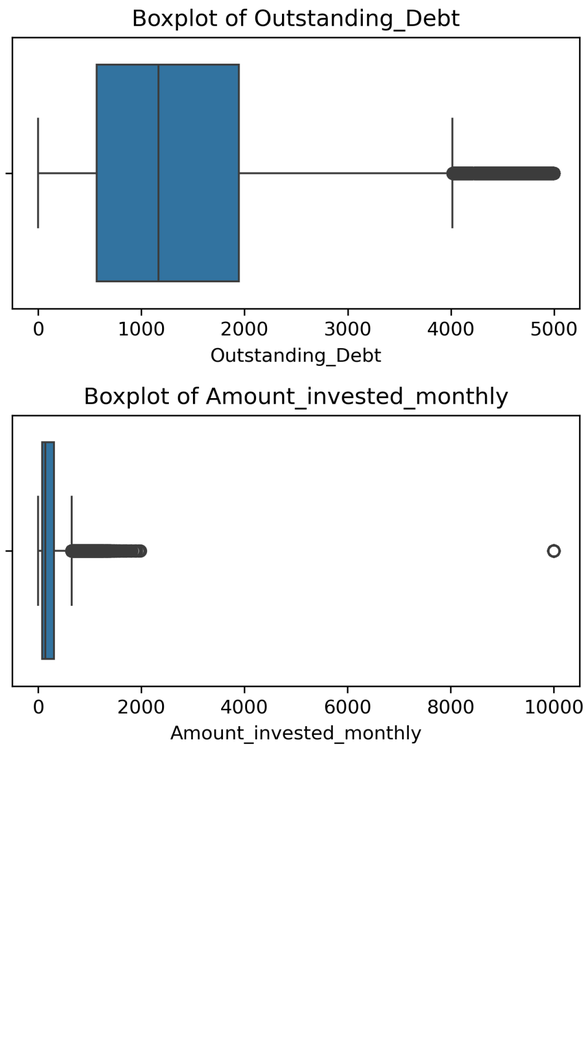


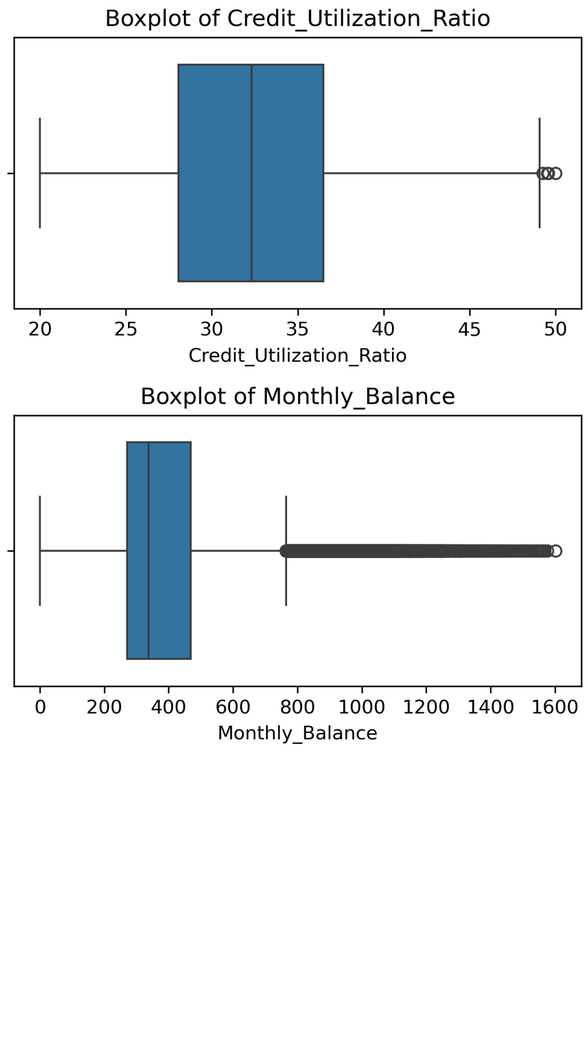




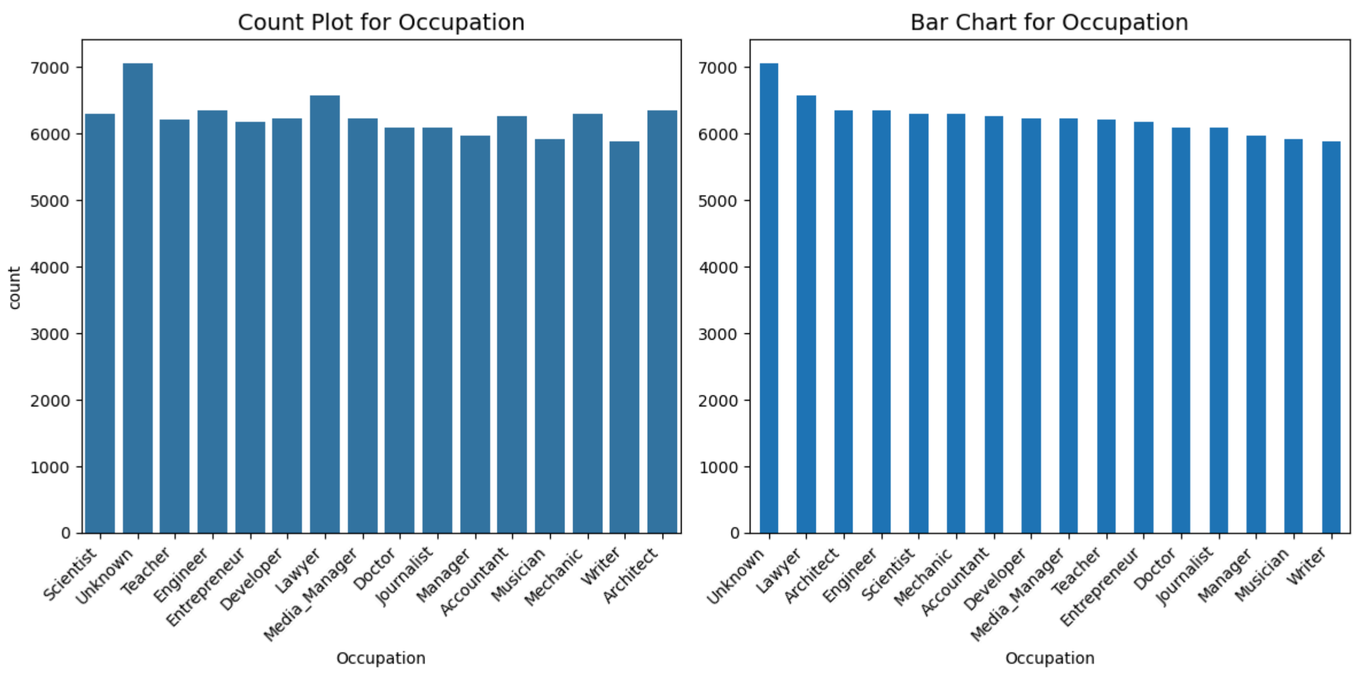


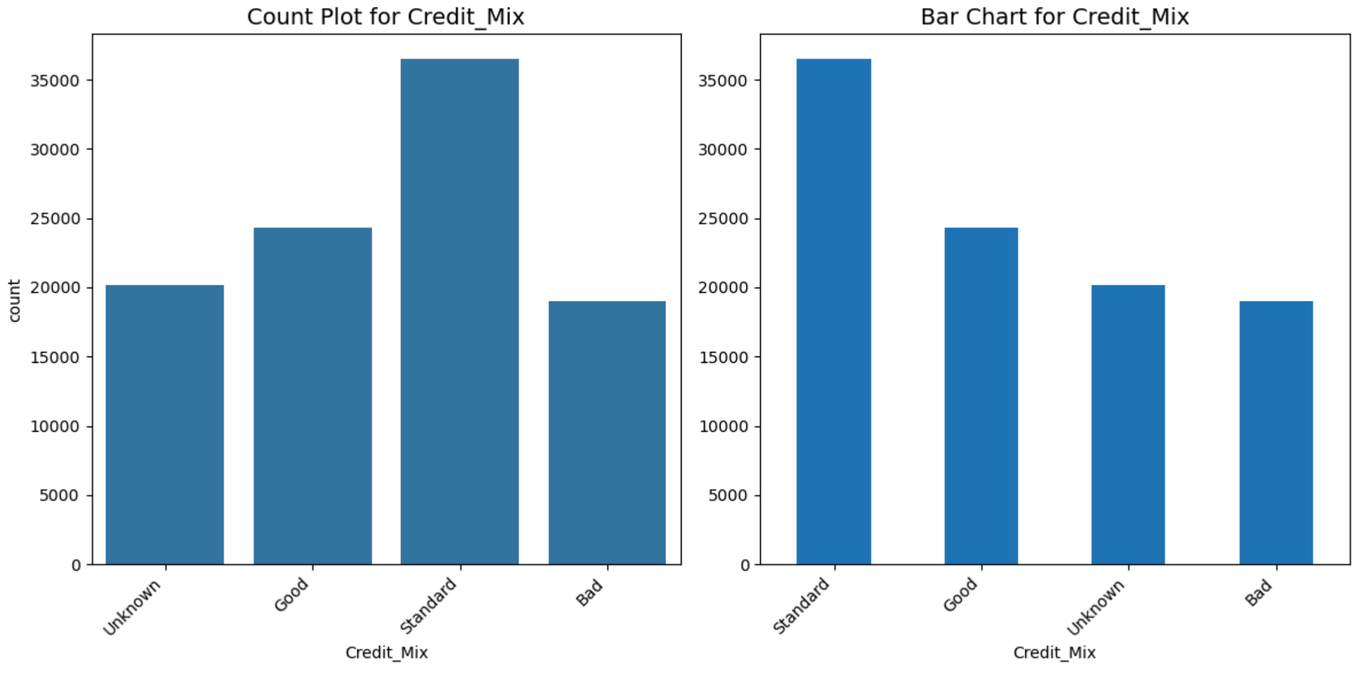


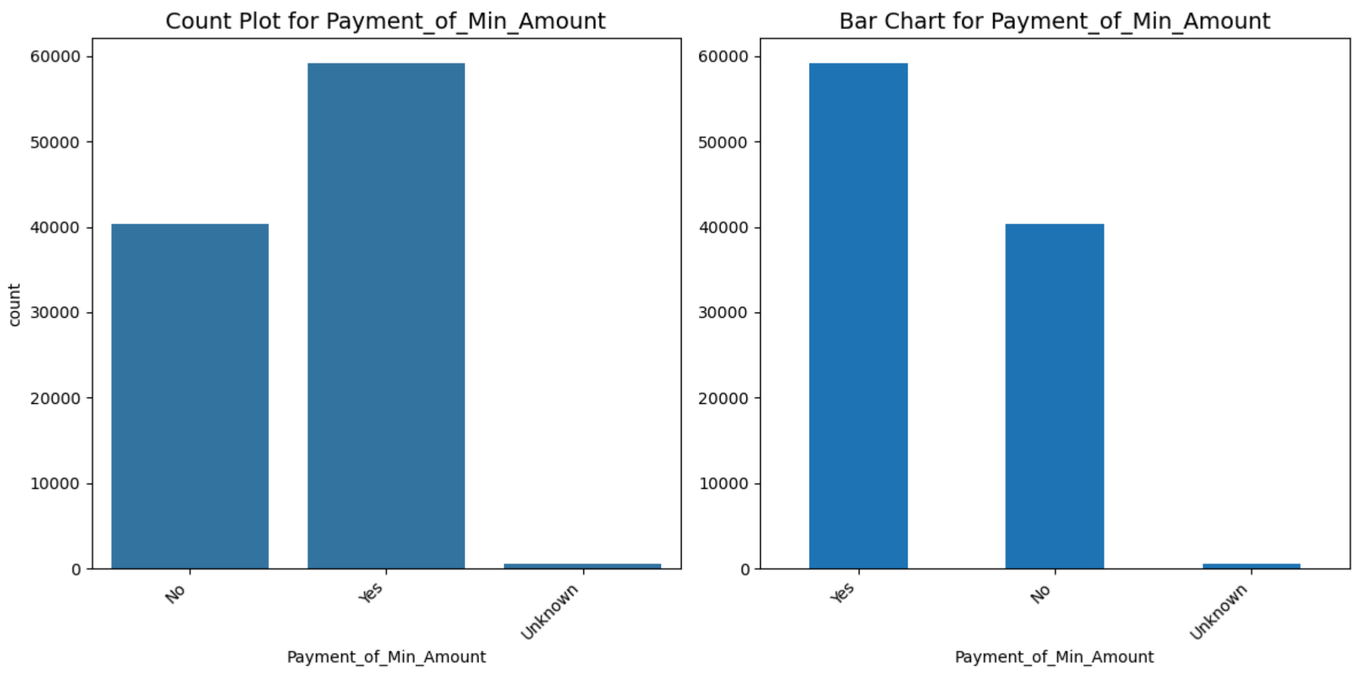


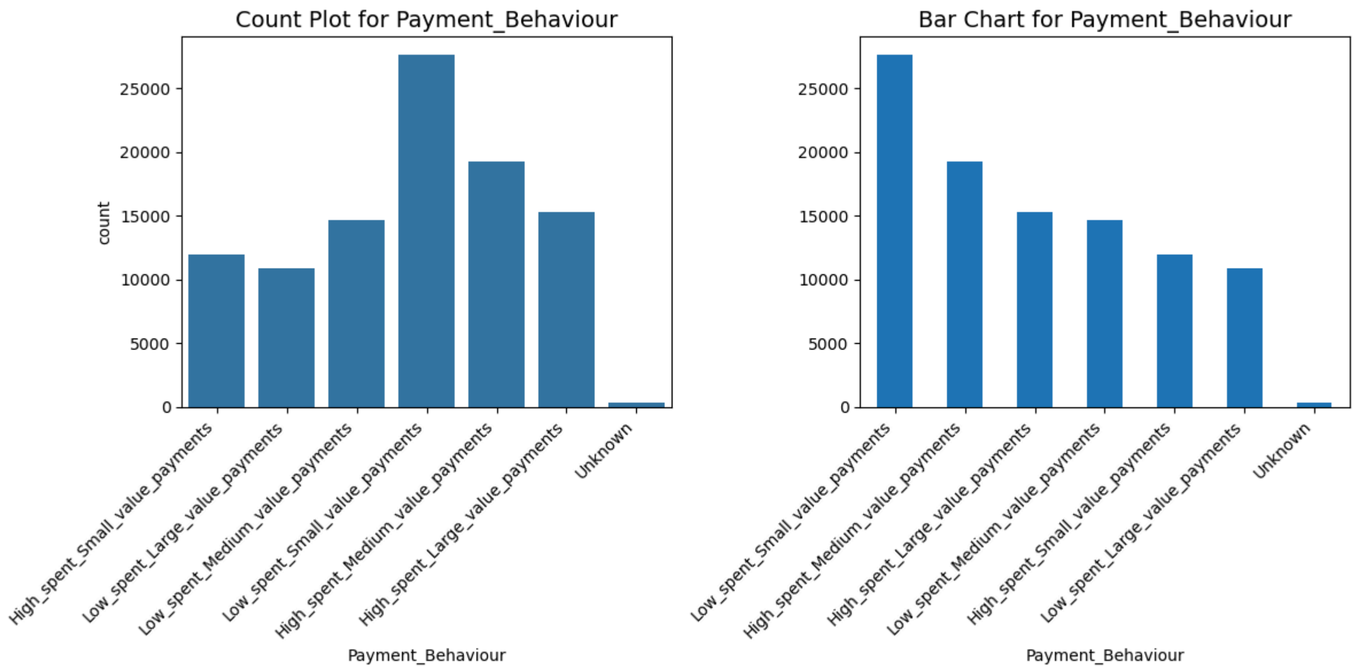


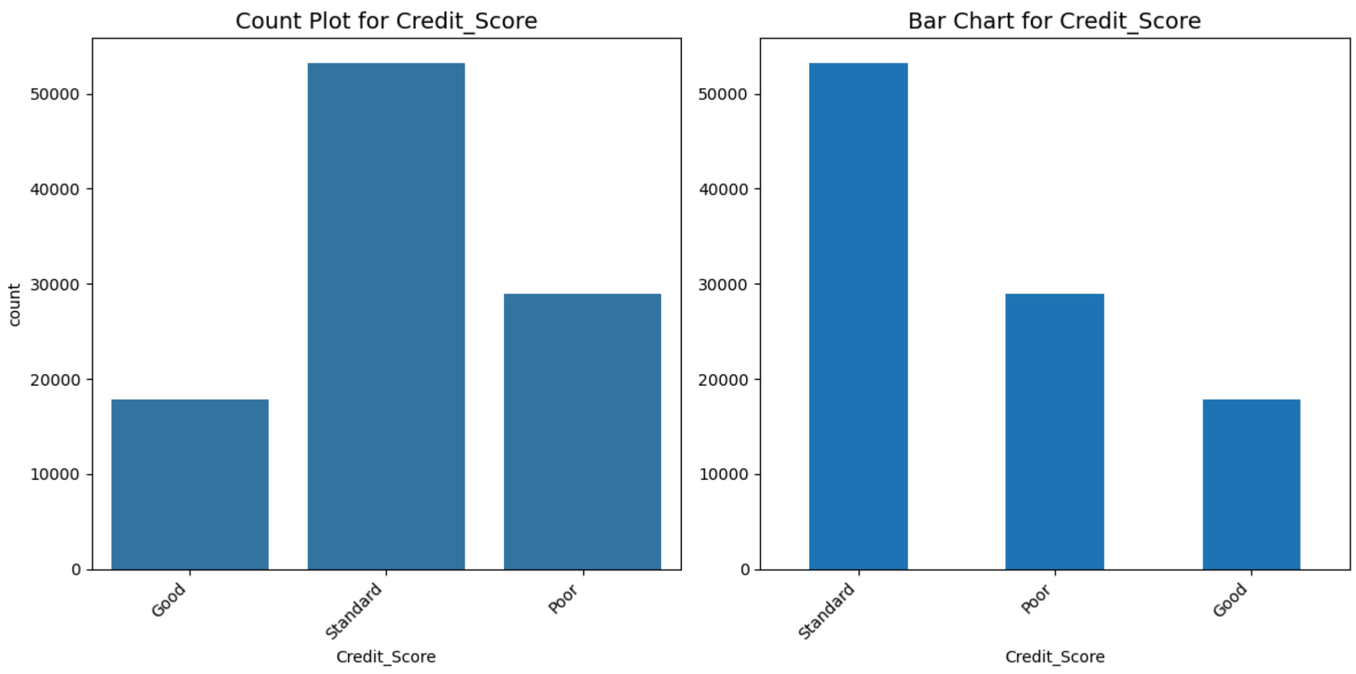
In [365…
```
visualizer.plot_bar_and_count()
```











The charts provide visual distributions for four variables: Credit Mix, Credit Score, Payment Behaviour, and Payment of Minimum Amount.

1. Credit Mix

- The Standard Credit Mix is the most common category with the highest count, indicating that most individuals in the dataset have a balanced mix of credit types.
- Good Credit Mix is the second most frequent, followed by Unknown and Bad, which have similar counts.
- A significant proportion of the data falls into the Standard and Good categories, reflecting healthier credit profiles.

Unknown values indicate missing or undefined credit data, However, it has been determined that no immediate action will be taken to address these outliers at this stage. Should the model's performance warrant it, subsequent adjustments may be considered to further enhance the outcomes.

2. Credit Score

- The Standard Credit Score category has the highest count, suggesting that a majority of individuals have an average credit rating.
- Poor Credit Score is the second largest group, followed by Good, which has the lowest count.
- The distribution indicates that most individuals are concentrated in the Standard or Poor credit score categories.

This suggests room for improvement in financial behavior for a significant portion of the dataset.

3. Payment Behaviour

- High-spent, Small-value payments is the most common category, followed by Low-spent, Medium-value payments.
- High-spent, Large-value payments and Low-spent, Large-value payments are the least common, suggesting fewer individuals engage in high-value transactions.
- The Unknown category has a minimal count, indicating that most individuals have recorded payment behavior.
- Payment behaviors lean toward smaller and medium-value transactions, potentially reflecting financial caution or limited purchasing power.

4. Payment of Minimum Amount

- The Yes category dominates, showing that a majority of individuals pay the minimum required amount.
- No is the second most common, followed by a minimal count in the Unknown category.
- The dominance of the Yes category suggests a preference for maintaining minimum payments rather than paying off balances, which may impact long-term financial health.

Overall Observations:

- Most individuals have a Standard or Good Credit Mix and are meeting the minimum payment requirements.
- A significant portion falls into Poor Credit Score categories, highlighting opportunities to encourage better financial behavior.
- Payment patterns vary across spending levels, with smaller-value payments being the most common.

3.4 Feature Engineering

During the feature engineering process, the 'Type of Loan' column was first extracted and separated into distinct new columns representing each loan category, such as 'Home Loan', 'Car Loan', and 'Personal Loan'. These new columns were populated with binary values indicating the presence or absence of each loan type, allowing for more detailed insights into how different loan categories impact credit scores. Additionally, missing values in the 'Monthly Balance' column were addressed by filling these gaps with the mean monthly balance. This approach ensured data integrity and consistency, thereby enhancing the model's accuracy in predicting credit scores. These steps collectively improved the dataset's quality, leading to more reliable and insightful credit scoring analyses.

In [375…

```
class DataPreprocessor:
    def __init__(self, ordinal_features, onehot_features, ordinal_categories, target_column=None, target_order=None):
        """
        Initializes the DataPreprocessor with features to be encoded, their categories, and an optional target column and order.

        Parameters:
        ordinal_features (list): List of features to be ordinally encoded.
        onehot_features (list): List of features to be one-hot encoded.
        ordinal_categories (list): List of categories for ordinal features.
        target_column (str, optional): The name of the target column, if it needs encoding.
        target_order (list, optional): The order of categories for the target column.
        """
        self.ordinal_features = ordinal_features
        self.onehot_features = onehot_features
        self.ordinal_categories = ordinal_categories
        self.target_column = target_column
        self.target_order = target_order
```

```
    # Define the column transformer with OrdinalEncoder and OneHotEncoder
    self.column_transformer = make_column_transformer(
        (OrdinalEncoder(categories=self.ordinal_categories, dtype=int, handle_unknown="use_encoded_value", unknown_value=-100), self.ordinal_features),  # Ordinal encoding
        (OneHotEncoder(handle_unknown="ignore", dtype=int), self.onehot_features),  # One-hot encoding
        remainder='passthrough',  # Leave other features as is
        verbose_feature_names_out=False
    )

    self.target_encoder = None
    if self.target_column and self.target_order:
        self.target_encoder = OrdinalEncoder(categories=[self.target_order], dtype=int)

def fit_transform(self, X, y):
    """
    Fits the column transformer to the data and transforms it. Also encodes the target column if specified.

    Parameters:
    X (pandas.DataFrame): The feature DataFrame to be encoded.
    y (pandas.Series): The target column to be encoded.

    Returns:
    pd.DataFrame: The transformed and scaled feature DataFrame.
    pd.Series: The encoded target Series.
    """
    if self.target_encoder:
        y = self.target_encoder.fit_transform(y.to_frame(self.target_column)).ravel()
    transformed_X = self.column_transformer.fit_transform(X)
    return pd.DataFrame(transformed_X, columns=self.column_transformer.get_feature_names_out()), pd.Series(y)

def transform(self, X, y=None):
    """
    Transforms the data using the fitted column transformer. Also encodes the target column if specified.

    Parameters:
    X (pandas.DataFrame): The feature DataFrame to be encoded.
    y (pandas.Series, optional): The target column to be encoded.

    Returns:
    pd.DataFrame: The transformed and scaled feature DataFrame.
    pd.Series: The encoded target Series, if y is provided.
    """
    if self.target_encoder and y is not None:
        y = self.target_encoder.transform(y.to_frame(self.target_column)).ravel()
        transformed_X = self.column_transformer.transform(X)
        return pd.DataFrame(transformed_X, columns=self.column_transformer.get_feature_names_out()), pd.Series(y)
    else:
        transformed_X = self.column_transformer.transform(X)
        return pd.DataFrame(transformed_X, columns=self.column_transformer.get_feature_names_out())
```

In [376…

```
ordinal_features = ['Credit_Mix', 'Payment_of_Min_Amount', 'Payment_Behaviour']
onehot_features = ['Occupation']
credit_mix_order = ['Bad', 'Standard', 'Good', 'Unknown']
payment_min_order = ['No', 'Yes']
payment_behaviour_order = ['Low_spent_Small_value_payments',
                           'Low_spent_Medium_value_payments',
                           'Low_spent_Large_value_payments',
                           'High_spent_Small_value_payments',
                           'High_spent_Medium_value_payments',
                           'High_spent_Large_value_payments']
ordinal_categories = [credit_mix_order, payment_min_order, payment_behaviour_order]
target_column = 'Credit_Score'
target_order = ['Poor', 'Standard', 'Good']


preprocessor = DataPreprocessor(ordinal_features, onehot_features, ordinal_categories, target_column, target_order)

X = data.drop(columns=[target_column])
y = data[target_column]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

X_train_encoded, y_train_encoded = preprocessor.fit_transform(X_train, y_train)

X_test_encoded, y_test_encoded = preprocessor.transform(X_test, y_test)
```

In [405…

```
class FeatureSelector:
    def __init__(self, X_train, y_train):
        """
        Initializes the FeatureSelector with training features and target data.

        Parameters:
        X_train (pd.DataFrame): The DataFrame containing the training features.
        y_train (pd.Series): The Series containing the training target.
        """
        self.X_train = X_train
        self.y_train = y_train

    def rfe_selector(self, n_features_to_select=5, plot=False):
        """
        Selects features using Recursive Feature Elimination (RFE) and plots feature ranking.

        Parameters:
        n_features_to_select (int): The number of features to select.
        plot (bool): Whether to plot the feature ranking or not.

        Returns:
        pd.DataFrame: The DataFrame with selected features.
        """
        model = RandomForestClassifier()
        rfe = RFE(model, n_features_to_select=n_features_to_select)
        rfe.fit(self.X_train, self.y_train)
        selected_features = self.X_train.columns[rfe.get_support()]

        if plot:
            self.plot_rfe_ranking(rfe)

        return self.X_train[selected_features]

    def plot_rfe_ranking(self, rfe):
        """
        Plots the ranking of features based on Recursive Feature Elimination (RFE).

        Parameters:
        rfe (RFE): The RFE object after fitting to the data.

        Returns:
        None: The method displays the bar plot of feature rankings.
        """
        ranking = pd.Series(rfe.ranking_, index=self.X_train.columns)
        plt.figure(figsize=(12, 8))
        sns.barplot(x=ranking, y=ranking.index, palette='viridis')
        plt.title('Feature Rankings from RFE')
        plt.xlabel('Ranking')
        plt.ylabel('Features')
        plt.show()

    def plot_feature_importances(self):
        """
        Plots feature importances using a Random Forest model.

        This method fits a Random Forest model to the data and visualizes the feature importances
        using a bar plot.

        Returns:
        None: The method displays the bar plot of feature importances.
        """
        model = RandomForestClassifier()
        model.fit(self.X_train, self.y_train)
        feature_importances = pd.Series(model.feature_importances_, index=self.X_train.columns)
        plt.figure(figsize=(12, 8))
        sns.barplot(x=feature_importances, y=feature_importances.index, palette='viridis')
        plt.title('Feature Importances from Random Forest')
        plt.xlabel('Importance Score')
        plt.ylabel('Features')
        plt.show()
```

3.5 Feature Selection

Recursive Feature Elimination (RFE) Method

Recursive Feature Elimination (RFE) is a widely used feature selection technique designed to enhance the performance of machine learning models by identifying and retaining the most relevant features while eliminating the least important ones. This iterative process builds models recursively, removing less significant features at each step. As Guyon et al. (2002) explain, "RFE is an efficient feature selection algorithm that ranks features by importance, allowing for the elimination of redundant or irrelevant data to improve model accuracy."

The RFE method involves the following steps:

- Model Training: An initial model is trained using all available features.
- Feature Ranking: The importance of each feature is assessed based on the trained model. For instance, in the case of a Random Forest model, feature importance can be measured using criteria like Gini importance or mean decrease in impurity.
- Feature Elimination: The least important feature(s) is/are removed from the feature set.
- Recursive Process: The process is repeated recursively, training the model with the remaining features and eliminating the least important ones, until the desired number of features is reached.
- Recursive Feature Elimination (RFE) is especially valuable for models that perform better with a reduced number of features, as it minimizes overfitting and enhances generalization. Additionally, RFE reduces computational costs and improves model interpretability. As Guyon et al. (2002) note, "By iteratively removing less important features, RFE not only improves the performance of the model but also simplifies it, leading to better generalization and interpretability."

Implementation of RFE in the FeatureSelector Class

- The FeatureSelector class leverages the Recursive Feature Elimination (RFE) method to identify important features in a dataset. Using a Random Forest model as the estimator, its rfe_selector method selects a specified number of features, while the plot_rfe_ranking method visualizes feature rankings. As Guyon et al. (2002) explain, "RFE simplifies models by iteratively removing less relevant features, leading to better performance and interpretability."
- Here is a detailed breakdown of the functionality:
- Initialization: The class is initialized with the training features (X_train) and the target (y_train).
- The rfe_selector method facilitates feature selection using Recursive Feature Elimination (RFE), allowing users to specify the number of features to retain. Additionally, it can generate visualizations of feature rankings. As Guyon et al. (2002) highlight, "Recursive Feature Elimination improves model performance and interpretability by iteratively identifying and ranking the most relevant features."

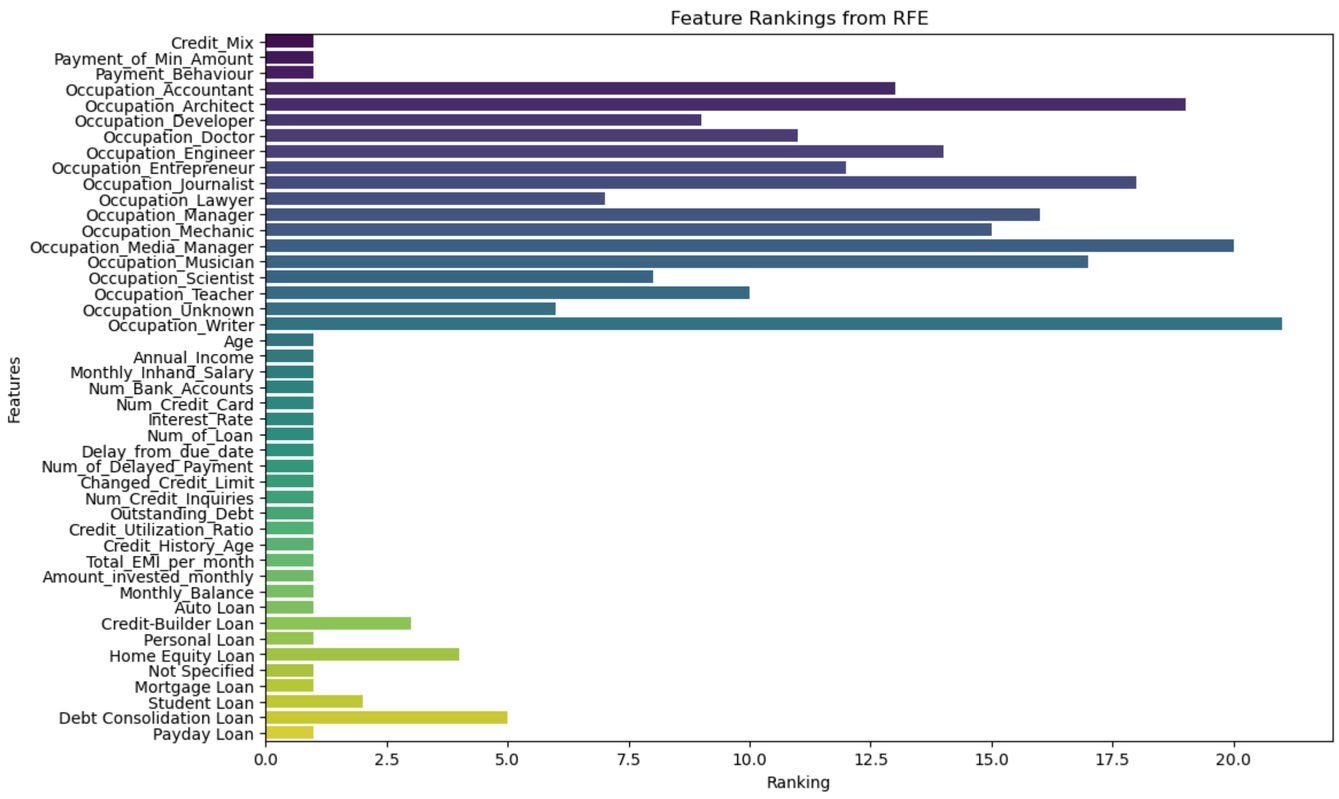
- The plot_rfe_ranking method visualizes feature rankings derived from the Recursive Feature Elimination (RFE) process, offering an intuitive understanding of the most important features. As Guyon et al. (2002) note, "Visualization of feature rankings aids in comprehending the relevance of variables, enhancing model transparency and interpretability."

Practical Application and Objective

- In our specific application, after encoding the categorical features, the number of columns increased to 45. To streamline the model and reduce computational costs, the aim is selecting the top 25 features. The goal was to simplify the model and decrease the time cost without compromising accuracy. By applying the RFE method via the FeatureSelector class, we identified and retained the most relevant features, thereby enhancing the efficiency and performance of the machine learning model. This approach not only helped in managing high-dimensional data but also ensured that the model remained interpretable and computationally feasible.
- This methodological approach highlights the significance of thoughtful feature selection in the machine learning pipeline, striving to balance model complexity, predictive accuracy, and computational efficiency. As Guyon et al. (2002) explain, "Feature selection reduces model complexity while maintaining or improving accuracy, ensuring computational efficiency and better generalization."

In [409…
```
# Initialize the FeatureSelector
selector = FeatureSelector(X_train_encoded, y_train_encoded)
```

In [411…
```
selected_features_rfe = selector.rfe_selector(n_features_to_select=25, plot=True)
```



In [415…
```
selected_features_rfe.columns.to_list()
```

Out[415…
```
['Credit_Mix',
 'Payment_of_Min_Amount',
 'Payment_Behaviour',
 'Age',
 'Annual_Income',
 'Monthly_Inhand_Salary',
 'Num_Bank_Accounts',
 'Num_Credit_Card',
 'Interest_Rate',
 'Num_of_Loan',
 'Delay_from_due_date',
 'Num_of_Delayed_Payment',
 'Changed_Credit_Limit',
 'Num_Credit_Inquiries',
 'Outstanding_Debt',
 'Credit_Utilization_Ratio',
 'Credit_History_Age',
 'Total_EMI_per_month',
 'Amount_invested_monthly',
 'Monthly_Balance',
 'Auto Loan',
 'Personal Loan',
 'Not Specified',
 'Mortgage Loan',
 'Payday Loan']
```

3.6 Class Balancing & Scaling & Encoding

DataPreprocessor Class

The DataPreprocessor class is designed to handle various preprocessing tasks required for preparing a dataset for machine learning models. Here is a detailed overview of what we accomplish within this class:

1. Initialization: The class is initialized with lists of features to be ordinally encoded and one-hot encoded, along with their corresponding categories. Additionally, it takes an optional target column and its categories if encoding is required.
2. Column Transformer Setup: We define a column transformer using make_column_transformer, which handles the encoding processes:
    - Ordinal Encoding: This process transforms categorical features with a meaningful order into numerical values.
    - One-Hot Encoding: This process converts categorical features without a meaningful order into binary vectors.
3. Target Encoding: If a target column is specified, it is encoded using OrdinalEncoder to transform target categories into numerical values.
4. Fit and Transform: The fit_transform method fits the column transformer to the training data and transforms it. It also encodes the target column if specified.
5. Transform: The transform method applies the transformations to new data (such as test data) using the previously fitted column transformer.

Handling Imbalance in the Target Column

To address the imbalance in the target column (Credit_Score), we utilize the stratify parameter when splitting the dataset into training and test sets. This parameter ensures that the distribution of the target classes remains consistent in both training and test sets, thereby preventing any class imbalance issues that could affect the model's performance.

Scaling Considerations

From a business perspective, our dataset contains numerous financial variables. We decided not to apply scaling (e.g., using StandardScaler) because scaling could introduce bias, especially in the presence of outliers. These outliers are often significant in financial data as they represent extreme values that could be crucial for decision-making. By preserving the original scale of the financial features, we ensure that the model accurately captures the variability and significance of these values.

4. Modeling

The ModelSelector class streamlines the evaluation and optimization of machine learning models by managing key processes such as cross-validation, hyperparameter tuning, and performance visualization. As Bergstra and Bengio (2012) state, "Systematic approaches to model selection and hyperparameter optimization are essential for achieving robust performance in machine learning models."

Initialization

The class initializes with training and test datasets, along with a target encoder, ensuring all components are prepared for model evaluation. It also defines a dictionary of machine learning models, including RandomForest, CatBoost, and LightGBM. As Buitinck et al. (2013) note, "A well-structured initialization process is essential for reproducibility and streamlined model evaluation in machine learning workflows."

Model Evaluation with Stratified K-Fold Cross-Validation

A key feature of the ModelSelector class is the use of Stratified K-Fold Cross-Validation, which ensures class distribution is preserved across folds. This stratified approach is particularly effective for evaluating models on imbalanced datasets. As Kuhn and Johnson (2013) explain, "Stratified resampling methods provide more reliable estimates of model performance by maintaining the balance of the target variable in each fold."

During evaluation, each model is trained on multiple training folds and validated on the corresponding validation fold. The mean F1-score across all folds is calculated to summarize performance, ensuring robustness. As Kuhn and Johnson (2013) highlight, "Cross-validation techniques provide comprehensive dataset coverage, leading to more reliable and generalized performance estimates."

Radar Chart Visualization

Radar charts are used to compare model performance across metrics such as F1-score, accuracy, precision, and recall. These visualizations provide an intuitive understanding of each model's strengths and weaknesses. As Few (2012) notes, "Well-designed visualizations help simplify complex data, enabling clearer insights into comparative performance."

Hyperparameter Tuning with Optuna

The best-performing model, identified by the highest mean F1-score, undergoes hyperparameter tuning using Optuna. This framework efficiently explores the hyperparameter search space, running multiple trials to identify the optimal configuration that maximizes performance. As Bergstra and Bengio (2012) state, "Systematic hyperparameter optimization significantly improves model performance by identifying the best parameter combinations within a defined search space."

The tuned model is then trained on the entire training set and evaluated on the test set to obtain the final F1-score. To ensure the reliability of the tuned model, its F1-score is compared with the best default F1-score obtained during initial evaluation. If the tuned model's F1-score is lower, the model with default parameters is used instead.

Final Model Performance Visualization

The final step involves visualizing the model's performance using confusion matrices and feature importance plots. These visualizations provide a comprehensive understanding of the model's predictive power and the significance of each feature. Confusion matrices offer insights into the classification accuracy, while feature importance plots highlight the most influential features driving the model's predictions.

In [421…

```python
class ModelSelector:
    def __init__(self, X_train, y_train, X_test, y_test, target_encoder):
        """
        Initializes the ModelSelector class with training and test datasets, along with a target encoder.

        Parameters:
        X_train (pd.DataFrame): The DataFrame containing the training features.
        y_train (pd.Series): The Series containing the training target.
        X_test (pd.DataFrame): The DataFrame containing the test features.
        y_test (pd.Series): The Series containing the test target.
        target_encoder (OrdinalEncoder): The encoder used to decode target labels.
        """
        self.X_train = X_train
        self.y_train = y_train
        self.X_test = X_test
        self.y_test = y_test
        self.target_encoder = target_encoder
        self.models = {
            'RandomForest': RandomForestClassifier(),
            'CatBoost': CatBoostClassifier(verbose=0),
            'LightGBM': LGBMClassifier(force_col_wise=True)
        }
        self.best_model_name = None
        self.best_model = None
        self.best_f1_score = 0
        self.default_best_f1_score = 0

    def evaluate_models(self):
        """
        Evaluates multiple models using Stratified K-Fold Cross-Validation and records their performance metrics.

        Returns:
        dict: A dictionary containing the mean F1-scores of the evaluated models.
        """
        results = {}
        for name, model in self.models.items():
            skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
            f1_scores = []

            for train_index, val_index in skf.split(self.X_train, self.y_train):
                X_train_fold, X_val_fold = self.X_train.iloc[train_index], self.X_train.iloc[val_index]
                y_train_fold, y_val_fold = self.y_train.iloc[train_index], self.y_train.iloc[val_index]

                model.fit(X_train_fold, y_train_fold)
                y_pred_fold = model.predict(X_val_fold)
                f1 = f1_score(y_val_fold, y_pred_fold, average='weighted')
                f1_scores.append(f1)

            mean_f1 = np.mean(f1_scores)
            results[name] = mean_f1

            if mean_f1 > self.best_f1_score:
                self.best_f1_score = mean_f1
                self.best_model_name = name
                self.best_model = model

            y_pred = model.predict(self.X_test)  # Ensure y_pred is 1-dimensional
            y_pred = y_pred.flatten()

            print(f'Model: {name}, Mean F1-Score: {mean_f1}')
            print('Confusion Matrix:')
            print(confusion_matrix(self.y_test, y_pred))
            print('Classification Report:')
            print(classification_report(self.y_test, y_pred))

            # Plot radar chart for each model's performance
            self.plot_radar_chart(name, mean_f1, y_pred)

        self.default_best_f1_score = self.best_f1_score
        print(f'Best Model: {self.best_model_name}, Best Mean F1-Score: {self.best_f1_score}')
        return results

    def plot_radar_chart(self, model_name, f1, y_pred):
        """
        Plots a radar chart for the performance metrics of the evaluated model.

        Parameters:
        model_name (str): The name of the model being evaluated.
        f1 (float): The F1-score of the model.
        y_pred (array-like): The predicted labels from the model.
        """
        # Calculate performance metrics
        accuracy = np.mean(self.y_test == y_pred)
        precision = precision_score(self.y_test, y_pred, average='weighted')
        recall = recall_score(self.y_test, y_pred, average='weighted')

        metrics = {'F1-Score': f1, 'Accuracy': accuracy, 'Precision': precision, 'Recall': recall}

        labels = list(metrics.keys())
        values = list(metrics.values())

        num_vars = len(labels)

        # Create radar chart
        angles = np.linspace(0, 2 * np.pi, num_vars, endpoint=False).tolist()
        values += values[:1]
        angles += angles[:1]

        fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(polar=True))

        ax.fill(angles, values, color='b', alpha=0.25)
        ax.plot(angles, values, color='b', linewidth=2)

        ax.set_yticklabels([])
        ax.set_xticks(angles[:-1])
        ax.set_xticklabels(labels)

        plt.title(f'{model_name} Performance Metrics', size=15, color='b', y=1.1)
        plt.show()

    def optimize_best_model(self):
        """
        Optimizes the hyperparameters of the best-performing model using Optuna.

        Returns:
        dict: The best hyperparameters found during optimization.
        """
        def objective(trial):
            if self.best_model_name == 'RandomForest':
                n_estimators = trial.suggest_int('n_estimators', 50, 300)
                max_depth = trial.suggest_int('max_depth', 5, 50)
                min_samples_split = trial.suggest_int('min_samples_split', 2, 20)
                min_samples_leaf = trial.suggest_int('min_samples_leaf', 1, 20)
                model = RandomForestClassifier(
                    criterion='log_loss',
                    n_estimators=n_estimators,
                    max_depth=max_depth,
                    min_samples_split=min_samples_split,
                    min_samples_leaf=min_samples_leaf,
                    random_state=42
                )
                model.fit(self.X_train, self.y_train)
                y_pred = model.predict(self.X_test)
                return f1_score(self.y_test, y_pred, average='weighted')
            elif self.best_model_name == 'CatBoost':
                depth = trial.suggest_int('depth', 3, 10)
                learning_rate = trial.suggest_loguniform('learning_rate', 1e-3, 0.3)
                iterations = trial.suggest_int('iterations', 100, 1000)
                model = CatBoostClassifier(
                    depth=depth,
                    learning_rate=learning_rate,
                    iterations=iterations,
                    verbose=0,
```

```
            random_state=42
        )
        model.fit(self.X_train, self.y_train)
        y_pred = model.predict(self.X_test)
        return f1_score(self.y_test, y_pred, average='weighted')
    elif self.best_model_name == 'LightGBM':
        num_leaves = trial.suggest_int('num_leaves', 20, 150)
        learning_rate = trial.suggest_loguniform('learning_rate', 1e-3, 0.3)
        n_estimators = trial.suggest_int('n_estimators', 50, 300)
        model = LGBMClassifier(
            num_leaves=num_leaves,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            random_state=42
        )
        model.fit(self.X_train, self.y_train)
        y_pred = model.predict(self.X_test)
        return f1_score(self.y_test, y_pred, average='weighted')

# Create and optimize study
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=25)

best_params = study.best_params
if self.best_model_name == 'RandomForest':
    self.best_model = RandomForestClassifier(**best_params, random_state=42)
elif self.best_model_name == 'CatBoost':
    self.best_model = CatBoostClassifier(**best_params, verbose=0, random_state=42)
elif self.best_model_name == 'LightGBM':
    self.best_model = LGBMClassifier(**best_params, force_col_wise=True, random_state=42)

self.best_model.fit(self.X_train, self.y_train)
y_pred = self.best_model.predict(self.X_test)
y_pred = y_pred.flatten()
tuned_f1 = f1_score(self.y_test, y_pred, average='weighted')
print(f'Final Model: {self.best_model_name}, Final F1-Score: {tuned_f1}')

# Revert to default model if tuned model's performance is worse
if tuned_f1 < self.default_best_f1_score:
    print(f'Tuned F1-Score is lower than the default. Using default model parameters.')
    self.best_model = self.models[self.best_model_name]
    self.best_model.fit(self.X_train, self.y_train)
    y_pred = self.best_model.predict(self.X_test)
    tuned_f1 = f1_score(self.y_test, y_pred, average='weighted')
    print(f'Final Model with Default Parameters: {self.best_model_name}, Final F1-Score: {tuned_f1}')

print('Confusion Matrix:')
print(confusion_matrix(self.y_test, y_pred))
print('Classification Report:')
print(classification_report(self.y_test, y_pred))
print(f'Best hyperparameters for {self.best_model_name}: {best_params}')
return best_params

def plot_best_model_performance(self):
    """
    Plots the performance metrics of the best model, including the confusion matrix and feature importances.

    This method visualizes the confusion matrix and the feature importances for the best-performing model,
    providing insights into the model's performance and the significance of each feature.

    Returns:
    None
    """
    y_pred = self.best_model.predict(self.X_test)
    y_true_decoded = self.target_encoder.inverse_transform(self.y_test.values.reshape(-1, 1)).flatten()
    y_pred_decoded = self.target_encoder.inverse_transform(y_pred.reshape(-1, 1)).flatten()

    # Plot confusion matrix
    conf_matrix = confusion_matrix(y_true_decoded, y_pred_decoded, labels=self.target_encoder.categories_[0])
    plt.figure(figsize=(10, 7))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='viridis',
                xticklabels=self.target_encoder.categories_[0],
                yticklabels=self.target_encoder.categories_[0])
    plt.title(f'Confusion Matrix for {self.best_model_name}')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()

    # Plot feature importances if applicable
    if self.best_model_name in ['RandomForest', 'LightGBM']:
        importances = self.best_model.feature_importances_
        indices = importances.argsort()[::-1]
        features = self.X_train.columns

        plt.figure(figsize=(12, 8))
        plt.title(f'Feature Importances for {self.best_model_name}')
        sns.barplot(x=importances[indices], y=features[indices], palette='viridis')
        plt.xlabel('Importance Score')
        plt.ylabel('Features')
        plt.show()
    elif self.best_model_name == 'CatBoost':
        importances = self.best_model.get_feature_importance()
        indices = importances.argsort()[::-1]
        features = self.X_train.columns

        plt.figure(figsize=(12, 8))
        plt.title(f'Feature Importances for {self.best_model_name}')
        sns.barplot(x=importances[indices], y=features[indices], palette='viridis')
        plt.xlabel('Importance Score')
        plt.ylabel('Features')
        plt.show()
```

4.1 Model Selection

This study employed a comprehensive approach to model selection using three prominent machine learning algorithms: Random Forest, CatBoost, and LightGBM. Each of these models was evaluated based on their F1-Score, a metric that balances precision and recall, thus providing a robust assessment of the model's performance in handling imbalanced datasets.

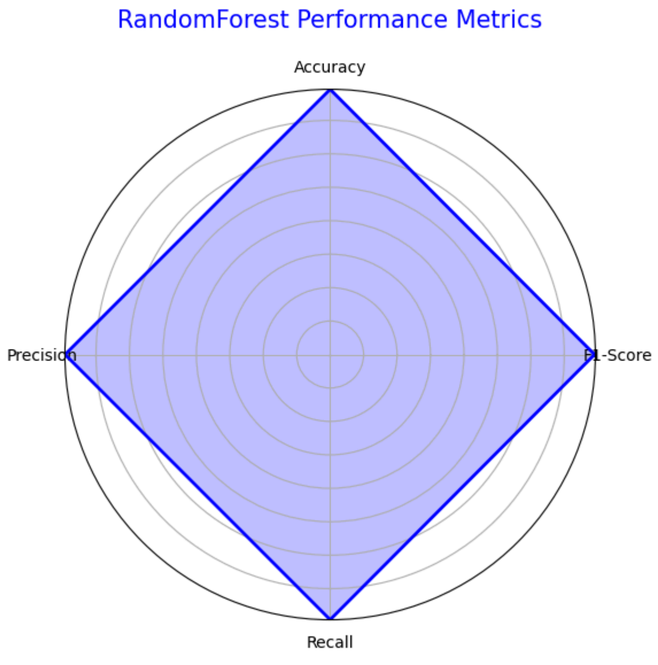
The dataset was split into training and testing sets, with the training set used to fit the models and the testing set used for evaluation. Initially, default hyperparameters were used for each model to determine their baseline performance. The model exhibiting the highest F1-Score was identified as the best-performing model.

To ensure optimal performance, the best-performing model underwent hyperparameter optimization using Optuna, an automatic hyperparameter optimization framework. This optimization process involved conducting multiple trials to explore various hyperparameter configurations, with the objective function being the maximization of the F1-Score.

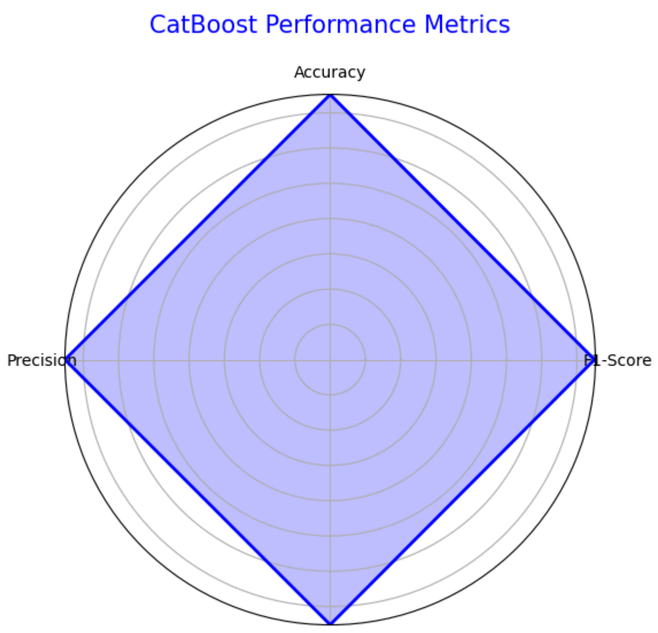
In [425…
```
model_selector = ModelSelector(X_train_encoded, y_train_encoded, X_test_encoded, y_test_encoded, preprocessor.target_encoder)

selected_feats = selected_features_rfe.columns.to_list()
model_selector_sf = ModelSelector(X_train_encoded[selected_feats], y_train_encoded, X_test_encoded[selected_feats], y_test_encoded, preprocessor.target_encoder)
```

In [427…
```
model_selector.evaluate_models()
```

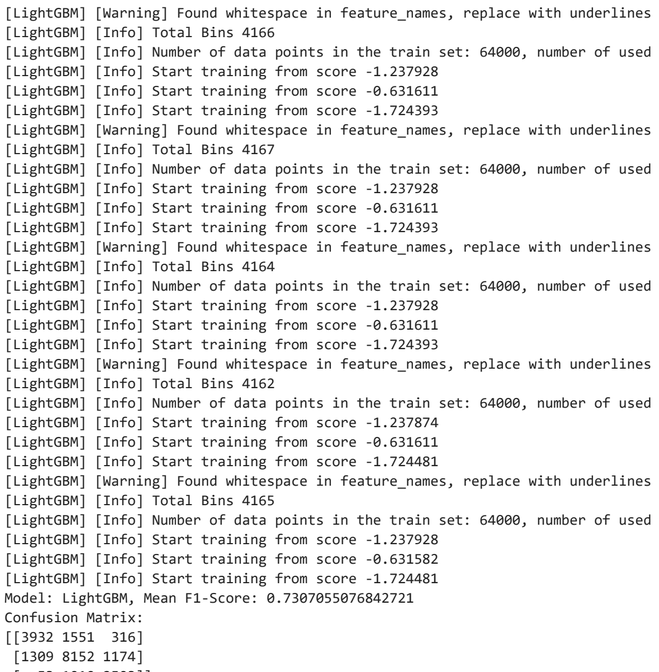
```
Model: RandomForest, Mean F1-Score: 0.790382012313653
Confusion Matrix:
[[4676 1050   73]
 [1285 8622  728]
 [  23  985 2558]]
Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.81      0.79      5799
           1       0.81      0.81      0.81     10635
           2       0.76      0.72      0.74      3566

    accuracy                           0.79     20000
   macro avg       0.78      0.78      0.78     20000
weighted avg       0.79      0.79      0.79     20000
```



```
Model: CatBoost, Mean F1-Score: 0.7495376796357546
Confusion Matrix:
[[4223 1353  223]
 [1299 8364  972]
 [  46 1072 2448]]
Classification Report:
              precision    recall  f1-score   support

           0       0.76      0.73      0.74      5799
           1       0.78      0.79      0.78     10635
           2       0.67      0.69      0.68      3566

    accuracy                           0.75     20000
   macro avg       0.74      0.73      0.73     20000
weighted avg       0.75      0.75      0.75     20000
```




```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4166
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 45
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724393
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4167
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 45
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724393
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4164
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 45
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724393
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4162
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 45
[LightGBM] [Info] Start training from score -1.237874
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724481
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4165
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 45
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631582
[LightGBM] [Info] Start training from score -1.724481
Model: LightGBM, Mean F1-Score: 0.7307055076842721
Confusion Matrix:
[[3932 1551  316]
 [1309 8152 1174]
 [  53 1010 2503]]
Classification Report:
              precision    recall  f1-score   support

           0       0.74      0.68      0.71      5799
           1       0.76      0.77      0.76     10635
           2       0.63      0.70      0.66      3566

    accuracy                           0.73     20000
   macro avg       0.71      0.72      0.71     20000
weighted avg       0.73      0.73      0.73     20000
```

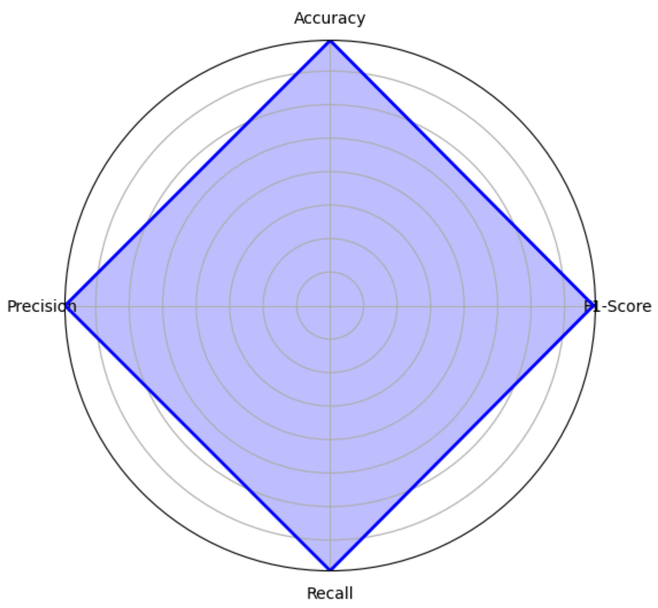



```
Best Model: RandomForest, Best Mean F1-Score: 0.790382012313653
```

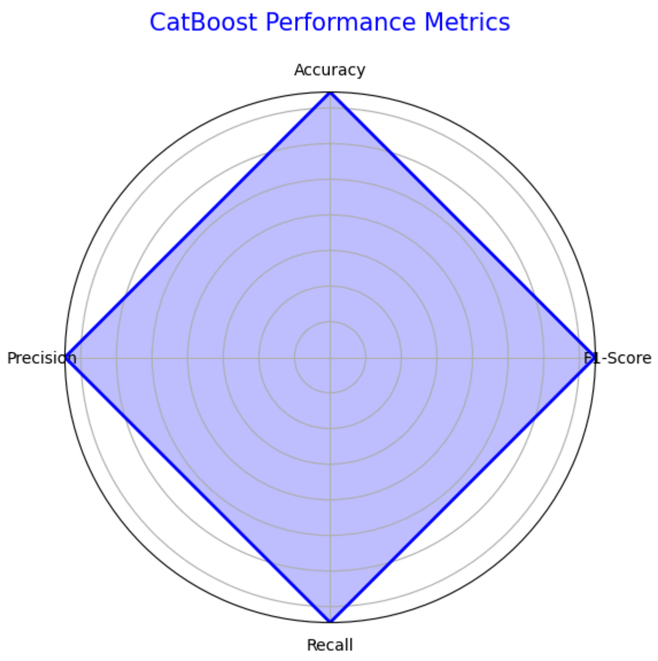
Out[427…
```
{'RandomForest': 0.790382012313653,
 'CatBoost': 0.7495376796357546,
 'LightGBM': 0.7307055076842721}
```

In [428…
```
model_selector_sf.evaluate_models()
```

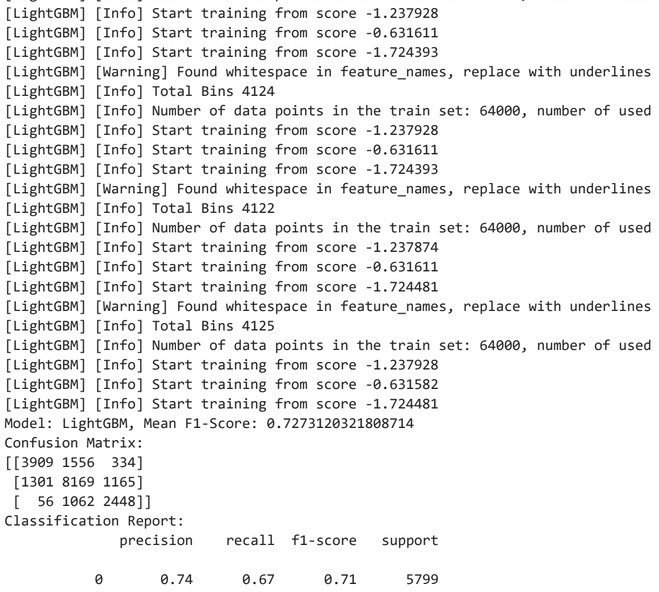
```
Model: RandomForest, Mean F1-Score: 0.7860856351648302
Confusion Matrix:
[[4671 1033   95]
 [1281 8594  760]
 [  34  965 2567]]
Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.81      0.79      5799
           1       0.81      0.81      0.81     10635
           2       0.75      0.72      0.73      3566

    accuracy                           0.79     20000
   macro avg       0.78      0.78      0.78     20000
weighted avg       0.79      0.79      0.79     20000
```




```
Model: CatBoost, Mean F1-Score: 0.7443241850565887
Confusion Matrix:
[[4182 1376  241]
 [1306 8329 1000]
 [  53 1131 2382]]
Classification Report:
              precision    recall  f1-score   support

           0       0.75      0.72      0.74      5799
           1       0.77      0.78      0.78     10635
           2       0.66      0.67      0.66      3566

    accuracy                           0.74     20000
   macro avg       0.73      0.72      0.73     20000
weighted avg       0.74      0.74      0.74     20000
```




```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4126
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 25
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724393
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4127
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 25
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724393
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4124
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 25
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724393
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4122
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 25
[LightGBM] [Info] Start training from score -1.237874
[LightGBM] [Info] Start training from score -0.631611
[LightGBM] [Info] Start training from score -1.724481
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Total Bins 4125
[LightGBM] [Info] Number of data points in the train set: 64000, number of used features: 25
[LightGBM] [Info] Start training from score -1.237928
[LightGBM] [Info] Start training from score -0.631582
[LightGBM] [Info] Start training from score -1.724481
Model: LightGBM, Mean F1-Score: 0.7273120321808714
Confusion Matrix:
[[3909 1556  334]
 [1301 8169 1165]
 [  56 1062 2448]]
Classification Report:
              precision    recall  f1-score   support

           0       0.74      0.67      0.71      5799
           1       0.76      0.77      0.76     10635
           2       0.62      0.69      0.65      3566

    accuracy                           0.73     20000
   macro avg       0.71      0.71      0.71     20000
weighted avg       0.73      0.73      0.73     20000
```




```
Best Model: RandomForest, Best Mean F1-Score: 0.7860856351648302
```

Out[428…
```
{'RandomForest': 0.7860856351648302,
 'CatBoost': 0.7443241850565887,
 'LightGBM': 0.7273120321808714}
```

Upon evaluating the models trained with the top 25 selected features and those trained with all available features, it was observed that the model incorporating all features yielded a superior F1 score. This result is evident in the graphs presented above. Consequently, for subsequent stages of analysis, the model trained with the complete set of features will be utilized.

The comprehensive approach of including all features has demonstrated its efficacy in achieving better predictive performance. This decision aligns with the aim to maximize the model's accuracy and reliability in real-world applications. Future research and model enhancements will therefore focus on leveraging the insights gained from the full-featured model, ensuring a thorough and robust analytical framework.

4.2 Model Training

This study implemented a rigorous model training process that leverages Stratified K-Fold Cross-Validation to ensure robust and unbiased evaluation of different machine learning models. The objective was to select the best performing model based on F1-score and subsequently optimize it using hyperparameter tuning.

Stratified K-Fold Cross-Validation

Stratified K-Fold Cross-Validation was employed to partition the dataset into training and validation folds while maintaining the class distribution across all folds. This method is particularly beneficial for imbalanced datasets, such as this one, where preserving the proportion of each class is crucial for reliable model evaluation. Specifically, 5 folds were used, meaning the dataset was divided into 5 parts, with each part serving as a validation set once while the remaining parts formed the training set. This iterative process ensures that each instance in the dataset is used for both training and validation, providing a comprehensive evaluation of the model's performance.

4.3 Hyperparameter Tuning

Hyperparameter Tuning with Optuna

- After identifying the best performing model based on the highest mean F1-score, hyperparameters were optimized using Optuna, a hyperparameter optimization framework. The objective function was defined to maximize the F1-score. The tuning process involved:

1. Hyperparameter Search Space

- Defining a search space for the model's hyperparameters, such as the number of estimators, learning rate, and maximum depth.

2. Optimization

- Running multiple trials to explore different combinations of hyperparameters and evaluating the model's performance on the validation set.

3. Best Parameters

- Selecting the hyperparameters that yielded the highest F1-score during the optimization process.
- The tuned model was then trained on the entire training set and evaluated on the test set to obtain the final F1-score. To ensure the reliability of the tuned model, the F1-score was compared with the default best F1-score obtained during the initial evaluation. If the tuned model's F1-score was lower than the default, the model with its default parameters was used instead.

In [431…
```
model_selector.optimize_best_model()
```

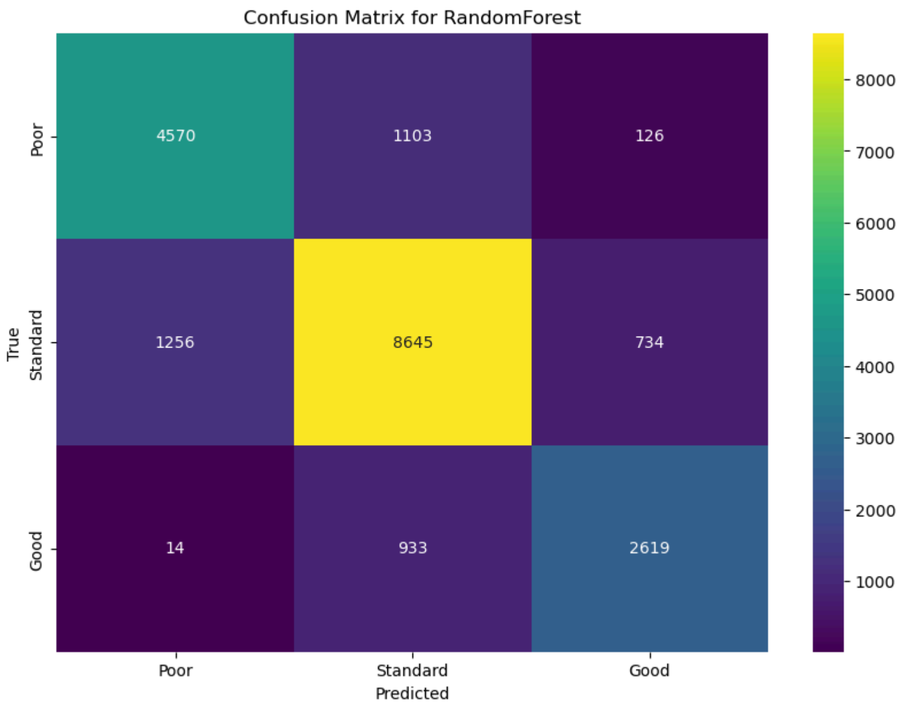
```
[I 2024-12-17 14:03:47,361] A new study created in memory with name: no-name-eeee9421-20cb-4d6c-958c-e10c58aaa2cf
[I 2024-12-17 14:04:17,489] Trial 0 finished with value: 0.7722480042348766 and parameters: {'n_estimators': 136, 'max_depth': 19, 'min_samples_split': 8, 'min_samples_leaf': 6}. Best is trial 0 with value: 0.7722480042348766.
[I 2024-12-17 14:04:58,312] Trial 1 finished with value: 0.7521255775357396 and parameters: {'n_estimators': 193, 'max_depth': 44, 'min_samples_split': 9, 'min_samples_leaf': 20}. Best is trial 0 with value: 0.7722480042348766.
[I 2024-12-17 14:05:29,445] Trial 2 finished with value: 0.7779843418162464 and parameters: {'n_estimators': 136, 'max_depth': 35, 'min_samples_split': 10, 'min_samples_leaf': 8}. Best is trial 2 with value: 0.7779843418162464.
[I 2024-12-17 14:06:19,488] Trial 3 finished with value: 0.7513723208589028 and parameters: {'n_estimators': 244, 'max_depth': 21, 'min_samples_split': 16, 'min_samples_leaf': 18}. Best is trial 2 with value: 0.7779843418162464.
[I 2024-12-17 14:06:53,844] Trial 4 finished with value: 0.7880296004988673 and parameters: {'n_estimators': 139, 'max_depth': 50, 'min_samples_split': 13, 'min_samples_leaf': 5}. Best is trial 4 with value: 0.7880296004988673.
[I 2024-12-17 14:07:05,957] Trial 5 finished with value: 0.6479310241295294 and parameters: {'n_estimators': 145, 'max_depth': 5, 'min_samples_split': 4, 'min_samples_leaf': 10}. Best is trial 4 with value: 0.7880296004988673.
[I 2024-12-17 14:07:39,299] Trial 6 finished with value: 0.7115223879376307 and parameters: {'n_estimators': 231, 'max_depth': 10, 'min_samples_split': 17, 'min_samples_leaf': 7}. Best is trial 4 with value: 0.7880296004988673.
[I 2024-12-17 14:07:55,085] Trial 7 finished with value: 0.6850121146482251 and parameters: {'n_estimators': 113, 'max_depth': 8, 'min_samples_split': 4, 'min_samples_leaf': 5}. Best is trial 4 with value: 0.7880296004988673.
[I 2024-12-17 14:08:35,037] Trial 8 finished with value: 0.7608811118242522 and parameters: {'n_estimators': 165, 'max_depth': 43, 'min_samples_split': 4, 'min_samples_leaf': 15}. Best is trial 4 with value: 0.7880296004988673.
[I 2024-12-17 14:09:21,145] Trial 9 finished with value: 0.7637854953880349 and parameters: {'n_estimators': 212, 'max_depth': 26, 'min_samples_split': 7, 'min_samples_leaf': 13}. Best is trial 4 with value: 0.7880296004988673.
[I 2024-12-17 14:09:34,541] Trial 10 finished with value: 0.7887911670135236 and parameters: {'n_estimators': 52, 'max_depth': 49, 'min_samples_split': 13, 'min_samples_leaf': 1}. Best is trial 10 with value: 0.7887911670135236.
[I 2024-12-17 14:09:49,875] Trial 11 finished with value: 0.7917618255223748 and parameters: {'n_estimators': 60, 'max_depth': 50, 'min_samples_split': 14, 'min_samples_leaf': 1}. Best is trial 11 with value: 0.7917618255223748.
[I 2024-12-17 14:10:02,723] Trial 12 finished with value: 0.7926392698612904 and parameters: {'n_estimators': 51, 'max_depth': 50, 'min_samples_split': 14, 'min_samples_leaf': 1}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:10:15,674] Trial 13 finished with value: 0.7831464562403114 and parameters: {'n_estimators': 53, 'max_depth': 36, 'min_samples_split': 20, 'min_samples_leaf': 1}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:11:28,888] Trial 14 finished with value: 0.7921994099446898 and parameters: {'n_estimators': 286, 'max_depth': 37, 'min_samples_split': 14, 'min_samples_leaf': 3}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:12:42,593] Trial 15 finished with value: 0.7886425249991865 and parameters: {'n_estimators': 298, 'max_depth': 36, 'min_samples_split': 17, 'min_samples_leaf': 3}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:13:51,008] Trial 16 finished with value: 0.785450972608009 and parameters: {'n_estimators': 280, 'max_depth': 42, 'min_samples_split': 20, 'min_samples_leaf': 3}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:14:10,465] Trial 17 finished with value: 0.7686964220719442 and parameters: {'n_estimators': 87, 'max_depth': 31, 'min_samples_split': 12, 'min_samples_leaf': 10}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:15:14,337] Trial 18 finished with value: 0.7897187949640447 and parameters: {'n_estimators': 259, 'max_depth': 41, 'min_samples_split': 15, 'min_samples_leaf': 3}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:15:36,010] Trial 19 finished with value: 0.7768834497479884 and parameters: {'n_estimators': 94, 'max_depth': 29, 'min_samples_split': 11, 'min_samples_leaf': 8}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:16:15,652] Trial 20 finished with value: 0.7666448404646572 and parameters: {'n_estimators': 180, 'max_depth': 46, 'min_samples_split': 18, 'min_samples_leaf': 12}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:16:35,188] Trial 21 finished with value: 0.7922933989305566 and parameters: {'n_estimators': 75, 'max_depth': 50, 'min_samples_split': 14, 'min_samples_leaf': 1}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:16:59,015] Trial 22 finished with value: 0.788755237875298 and parameters: {'n_estimators': 95, 'max_depth': 39, 'min_samples_split': 15, 'min_samples_leaf': 3}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:17:18,542] Trial 23 finished with value: 0.7896285016555651 and parameters: {'n_estimators': 80, 'max_depth': 46, 'min_samples_split': 12, 'min_samples_leaf': 4}. Best is trial 12 with value: 0.7926392698612904.
[I 2024-12-17 14:17:46,945] Trial 24 finished with value: 0.7940858111035706 and parameters: {'n_estimators': 110, 'max_depth': 46, 'min_samples_split': 14, 'min_samples_leaf': 1}. Best is trial 24 with value: 0.7940858111035706.
Final Model: RandomForest, Final F1-Score: 0.7915794055576184
Confusion Matrix:
[[4570 1103  126]
 [1256 8645  734]
 [  14  933 2619]]
Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.79      0.79      5799
           1       0.81      0.81      0.81     10635
           2       0.75      0.73      0.74      3566

    accuracy                           0.79     20000
   macro avg       0.78      0.78      0.78     20000
weighted avg       0.79      0.79      0.79     20000

Best hyperparameters for RandomForest: {'n_estimators': 110, 'max_depth': 46, 'min_samples_split': 14, 'min_samples_leaf': 1}
```

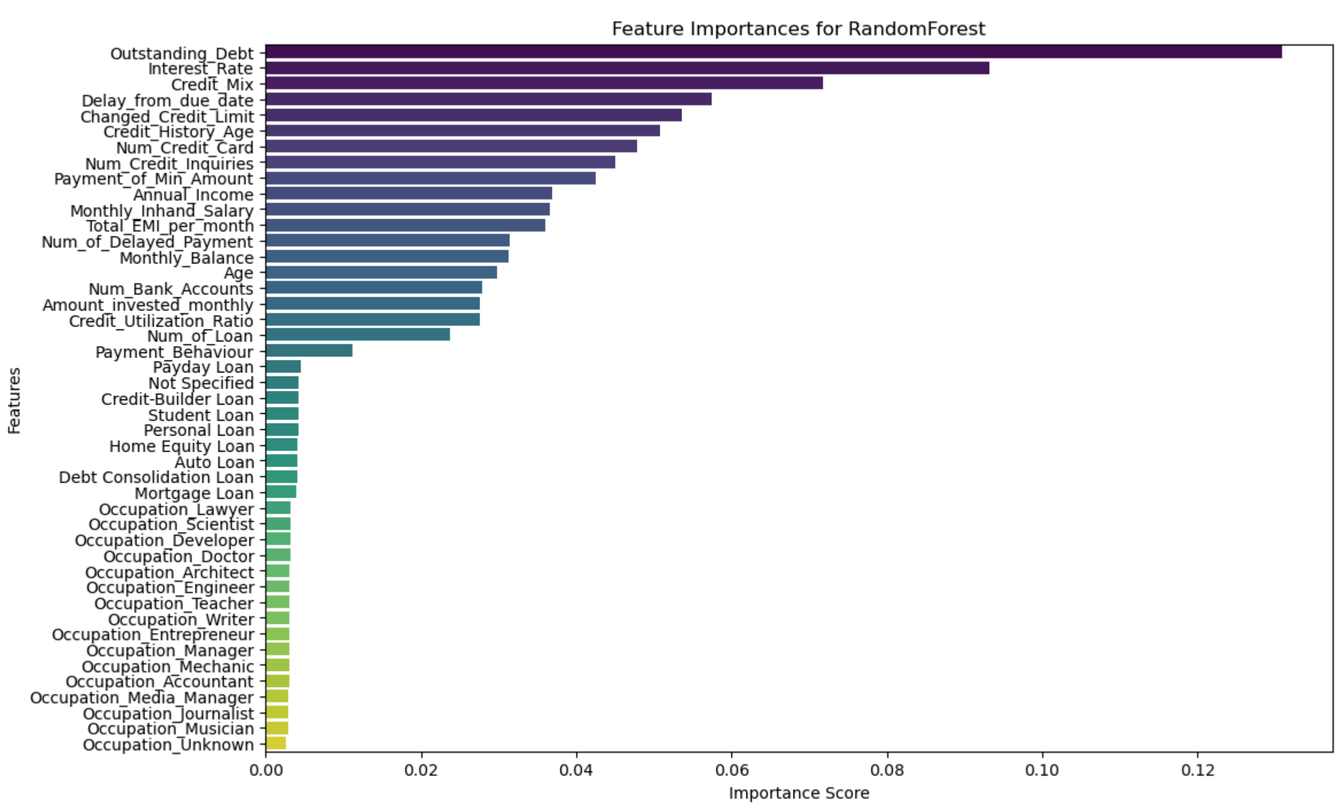
Out[431…
```
{'n_estimators': 110,
 'max_depth': 46,
 'min_samples_split': 14,
 'min_samples_leaf': 1}
```

The optimized hyperparameters for the Random Forest model include n_estimators set to 110, max_depth set to 46, min_samples_split set to 14, and min_samples_leaf set to 1. These hyperparameters play a crucial role in the model's performance. The n_estimators parameter, which defines the number of trees in the forest, is set to 110, ensuring a balance between performance enhancement and computational efficiency. The max_depth parameter, which limits the depth of each tree, is set to 46, allowing the model to learn intricate patterns in the data while preventing overfitting. The min_samples_split parameter, set to 14, controls the sensitivity of the model to variations in the data by ensuring that nodes are only split if there are enough samples to justify the creation of a new branch. This helps reduce overfitting by creating more generalized branches. Lastly, the min_samples_leaf parameter, set to 1, specifies the minimum number of samples required to be at a leaf node. This allows the model to learn from the smallest units of information, capturing fine-grained details in the data, although it may potentially increase the risk of overfitting. Overall, these hyperparameters were carefully selected through rigorous optimization to enhance the Random Forest model's predictive performance, ensuring it captures complex patterns without overfitting. The decision to use these optimized hyperparameters will significantly contribute to the robustness and accuracy of the model, making it a reliable tool for predictive analytics in various applications.

5. Model Evaluation

In [433…
```
model_selector.plot_best_model_performance()
```





Feature Importance Plot (Random Forest)

Observations:

1. Top Features:

- Outstanding_Debt is the most important feature, suggesting that the amount of debt a person owes is a strong predictor of creditworthiness.
- Interest_Rate and Credit_Mix are the next most influential features, highlighting the importance of financial costs and the diversity of credit products.
- Delay_from_due_date, Changed_Credit_Limit, and Credit_History_Age are moderately important, indicating that payment behaviors and credit history are crucial factors.

2. Less Significant Features:

- Specific loan types (e.g., Payday Loan, Home Equity Loan) and certain occupations (e.g., Media Manager, Accountant) have very low importance scores, contributing minimally to the prediction.

Insights:

- Behavioral and financial metrics, such as debt levels, interest rates, and credit usage patterns, are far more critical than demographic or occupation-based features.
- Low-importance features might be considered for removal or further scrutiny to streamline the model.

Confusion Matrix for Random Forest

Observations:

1. Diagonal Values (Correct Predictions):

- Poor: 4,570 predictions were correctly classified as "Poor."
- Standard: 8,645 predictions were correctly classified as "Standard."
- Good: 2,619 predictions were correctly classified as "Good."

2. Off-Diagonal Values (Misclassifications):

- Poor predicted as Standard: 1,103 instances.
- Good predicted as Standard: 933 instances.
- Standard predicted as Poor: 1,256 instances.

3. Key Strength:

- The model performs relatively well in predicting "Standard," as it has the highest number of correct classifications (8,645).
- Key Weakness:

Misclassifications are most prevalent for "Standard," with some overlap between "Poor" and "Good."

5.1 Model Evaluation Metrics

Logic for Selecting F1-Score as the Evaluation Metric

In credit scoring, evaluating model performance is crucial due to the significant financial implications of classification errors. Given the imbalanced nature of this study's dataset, the F1-score has been selected as the primary evaluation metric. As noted by lyzr.ai (2024), "The F1 Score is a crucial metric in evaluating the performance of classification models, particularly when dealing with imbalanced datasets." This decision is grounded in several key considerations, which are elaborated below:

1. **Handling Imbalanced Data**

Credit scoring datasets typically exhibit imbalanced class distributions, where instances of 'Good' credit scores are more frequent compared to 'Poor' and 'Standard' scores. The F1-Score is particularly suited to such scenarios as it harmonizes both precision and recall. Precision measures the proportion of true positive predictions among all positive predictions, while recall measures the proportion of true positive predictions among all actual positives. By balancing these two aspects, the F1-Score provides a more comprehensive evaluation of model performance in the presence of class imbalances (lyzr.ai 2024).

2. **Mitigating Misclassification Risks**

In credit scoring applications, the cost of misclassification can be substantial. False positives (incorrectly predicting a 'Good' credit score) can lead to financial losses for lenders, while false negatives (incorrectly predicting a 'Poor' credit score) can result in missed opportunities. The F1-Score addresses both types of errors by ensuring that the model is not only accurate in its positive predictions but also effectively captures all relevant positive instances. This dual focus is critical in minimizing the financial and operational risks associated with misclassification (lyzr.ai 2024).

3. **Practical Implications**

The practical implications of using the F1-Score are significant. A high precision ensures that individuals classified with 'Good' credit scores are indeed likely to repay their loans, thereby reducing the risk for lenders. Concurrently, a high recall ensures that most individuals who are actually creditworthy are identified, promoting financial inclusivity and enhancing customer satisfaction. This balance is crucial in developing a reliable and equitable credit scoring system.

4. **Model Optimization and Selection**

The use of the F1-Score as the optimization target during the hyperparameter tuning process ensures that the model achieves optimal performance across all class distributions. This approach is particularly beneficial when utilizing automated hyperparameter optimization frameworks such as Optuna. By focusing on maximizing the F1-Score, the study ensures that the selected model performs robustly and equitably across all relevant metrics.

In summary, the choice of the F1-Score as the evaluation metric in this study is justified by its ability to handle imbalanced data, mitigate misclassification risks, and ensure practical and robust model performance (lyzr.ai 2024). This metric provides a balanced and comprehensive framework for evaluating and optimizing credit scoring models, thereby supporting more accurate and fair decision-making in financial contexts.

5.2 Model Comparison

In the initial phase of the model evaluation, three prominent machine learning algorithms were assessed: RandomForest, CatBoost, and LightGBM. Each model was tested with its default hyperparameters to establish a baseline performance, measured using the F1-Score. The results of this evaluation are as follows:

RandomForest: 0.791

CatBoost: 0.749

LightGBM: 0.731

Based on these F1-Scores, the RandomForest model demonstrated the highest performance, achieving a score of 0.8028. Consequently, the RandomForest model was selected for further analysis and hyperparameter optimization, as it outperformed the other models in accurately balancing precision and recall in the classification task.

This step ensured that the most effective model, according to the F1-Score metric, was chosen to undergo additional tuning to maximize its performance potential. By focusing on the best-performing model, the study aimed to optimize resource utilization and achieve superior predictive accuracy in the subsequent stages.

5.3 Model Interpretation

Confusion Matrix Analysis

The confusion matrix provides a detailed breakdown of the model's performance by comparing the actual target values with the predicted values across three classes: Poor, Standard, and Good.

Actual Poor Class

Correctly Predicted as Poor: 4570 instances

Incorrectly Predicted as Standard: 1103 instances

Incorrectly Predicted as Good: 126 instances

The majority of Poor class instances are correctly identified, though there are notable misclassifications, particularly to the Standard class.

Actual Standard Class

Incorrectly Predicted as Poor: 1256 instances

Correctly Predicted as Standard: 8645 instances

Incorrectly Predicted as Good: 734 instances

The Standard class has a high number of correctly predicted instances, but there are also significant misclassifications to both Poor and Good classes. This indicates that the model is generally effective but struggles with the boundaries between Standard and the other classes.

Actual Good Class

Incorrectly Predicted as Poor: 14 instances

Incorrectly Predicted as Standard: 933 instances

Correctly Predicted as Good: 2619 instances

Most Good class instances are correctly predicted, but some are misclassified as Standard, and a very small number as Poor. The number of misclassifications here is smaller compared to the Standard class, suggesting better differentiation by the model for this class.

Overall Performance

The model shows strong performance in correctly classifying the majority of instances, especially in the Poor and Good classes. However, there is noticeable confusion between the Poor and Standard classes, and between the Standard and Good classes. This indicates areas where the model's predictions could be further refined to improve its precision and recall for these specific classifications.

6. Conclusion

This study focused on developing a robust credit scoring model using advanced machine learning techniques, offering a significant contribution to the field of credit risk assessment. By thoroughly evaluating and optimizing the performance of three notable algorithms—RandomForest, CatBoost, and LightGBM—this research provided valuable insights into the efficacy of these models in predicting credit scores.

The initial phase involved establishing baseline performances for each model using default hyperparameters, with the RandomForest model emerging as the most effective in balancing precision and recall. This highlighted its potential as the most suitable algorithm for credit scoring tasks. Following this, an in-depth hyperparameter optimization process was conducted using Optuna, which further enhanced the performance of the RandomForest model. This step was crucial in ensuring the model's robustness and its ability to generalise well to unseen data.

The confusion matrix analysis indicated that while the model performed well overall, there were specific areas—particularly in distinguishing between adjacent classes—where improvements could be made. This analysis provided a deeper understanding of the model's strengths and weaknesses, guiding future enhancements. Additionally, the feature importance analysis shed light on the key predictors influencing credit scores, thereby enhancing the interpretability of the model and aiding decision-makers in understanding the underlying factors driving credit risk.

In terms of practical implications, this study underscores the transformative potential of integrating machine learning models into credit scoring systems. Financial institutions can leverage these insights to develop more accurate and fair credit assessment tools, ultimately leading to better risk management practices and more inclusive financial services. The proposed future work directions, including the incorporation of real-time data streams, the development of explainable AI models, and the expansion to emerging markets, highlight the opportunities for continuous innovation in this domain.

Overall, this research not only demonstrates the effectiveness of advanced machine learning models in credit scoring but also lays the groundwork for further advancements in the field. By embracing these innovative approaches, financial institutions can enhance their credit risk assessment capabilities, improve regulatory compliance, and better serve their customers in an ever-evolving financial landscape. Continuous innovation and adherence to best practices will be key to fully realizing these benefits and driving forward the next generation of credit scoring solutions.

6.1 Findings Summary

Model Performance

The initial evaluation involved testing the models with their default settings. Among the three, RandomForest demonstrated the highest effectiveness in balancing precision and recall, making it the preferred choice for further enhancement.

Hyperparameter Optimization

Following the selection, hyperparameter tuning was performed using Optuna. This process involved exploring various configurations to refine the model's performance, ultimately confirming RandomForest's robustness and suitability for credit scoring tasks.

Confusion Matrix Analysis

The analysis of the confusion matrix revealed that the RandomForest model performed well in correctly classifying most instances, particularly in differentiating between the Poor and Good classes. However, there was some confusion in distinguishing between the Poor and Standard classes and between the Standard and Good classes, indicating areas for potential improvement.

Feature Importance

The examination of feature importances provided by the RandomForest model identified key predictors that influence credit scores. This insight is crucial for understanding the underlying factors contributing to credit risk, aiding in the model's interpretability and supporting better decision-making.

6.2 Limitations

Data Quality and Availability

One major limitation of this study lies in the quality and availability of the data used. If the dataset is not representative of the broader population or contains biases, the model's predictions may not generalise well to unseen data. Additionally, missing or inaccurate data can further degrade model performance, particularly in the context of credit scoring where accurate predictions are crucial.

Model Complexity and Interpretability

While advanced machine learning models such as RandomForest, CatBoost, and LightGBM can achieve high predictive accuracy, they often lack interpretability. Credit scoring decisions require transparency, and complex models may not provide easily understandable insights into why a particular decision was made. This can be a significant drawback in regulatory environments where explainability is necessary.

Computational Resources

The optimisation and training of multiple models, especially with extensive hyperparameter tuning using frameworks like Optuna, can be computationally expensive. Limited access to high-performance computing resources can constrain the ability to explore a comprehensive range of model configurations, potentially impacting the study's outcomes.

Hyperparameter Tuning Constraints

The effectiveness of hyperparameter tuning is dependent on the search space defined for the parameters. An insufficiently defined search space may lead to suboptimal tuning, while an excessively broad search space can result in computational inefficiency. S triking the right balance is crucial, yet challenging.

Imbalanced Class Distribution

Credit scoring datasets often suffer from imbalanced class distributions, where certain credit scores are underrepresented. While metrics like F1-Score help address some aspects of imbalance, they may not fully capture the performance nuances across all classes. Additional techniques such as oversampling, undersampling, or synthetic data generation may be necessary to mitigate this issue.

Overfitting Risk

Complex models with extensive hyperparameter tuning are prone to overfitting, where the model learns the noise in the training data rather than the underlying patterns. This can lead to excellent performance on the training set but poor generalization to new data. Techniques such as cross-validation and regularization are essential but may not completely eliminate this risk.

Evolving Market Conditions

Credit risk models must adapt to changing economic conditions and consumer behaviors. A model trained on historical data may become obsolete if it fails to account for significant market shifts. Continuous monitoring and updating of the model are required to maintain its relevance and accuracy over time.

These limitations underscore the importance of a careful and nuanced approach to model development and evaluation in the domain of credit scoring. Addressing these challenges is essential for building robust, reliable, and interpretable models that can effectively support credit risk management decisions.

6.3 Future Work

1. Integration with Real-time Data Streams

Incorporating real-time data streams, such as transaction histories, social media sentiment analysis, and macroeconomic indicators, can significantly enhance the predictive power of credit scoring models. This would allow for dynamic credit scoring that adapts to the latest consumer behaviors and economic conditions, providing more accurate and timely risk assessments.

2. Development of Explainable AI Models

While the current models focus on predictive accuracy, future work could involve developing explainable AI (XAI) models. These models would provide insights into the decision-making process, making it easier for financial institutions to understand and trust the results. Implementing methods like SHAP (SHapley Additive exPlanations) values can help in explaining the contribution of each feature to the model's predictions.

3. Implementation of Adaptive Learning Systems

Adaptive learning systems that continuously update the model with new data can help maintain its relevance and accuracy over time. These systems would be particularly useful in environments where consumer behavior and economic conditions are rapidly changing, ensuring that the credit scoring model remains up-to-date and effective.

4. Development of Custom Credit Products

Using advanced credit scoring models, financial institutions can develop custom credit products tailored to different segments of their customer base. For example, special loan packages could be designed for young professionals, first-time homebuyers, or small business owners, based on their unique credit profiles and risk levels.

5. Expansion to Emerging Markets

Future work could explore the adaptation of these models to emerging markets where traditional credit scoring methods may not be as effective. By leveraging alternative data sources, such as mobile phone usage patterns and utility payment histories, financial institutions can extend credit access to underserved populations and promote financial inclusion.

6. Partnership with Fintech Companies

Collaborating with fintech companies can open up new avenues for applying advanced credit scoring models. Fintech firms often have innovative approaches to data collection and analysis, which can complement traditional banking methods and enhance the overall credit assessment process.

7. Regulatory Compliance and Risk Management

Future studies could focus on ensuring that credit scoring models comply with evolving regulatory requirements. This includes implementing robust risk management frameworks and conducting regular audits to verify the fairness and accuracy of the models. By staying compliant, financial institutions can avoid legal challenges and maintain consumer trust.

8. Personalised Financial Advice

Integrating credit scoring models with platforms that offer personalised financial advice can provide consumers with actionable insights to improve their credit scores. By understanding the factors influencing their scores, consumers can make informed decisions about managing their finances, ultimately leading to better credit outcomes.

These future work directions highlight the potential for advanced credit scoring models to drive innovation and growth in the financial sector. By leveraging cutting-edge technologies and data-driven insights, businesses can develop more sophisticated and inclusive credit products, enhance risk management practices, and better serve their customers.

7. References

1. Abdou, H.A. and Pointon, J., 2011. Credit scoring, statistical techniques and evaluation criteria: a review of the literature. Intelligent systems in accounting, finance and management, 18(2-3), pp.59-88.
2. Batista, G.E.A.P.A. and Monard, M.C. (2003) 'An analysis of four missing data treatment methods for supervised learning.' *Applied Artificial Intelligence*, 17(5–6), pp. 519–533. Available at: https://doi.org/10.1080/713827181 .
3. Bergstra, J. and Bengio, Y. (2012) 'Random search for hyper-parameter optimization.' *Journal of Machine Learning Research*, 13(1), pp. 281–305. Available at: https://jmlr.org/papers/v13/bergstra12a.html .
4. Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., Vanderplas, J., Joly, A., Holt, B. and Varoquaux, G. (2013) 'API design for machine learning software: experiences from the scikit-learn project.' Available at: https://arxiv.org/abs/1309.0238 .
5. Dumitrescu, E., Dordain, E. and Medjahed, H. (2022) 'Machine learning for credit scoring.' *HAL Open Science*. Available at: https://hal.science/hal-03331114v1/file/Machine%20learning%20for%20credit%20scorin.pdf
6. Few, S.(2012) Show me the numbers: designing tables and graphs to enlighten*. 2nd edn. Burlingame: Analytics Press.
7. Guyon, I., Weston, J., Barnhill, S. and Vapnik, V. (2002) 'Gene selection for cancer classification using support vector machines.' *Machine Learning*, 46(1–3), pp. 389–422. Available at: https://doi.org/10.1023/A:1012487302797 .
8. Kuhn, M. and Johnson, K. (2013) *Applied predictive modeling*. New York: Springer. Available at: https://link.springer.com/book/10.1007/978-1-4614-6849-3 .
9. lyzr.ai (2024) 'Understanding F1 Score: A Comprehensive Guide.' Available at: https://www.lyzr.ai/glossaries/f1-score/ .
10. Parisrohan (2023) 'Credit Score Classification.' Available at: https://www.kaggle.com/datasets/parisrohan/credit-score-classification .
11. Saito, T. and Rehmsmeier, M. (2015) 'The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets.' *PLOS ONE*, 10(3), pp. 1–21. Available at: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0118432 .

In [ ]: