**Python Code Explanation & output attachment**

**Step 1:** **Importing the case database.db into Python Environment – IDE (Jupyter Notebook)**

**Description:** I have imported the .db file directly into the python environment.

**Implemented Code:**

import pandas as pd

import numpy as np

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.model\_selection import train\_test\_split

import sqlite3

db\_path = r'C:\Users\Admin\OneDrive\Desktop\Cloudwalk Test Case\case database\case database.db' # Change this to your actual file path

conn = sqlite3.connect(db\_path)

conn.execute("PRAGMA busy\_timeout = 30000")

sql\_queries = {

'users': "SELECT \* FROM users;",

'emotional\_data': "SELECT \* FROM emotional\_data;",

'loans': "SELECT \* FROM loans;"

}

df\_users = pd.read\_sql(sql\_queries['users'], conn)

df\_emotional = pd.read\_sql(sql\_queries['emotional\_data'], conn)

df\_loans = pd.read\_sql(sql\_queries['loans'], conn)

conn.close()

print("Users DataFrame:")

print(df\_users.head())

print("\nEmotional Data DataFrame:")

print(df\_emotional.head())

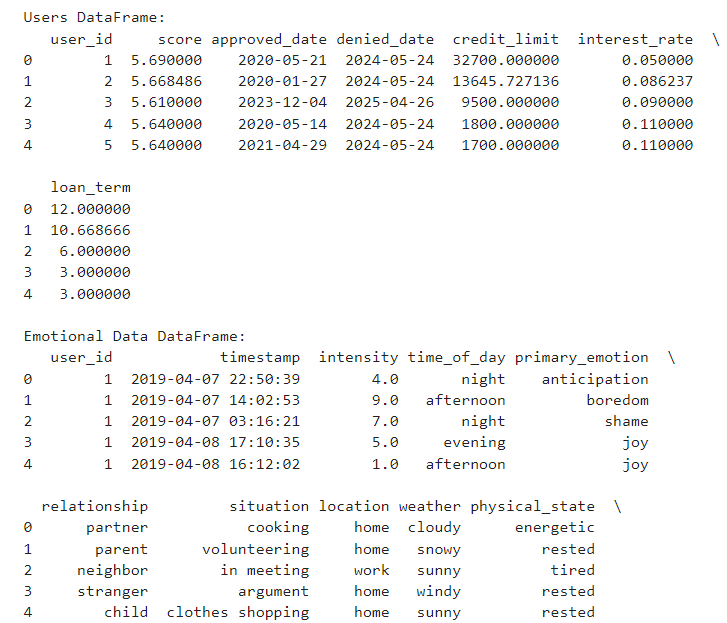
print("\nLoans DataFrame:")

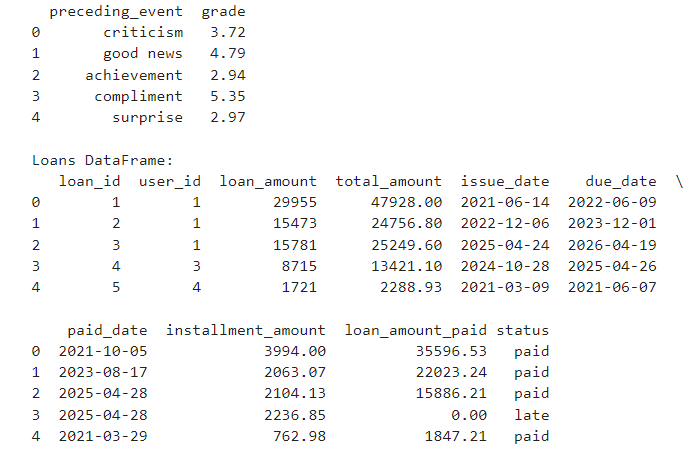
print(df\_loans.head())

**Explanation of the code:**

* **Database Path:** The path to the SQLite database file is specified. It's important to ensure this path is correct to establish a successful connection.
* **Connecting to the Database:** sqlite3.connect(db\_path) creates a connection to the specified SQLite database.
* **Busy Timeout:** The PRAGMA busy\_timeout command is used to set a timeout (30 seconds in this case) for situations where the database is locked (e.g., another process is writing to it).
* **QL Queries Dictionary:** This dictionary defines SQL queries to select all records from three tables: users, emotional\_data, and loans.
* **Reading SQL into DataFrames:** The pd.read\_sql() function executes the SQL query and loads the result into a pandas DataFrame. This allows for easy manipulation and analysis of the data within Python.
* **Connection Closure:** It's essential to close the database connection after the data has been loaded to free up resources.
* **Output Data Verification:** The first few rows of each DataFrame are printed to the console using .head() to confirm that the data has been loaded correctly and is structured as expected.

**Result: Executed (Output Screenshot attached below)**





**Step 2: Data Exploration (Retrieving the Information of the Data)**

**Description:**  The primary purpose of this code is to explore the datasets for the users, emotional data, and loans. It provides basic information about each DataFrame, including the number of entries, data types, and summary statistics, which are crucial for understanding the characteristics of the data before performing any analysis.

**Implemented Code:**

print("Users DataFrame Info:")

print(df\_users.info())

print("\nEmotional Data DataFrame Info:")

print(df\_emotional.info())

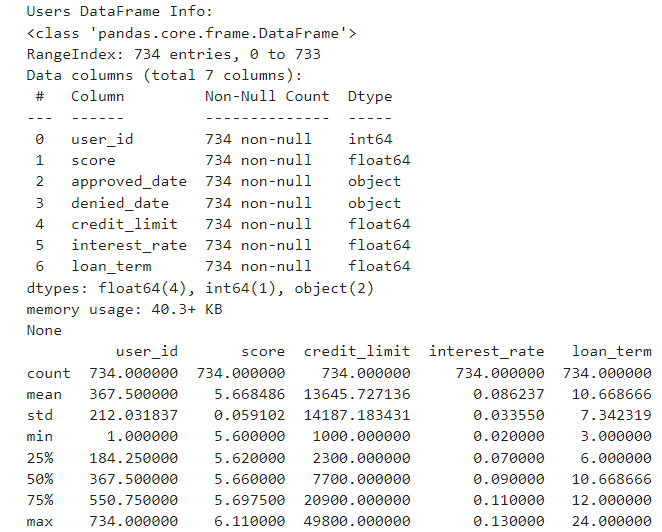
print("\nLoans DataFrame Info:")

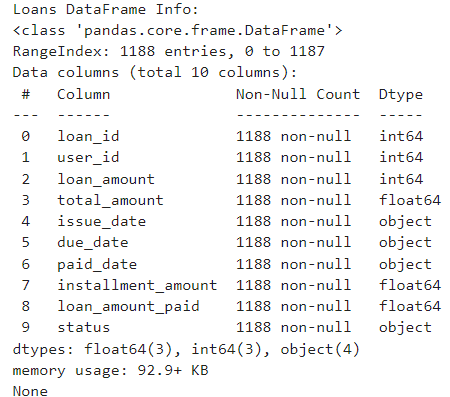
print(df\_loans.info())

**Explanation of the code:**

* **df\_users.info()**: This method displays a concise summary of the DataFrame, including the number of non-null entries, the data types of each column, and memory usage.
* **df\_users.describe()**:This method computes summary statistics for numerical columns, such as count, mean, standard deviation, minimum, maximum, and quartiles.
* **Note:** This is same for all the 3 tables.

**Result: Executed (Output Screenshot attached below)**



**Step 3: Merging 3 tables into Single Data Frame**

**Description:** The primary purpose of this code is to combine data from different sources (loans, users, and emotional data) into a single cohesive DataFrame. This allows for comprehensive analysis across various features, facilitating insights into how user emotions may relate to their loan data.

**Implemented Code:**

data = df\_loans.merge(df\_users, on='user\_id', how='left').merge(df\_emotional, on='user\_id', how='left')

data

**Explanation of the code:**

**1. Merging Dataframe:**

**df\_loans.merge(df\_users, on='user\_id', how='left')**:

**Function:** This operation merges the df\_loans DataFrame with the df\_users DataFrame based on the common column user\_id.

**Parameters:**

**on='user\_id'**: This specifies that the merge should be done using the user\_id column, which is common in both DataFrames.

**how='left'**: This indicates a left join, meaning all records from df\_loans will be retained, and only matching records from df\_users will be included. If there is no match, the corresponding user data will be filled with NaN values.

**Purpose:** To attach user-related information (like credit score, etc.) to each loan record.

**.merge(df\_emotional, on='user\_id', how='left')**:

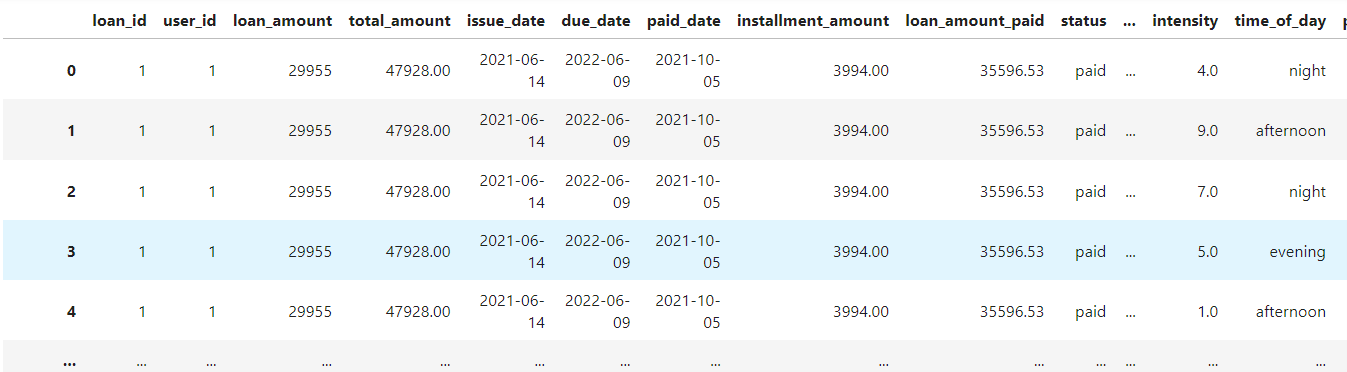
**Function:** This further merges the resulting DataFrame (which now contains loan and user data) with the df\_emotional DataFrame using the user\_id column.

**Purpose:** To include emotional data (like emotional intensity and primary emotions) associated with each loan record. Again, a left join is used to ensure that all records from the previous merge are retained, while only matching emotional data will be included.

**2. Final DataFrame:**

* The resulting DataFrame data now contains:
* All columns from df\_loans, along with corresponding user data from df\_users and emotional data from df\_emotional.
* Each loan record is associated with the relevant user and emotional information, allowing for more nuanced analysis.

**Result: Executed (Output screenshot attached below)**





**Step 3(a): Data Exploration of Merged tables**

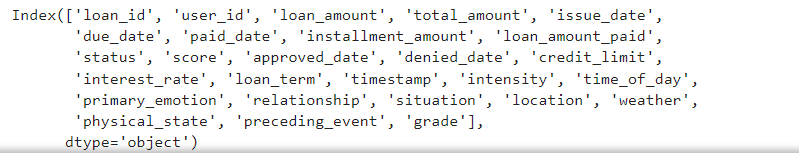
**Implemented code:**

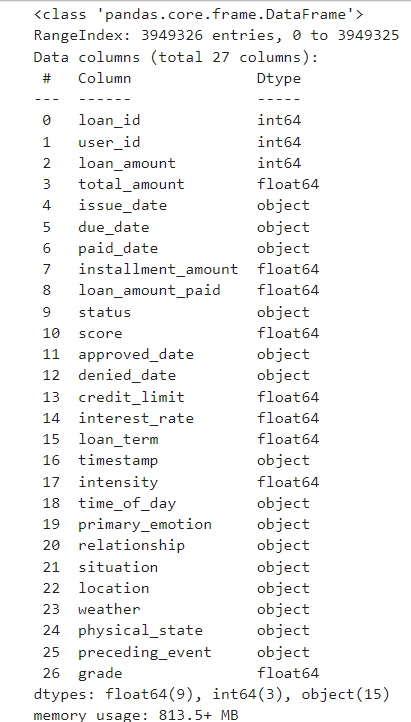
data.columns

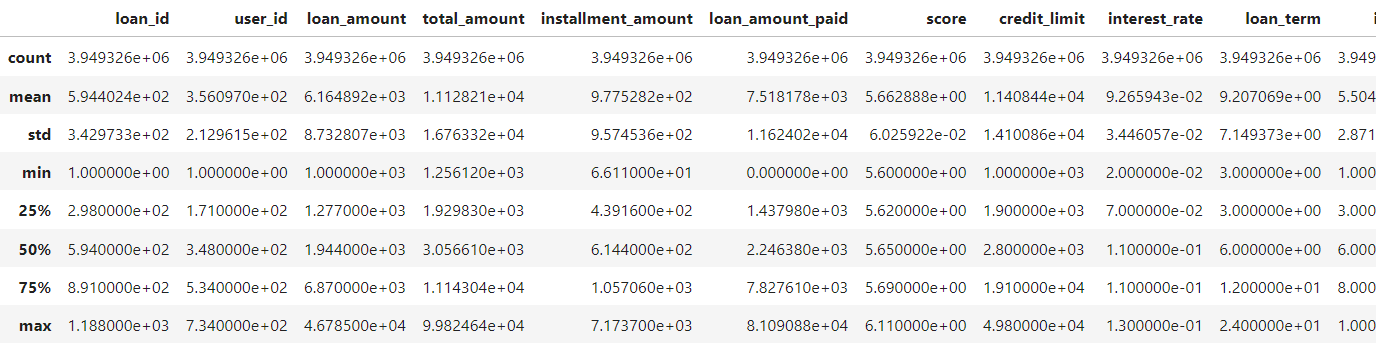
data.info()

data.describe()

**Result: Executed (output screenshot attached below)**







**Step 4: Exploratory Data Analysis (Data Pre-processing)**

**Step 4(a): Finding null values and removing null values**

1. **Finding the Null Values:**

**Description:** The purpose of this code is to identify any missing values (nulls) in the DataFrame, which contains the merged information from loans, users, and emotional data. Checking for null values is an important data cleaning step that helps ensure the integrity of the analysis.

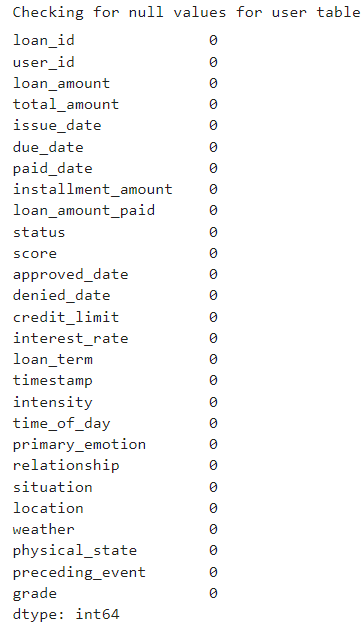
**Implemented code:**

print("Checking for null values for user table:") data.isnull().sum()

**Explanation of Code:**

* **data.isnull()**: This method returns a DataFrame of the same shape as data, with True for each null value and False for non-null values.
* **.sum()**: This method sums up the True values (which represent nulls) for each column in the DataFrame.

**Result: Executed (Output Screenshot attached below)**



**Analysis:**

Here we couldn’t able to find any one of the null values because null values are already treated in using SQL. We imported the .db into python form SQLite database.

1. **Removing Null Values:**

**Description:** This code snippet handles missing values in two ways:

* **Filling missing values in numeric columns** using the mean.
* **Dropping rows with missing values** in specific columns (paid\_date and denied\_date). After these operations, it checks if any missing values remain in the data DataFrame.

**Implementation Code:**

data.fillna(data.mean(numeric\_only=True), inplace=True)

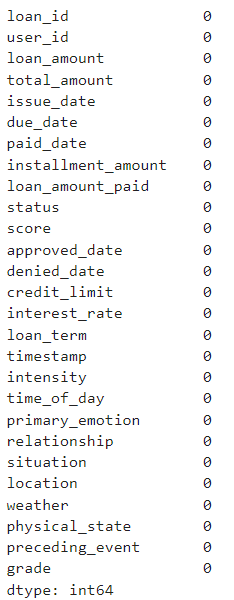
data.dropna(subset=['paid\_date', 'denied\_date'], inplace=True)

data.isnull().sum()

**Explanation of Code:**

* **data.fillna()**: This method replaces missing values (NaN) in the DataFrame.
* **data.mean(numeric\_only=True)**: This calculates the mean for only numeric columns in the data DataFrame.
* **The numeric\_only**=True parameter ensures that the mean is calculated only for numeric columns, ignoring any non-numeric columns.
* **inplace=True**: This ensures that the changes are applied directly to the data DataFrame, without needing to assign it back to data.
* **data.dropna()**: This method removes rows where missing values (NaN) exist.
* **subset=['paid\_date', 'denied\_date']**: This parameter specifies that rows should only be dropped if they have missing values in the paid\_date or denied\_date columns.
* **inplace=True:** This ensures that the changes are applied directly to the data DataFrame, without needing to reassign it.

**Result: Executed (Output of the screenshot is attached below)**

****

**Analysis:** The above method I used helps to remove the null values present in each column. If we remove the unnecessary null values, it will very much helpful for the further analysis

**Step 5: Outlier Detection & Treating the Outliers**

1. **Outlier Detection:**

**Description:** This code generates a grid of boxplots for multiple numerical columns in the dataset (data). Boxplots are helpful for visualizing the distribution of numerical data, highlighting the median, interquartile range (IQR), and identifying outliers. The goal is to examine the spread and detect any potential outliers in features like loan\_amount, credit\_limit, interest\_rate, etc.

**Implemented code:**

import seaborn as sns

import matplotlib.pyplot as plt

interest\_columns = ['loan\_amount', 'total\_amount', 'installment\_amount', 'loan\_amount\_paid', 'score', 'credit\_limit', 'interest\_rate', 'loan\_term', 'intensity', 'grade']

nrows = 3

ncols = 4

fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(20, 15))

axes = axes.flatten()

palette = sns.color\_palette('Set2', len(interest\_columns)) # Using a color palette

for i, column in enumerate(interest\_columns):

sns.boxplot(x=data[column], ax=axes[i], color=palette[i])

axes[i].set\_title(column)

for j in range(len(interest\_columns), len(axes)):

fig.delaxes(axes[j])

plt.tight\_layout()

plt.show()

**Explanation of code:**

**Interest Columns:**

* The interest\_columns list contains the names of the numerical columns from the merged dataset that we want to visualize using boxplots.
* These columns include important features like loan\_amount, credit\_limit, interest\_rate, and more.

**Subplot Grid Setup:**

* This creates a grid of 3 rows and 4 columns to organize the boxplots.
* Even though there are 10 columns to visualize, the grid is designed for 12 plots, allowing extra space for a clean layout.
* figsize=(20, 15) adjusts the overall size of the grid to ensure it fits well on the screen.

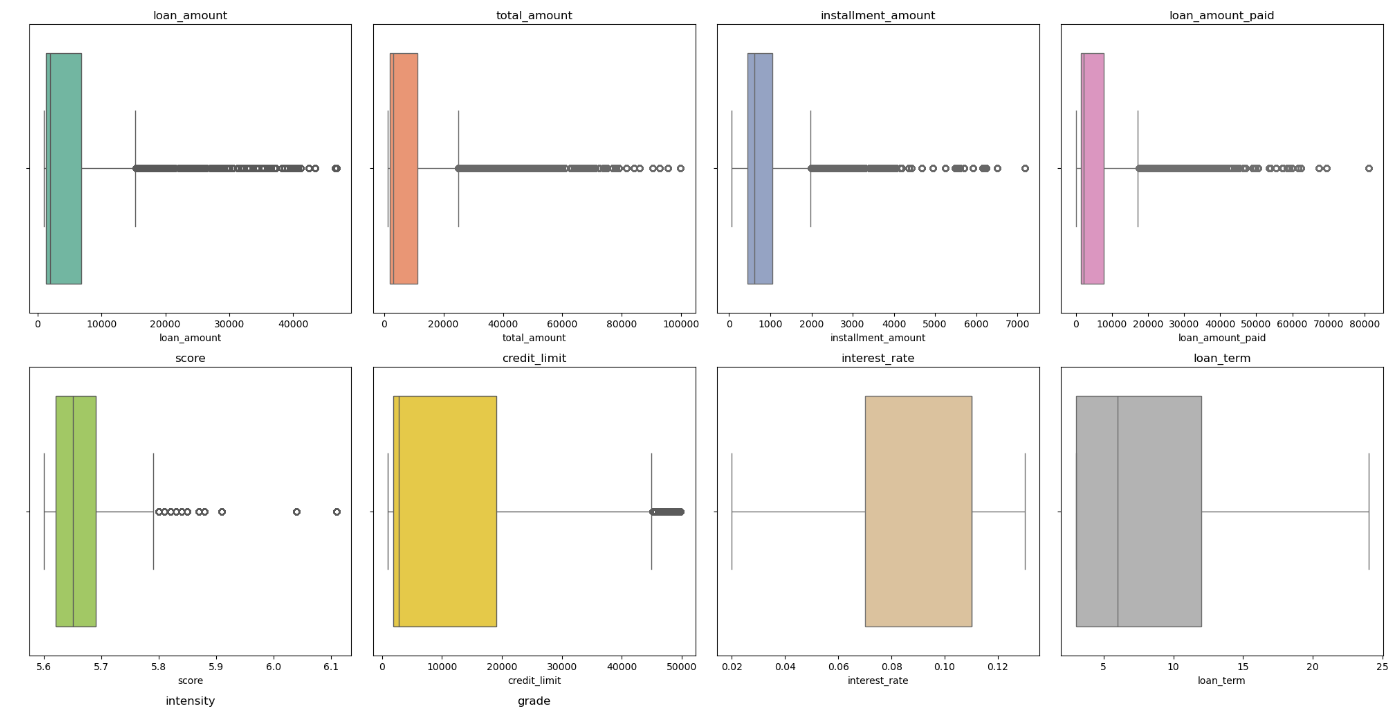
**Seaborn's sns.boxplot:** This function is used to create the boxplots for each numerical column. It shows:

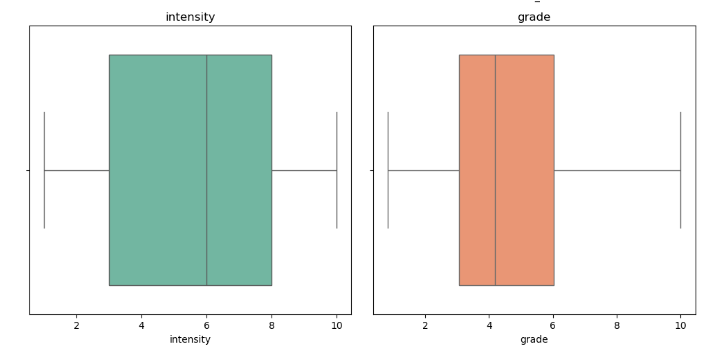
* **Median:** The line inside the box.
* **Interquartile Range (IQR):** The range between the 25th and 75th percentiles (Q1 and Q3).
* **Outliers:** Points outside the whiskers, which are 1.5 times the IQR.

**Color Pallete & Layout Adjustment:**

* **Color Palette:** The color palette Set2 is used to give each boxplot a unique color, improving visual clarity.
* **Boxplot:** For each column in interest\_columns, a boxplot is created using sns.boxplot() and placed in the corresponding position on the grid (axes[i]).
* **Titles:** Each boxplot is labeled with the corresponding column name for easy identification.
* If there are more subplot spaces than columns to plot (because the grid has 12 spaces but only 10 columns), the remaining axes are **deleted** using fig.delaxes(axes[j]
* **palette = sns.color\_palette('Set2'):** This uses a predefined color palette called 'Set2' to assign different colors to each boxplot, making it visually appealing and easier to distinguish between plots.
* This code will produce **10 boxplots** (one for each column in interest\_columns), neatly arranged in a 3x4 grid. It provides an efficient way to check the distribution of these key numerical features.

**Result: Executed (Output of the Screenshot is attached below)**





**Analysis:**

After using the boxplot to find the outliers we found the Outliers in 6 columns (loan\_amount, total\_amount, installment\_amount, loan\_amount\_paid, score, credit\_limit). The bubbles we found after the wick are Outliers / Extreme values which affects ouir further analysis process. So, in the next step I have removed these outliers to clean the data for effective use of analysis process. This visualization is helpful for identifying potential issues in the data, such as extreme outliers that might affect statistical analysis or modeling.

1. **Treating / Removing Outliers:**

**Methods Used to treat Outliers: Winsorization, IQR and Z-Score.**

**Description:** The code is designed to clean the numerical data in a DataFrame (data) by handling outliers through different techniques. The primary methods used include **Winsorization**, **IQR-based filtering**, and **Z-score filtering**. This ensures that the dataset has fewer extreme values that could negatively impact subsequent analysis or modeling.

**Implemented Code:**

import numpy as np

import pandas as pd

from scipy import stats

from scipy.stats import mstats

def winsorize\_data(df, columns, lower\_percentile=0.05, upper\_percentile=0.95):

for column in columns:

# Winsorize the column

df[column] = mstats.winsorize(df[column], limits=[lower\_percentile, 1 - upper\_percentile])

return df

def remove\_outliers\_iqr(df, columns, multiplier=1.5):

for column in columns:

Q1 = df[column].quantile(0.25) # 1st Quartile

Q3 = df[column].quantile(0.75) # 3rd Quartile

IQR = Q3 - Q1

df = df[(df[column] >= (Q1 - multiplier \* IQR)) & (df[column] <= (Q3 + multiplier \* IQR))]

return df

def remove\_outliers\_zscore(df, columns, threshold=3):

for column in columns:

z\_scores = np.abs(stats.zscore(df[column]))

df = df[z\_scores < threshold]

return df

numeric\_columns\_data = [

'loan\_amount', 'total\_amount', 'installment\_amount', 'loan\_amount\_paid', 'score', 'credit\_limit'

]

data = winsorize\_data(data, numeric\_columns\_data)

data = remove\_outliers\_iqr(data, numeric\_columns\_data)

data = remove\_outliers\_zscore(data, numeric\_columns\_data)

print(data.describe())

**Explanation of Code:**

**Winsorization:**

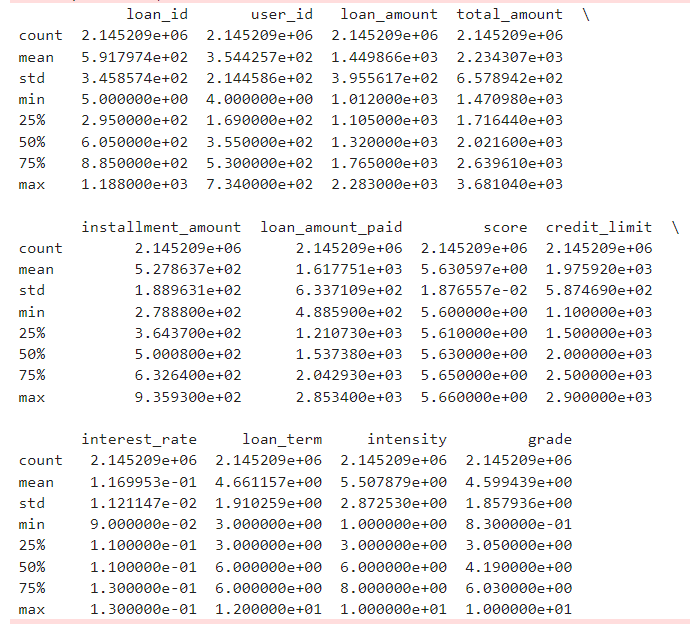
* Winsorization is a technique that limits extreme values in a dataset, effectively reducing the influence of outliers without removing them.
* For each column in columns, it transforms the values by capping the lower and upper extremes at the specified percentiles (here, 5th and 95th percentiles).
* Winsorization helps reduce the effect of extreme outliers while preserving all data points.

**IQR (Interquartile Range) Method:**

* **Lower\_percentile:** The lower limit for Winsorization (5% by default).
* **Upper\_percentile:** The upper limit (95% by default).
* The Interquartile Range (IQR) is the range between the 1st Quartile (Q1) and the 3rd Quartile (Q3) of the data. (IQR = Q3 - Q1.)
* Data points that fall below Q1 - 1.5 \* IQR or above Q3 + 1.5 \* IQR are considered **outliers** and are removed from the dataset.

**Z-Score:**

* The **Z-score** measures how far away a value is from the mean, in terms of standard deviations.
* Values with a Z-score greater than a specified threshold (e.g., 3) are considered **outliers** and are removed. This threshold corresponds to roughly the top and bottom 0.3% of the data for a normal distribution.
* Each of the three outlier treatment functions (winsorize\_data, remove\_outliers\_iqr, and remove\_outliers\_zscore) is applied **sequentially** to the data DataFrame.
* data.describe() outputs a summary of the dataset, showing key statistics like mean, standard deviation, min, and max values after the outlier treatment.

**Result: Executed (Output of the screenshot is attached below)**  
  


**Analysis:** After using above outliers techniques to treat the outliers all the outliers were removed. To check the outliers were present or not I again used the box plot graph to check the same.

1. **Checking whether the outliers are present are not using boxplot after outlier treatment.**

**Description:** After removing the outliers using the above 3 techniques. We need to check whether the outliers are still existing or not. To cross check and verify I again used the boxplot to check the same.

**Implemented code: (Used the above same code of boxplot to check if Outlier is present or not)**

import seaborn as sns

import matplotlib.pyplot as plt

interest\_columns = ['loan\_amount',

'total\_amount',

'installment\_amount',

'loan\_amount\_paid',

'score',

'credit\_limit'

]

nrows = 3

ncols = 4

fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(20, 15))

axes = axes.flatten()

palette = sns.color\_palette('Set2', len(interest\_columns)) # Using a color palette

for i, column in enumerate(interest\_columns):

sns.boxplot(x=data[column], ax=axes[i], color=palette[i])

axes[i].set\_title(column)

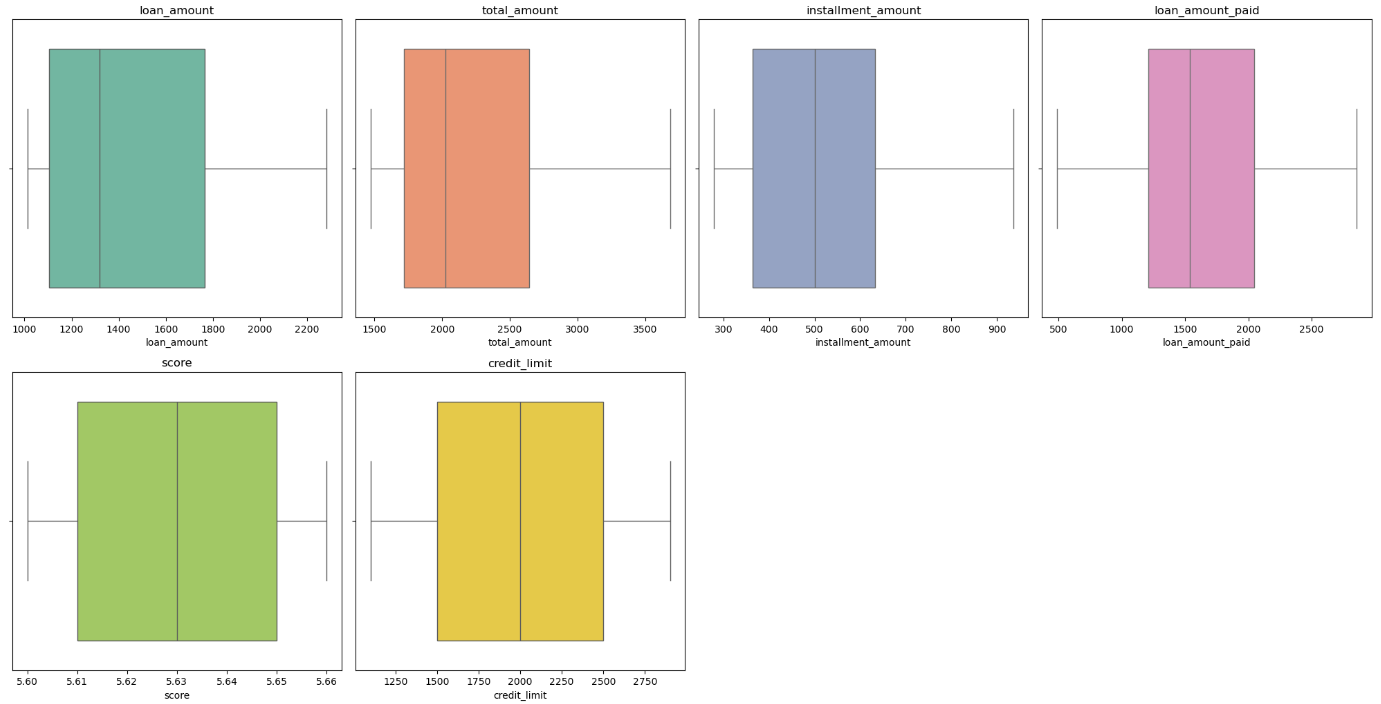
for j in range(len(interest\_columns), len(axes)):

fig.delaxes(axes[j])

plt.tight\_layout()

plt.show()

**Result: Executed (Output screenshot attached below)**

  
  
**Analysis:** After using the Winsorization, IQR & Z-Score technique, outliers were removed.  
Earlier, outliers were present in the column loan\_amount, total\_amount, installment\_amount, loan\_amount\_paid, score, credit\_limit. By treating the outliers, it was successfully removed. It can be seen through above code.

**Step 6: Four Moment Business Decisions**

**Description:** By using the 4 moment business decisions we will get the information about the data distribution, i.e., normal or abnormal distribution of the data.

1. **Measures of Central Tendency / First Moment Business Decisions**

**Implemented Code:**

import pandas as pd

numerical\_columns = ['loan\_amount', 'total\_amount', 'installment\_amount', 'loan\_amount\_paid', 'score', 'credit\_limit', 'interest\_rate', 'loan\_term', 'intensity', 'grade']

categorical\_columns = ['approved\_date', 'denied\_date', 'timestamp', 'time\_of\_day',

'primary\_emotion', 'relationship', 'situation', 'location',

'weather', 'physical\_state', 'preceding\_event', 'issue\_date',

'due\_date', 'paid\_date', 'status']

mean\_values = data[numerical\_columns].mean()

print("Mean (Numerical Columns):")

print(mean\_values)

median\_values = data[numerical\_columns].median()

print("\nMedian (Numerical Columns):")

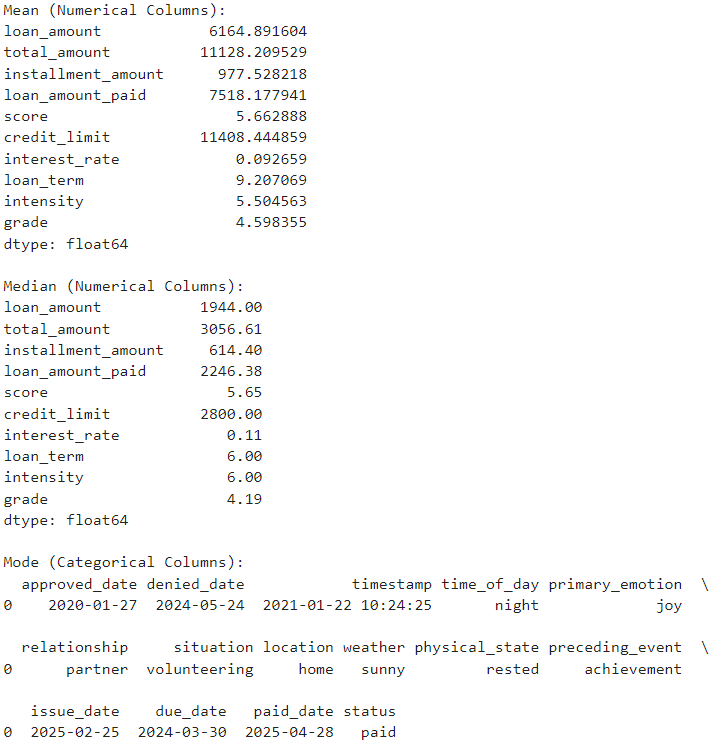
print(median\_values)

print("\nMode (Categorical Columns):")

print(mode\_values)  
  
**Explanation of Code:**

* mean(): The mean() function calculates the average value of each numerical column in your dataset.
* median(): The median() function calculates the middle value of each numerical column.
* mode(): The mode() represents the most frequent value in each categorical column. This is used for only categorical column.

**Result: Executed (Output Screenshot attached below)**



**Analysis:** Mean and Median values are not so close hence all the columns are not symmetrically distributed. Hence the transformation of the data is very much necessary.  
  
**Mean (Numerical Columns)**:

* **Loan Amount (6,164.89)**: The average loan amount is around 6,164 units. This suggests that, on average, the users have borrowed a moderate amount of loans.
* **Total Amount (11,128.20)**: This reflects the total repayment amount (including interest), which is almost double the loan amount, indicating significant interest or fees on top of the borrowed amount.
* **Installment Amount (977.52)**: The average installment paid is 977 units, showing how much borrowers are paying in each installment.
* **Score (5.66)**: This likely represents an average user score based on the emotional data.
* **Credit Limit (11,408.44)**: The average credit limit provided to users is around 11,408 units.
* **Interest Rate (0.0926)**: The average interest rate is about 9.26%, a typical figure for personal or small business loans.
* **Intensity (5.50)**: The emotional intensity averages around 5.5, showing that users tend to experience moderate emotional intensity.

**Median (Numerical Columns)**:

* **Loan Amount (1,944.00)**: The median value is much lower than the mean, indicating that many users have taken small loans, but a few large loans are pulling the mean up.
* **Installment Amount (614.40)**: Half of the borrowers pay installments less than 614 units, and the other half pay more.
* **Credit Limit (2,800.00)**: The median credit limit is much smaller than the mean, showing that many users have lower credit limits, while a few have very high limits.
* **Loan Term (6.00)**: The typical loan duration is 6 months, which aligns with the shorter-term nature of many personal or small business loans.

**Mode (Categorical Columns)**:

* **Approved Date (2020-01-27)**: The most common loan approval date is 2020-01-27.
* **Primary Emotion (Joy)**: The most frequent emotion reported is **joy**, indicating that users often experience positive emotions during the loan application or usage process.
* **Relationship (Partner)**: The most common relationship context is **partner**, showing that users are often with their significant other when using the service.
* **Situation (Volunteering)**: Many users seem to be in a **volunteering** situation during the emotional records, which might indicate positive emotions associated with these activities.
* **Location (Home)**: The most frequent location is **home**, likely because users tend to access the service or loan-related activities from home.
* **Weather (Sunny)**: The most common weather condition reported is **sunny**.
* **Status (Paid)**: The most common loan status is **paid**, which is a positive indicator of loan repayment.

1. **Measures of Dispersion / Second Moment Business Decision:**

**Description:** Variance, standard deviation, and range are essential for understanding how much the data varies, which is crucial for modeling and decision-making.

**Implemented Code:**

print("\nVariance (Data):", data[numerical\_columns].var())

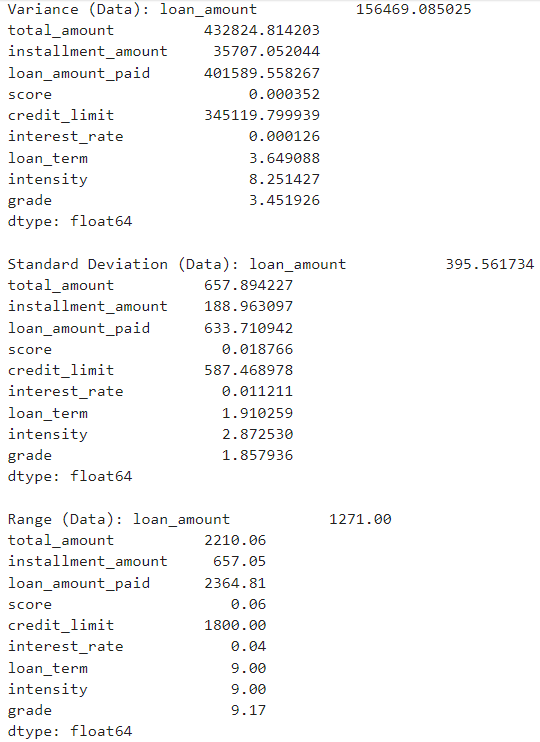
print("\nStandard Deviation (Data):", data[numerical\_columns].std())

print("\nRange (Data):", data[numerical\_columns].max() - data[numerical\_columns].min())

**Explanation of Code:**

* Variance measures how much the data points in a dataset deviate from the mean. It tells you the degree of spread in the data.
* A high variance indicates that the data points are spread out far from the mean.
* A low variance means that the data points are close to the mean.
* Standard deviation (SD) is the square root of variance and is also a measure of spread. It is in the same units as the data, making it easier to interpret than variance.
* A high SD means the data points are more spread out from the mean.
* A low SD indicates that the data points are closely clustered around the mean.
* Purpose: The range is the difference between the maximum and minimum values in a dataset. It gives a rough estimate of the spread of data.
* The formula is: Range = data[numerical\_columns].max() - data[numerical\_columns].min()
* Range is useful for quickly understanding the extent of variability in the data

**Result: Executed (Output of the screenshot is attached below).**



**Analysis:**

**Variance (Data):**

* **Loan Amount (156,469.08)**: The high variance indicates a large spread in the loan amounts, meaning that the loan values differ significantly from the mean.
* **Total Amount (432,824.81)**: Similarly, this high variance suggests a large variability in total amounts. This could be due to a mix of high and low loan disbursements.
* **Score (0.000352)**: The variance is extremely low, which means that the values in this column are almost the same or have very little spread.
* **Credit Limit (345,119.79)**: There's considerable variability in the credit limits issued.

**Standard Deviation (Data):**

* **Loan Amount (395.56)**: On average, the loan amounts deviate by around 395 units from the mean.
* **Installment Amount (188.96)**: The deviation from the mean for installment amounts is 188 units, indicating moderate spread.
* **Intensity (2.87)**: Emotional intensity has a moderate standard deviation, suggesting varying degrees of intensity across users.
* **Interest Rate (0.011)**: The interest rates show a very small deviation, meaning they are fairly consistent across the dataset.

**Range (Data):**

* **Loan Amount (1271)**: The range is 1271, meaning the highest loan amount is 1271 units more than the smallest loan amount.
* **Total Amount (2210.06)**: The spread between the maximum and minimum total amounts is about 2210 units.
* **Score (0.06)**: The small range of 0.06 means the scores are very similar across the data.
* **Intensity (9.00)**: The range for emotional intensity is 9, meaning users report a wide range of emotional intensities.

1. **Skewness / Third Moment Business Decision**

**Description:** The skewness calculates the asymmetry of the distribution of values in each numerical column.

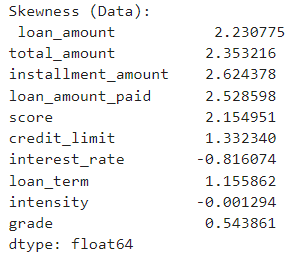
**Implemented Code:**

print("\nSkewness (Data):\n", data[numerical\_columns].skew())

**Explanation of Code:**

* **skew():** The skew() function calculates the asymmetry of the distribution of values in each numerical column.
* **Skewness Value ≈ 0:** The data is symmetrically distributed (normal distribution).
* **Skewness > 0:** The data is right-skewed or positively skewed, meaning the tail on the right side is longer or fatter. This indicates that there are outliers on the higher side of the distribution.
* **Skewness < 0:** The data is left-skewed or negatively skewed, meaning the tail on the left side is longer or fatter. This indicates outliers on the lower side of the distribution.

**Result: Executed (Output of the screenshot is attached below)**



**Analysis:**

* **Loan Amount (2.23):** Right-skewed (positively skewed). The distribution has a long tail on the right side, meaning there are some high-value loan amounts, while most are smaller. We want to apply a log transformation to reduce skewness.
* **Total Amount (2.35):** Highly right-skewed. Similar to loan\_amount, a few large total amounts are dragging the distribution to the right. A log transformation could help here as well.
* **Installment Amount (2.62):** Highly right-skewed. Most installment amounts are small, with a few high-value installments creating the skew. A log transformation or Box-Cox transformation would be suitable to normalize the data.
* **Loan Amount Paid (2.53):** Right-skewed. There are a few users who have paid large loan amounts, while most paid smaller amounts. Use a log transformation to deal with this skewness.
* **Score (2.15):** Right-skewed. There are outliers with high scores, but most users have low or average scores. Consider normalization or transformations to make this distribution more symmetric.
* **Credit Limit (1.33):** Right-skewed. There are a few users with high credit limits, while most have lower limits. Log transformation would help here as well.
* **Interest Rate (-0.82):** Left-skewed (negatively skewed). Most users have higher interest rates, with some outliers having very low interest rates. No immediate transformation is needed here unless you want to treat the skew.
* **Loan Term (1.16):** Right-skewed. There are some users with very long loan terms compared to the majority. You might consider applying a log transformation to reduce skewness.
* **Intensity (-0.001):** Almost symmetric (close to 0). This is nearly a normal distribution, so no transformation is needed.
* **Grade (0.54):** Slightly right-skewed. This distribution is moderately skewed to the right, and may benefit from mild transformations.

Hence, we need to apply the transformation techniques for all the numerical columns.

1. **Kurtosis / Fourth Moment Business Decision:**

**Description: Kurtosis** measures the "tailedness" of a distribution, or how heavily the tails of the distribution differ from the tails of a normal distribution.

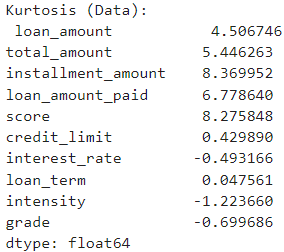
**Implemented Code:**

print("\nKurtosis (Data):\n", data[numerical\_columns].kurt())

**Explanation of Code:**

* **kurt():** **Kurtosis** measures the "tailedness" of a distribution, or how heavily the tails of the distribution differ from the tails of a normal distribution.
* **Positive kurtosis (>3):** Heavier tails than a normal distribution (more outliers).
* **Negative kurtosis (<3)**: Lighter tails than a normal distribution (fewer outliers).

**Result: Executed (Output screenshot of the code is attached below)**



**Analysis:**

* **Skewness > 0:** The data is right-skewed or positively skewed, meaning the tail on the right side is longer or fatter. This indicates that there are outliers on the higher side of the distribution.
* **Skewness < 0**: The data is left-skewed or negatively skewed, meaning the tail on the left side is longer or fatter. This indicates outliers on the lower side of the distribution.
* **Loan Amount (2.23)**: **Right-skewed (positively skewed)**. The distribution has a long tail on the right side, meaning there are some high-value loan amounts, while most are smaller. We want to apply a **log transformation** to reduce skewness.
* **Total Amount (2.35)**: **Highly right-skewed**. Similar to loan\_amount, a few large total amounts are dragging the distribution to the right. A **log transformation** could help here as well.
* **Installment Amount (2.62)**: **Highly right-skewed**. Most installment amounts are small, with a few high-value installments creating the skew. A **log transformation** or **Box-Cox transformation** would be suitable to normalize the data.
* **Loan Amount Paid (2.53)**: **Right-skewed**. There are a few users who have paid large loan amounts, while most paid smaller amounts. Use a **log transformation** to deal with this skewness.
* **Score (2.15)**: **Right-skewed**. There are outliers with high scores, but most users have low or average scores. Consider normalization or transformations to make this distribution more symmetric.
* **Credit Limit (1.33)**: **Right-skewed**. There are a few users with high credit limits, while most have lower limits. **Log transformation** would help here as well.
* **Interest Rate (-0.82)**: **Left-skewed (negatively skewed)**. Most users have higher interest rates, with some outliers having very low interest rates. **No immediate transformation** is needed here unless you want to treat the skew.
* **Loan Term (1.16)**: **Right-skewed**. There are some users with very long loan terms compared to the majority. You might consider applying a **log transformation** to reduce skewness.
* **Intensity (-0.001)**: **Almost symmetric (close to 0)**. This is nearly a normal distribution, so no transformation is needed.
* **Grade (0.54)**: **Slightly right-skewed**. This distribution is moderately skewed to the right, and may benefit from mild transformations.

Hence, we need to use the Transformation technique to normalize the data.

**Step 7: Graphical Representation for better Understanding of Distribution of Data**

**Description:** The purpose of this code is to generate density (KDE) plots for the numerical columns in the DataFrame (data) while also displaying key statistics such as **mean**, **median**, **variance**, **standard deviation**, **skewness**, and **kurtosis** for each column. This visualization helps understand the distribution and central tendency of each numerical feature. Here, I have only provided the code of KDE Plot. But I also used Histogram and Q-Q Plot at last I have plotted all 3-plot side by side. Attached the Screenshot in the output section.

**Implemented Code:**

import matplotlib.pyplot as plt

import seaborn as sns

from scipy.stats import skew, kurtosis

data = data

columns = numerical\_columns

fig, axes = plt.subplots(nrows=len(columns), ncols=1, figsize=(8, 6 \* len(columns)))

for i, column in enumerate(columns):

data\_column = data[column].dropna()

mean\_val = data\_column.mean()

median\_val = data\_column.median()

variance\_val = data\_column.var()

std\_dev\_val = data\_column.std()

skewness\_val = skew(data\_column)

kurtosis\_val = kurtosis(data\_column)

sns.kdeplot(data\_column, fill=True, ax=axes[i], color='cyan')

axes[i].set\_title(f'Density Plot for {column} with Statistics')

axes[i].set\_xlabel(column)

axes[i].set\_ylabel('Density')

axes[i].axvline(mean\_val, color='red', linestyle='dashed', linewidth=2, label=f'Mean: {mean\_val:.2f}')

axes[i].axvline(median\_val, color='green', linestyle='dashed', linewidth=2, label=f'Median: {median\_val:.2f}')

annotations = [

f'Variance: {variance\_val:.2f}',

f'Standard Deviation: {std\_dev\_val:.2f}',

f'Skewness: {skewness\_val:.2f}',

f'Kurtosis: {kurtosis\_val:.2f}'

]

for j, annotation in enumerate(annotations):

axes[i].text(1.1, 0.85 - 0.05 \* j, annotation, transform=axes[i].transAxes, fontsize=10, color='blue')

axes[i].legend()

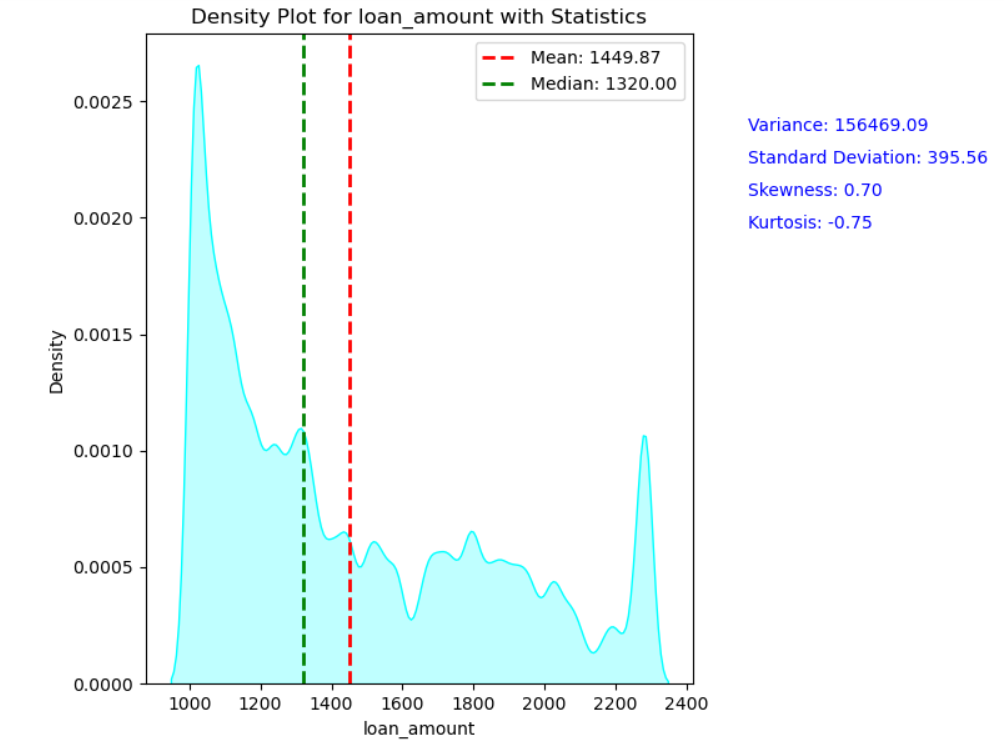
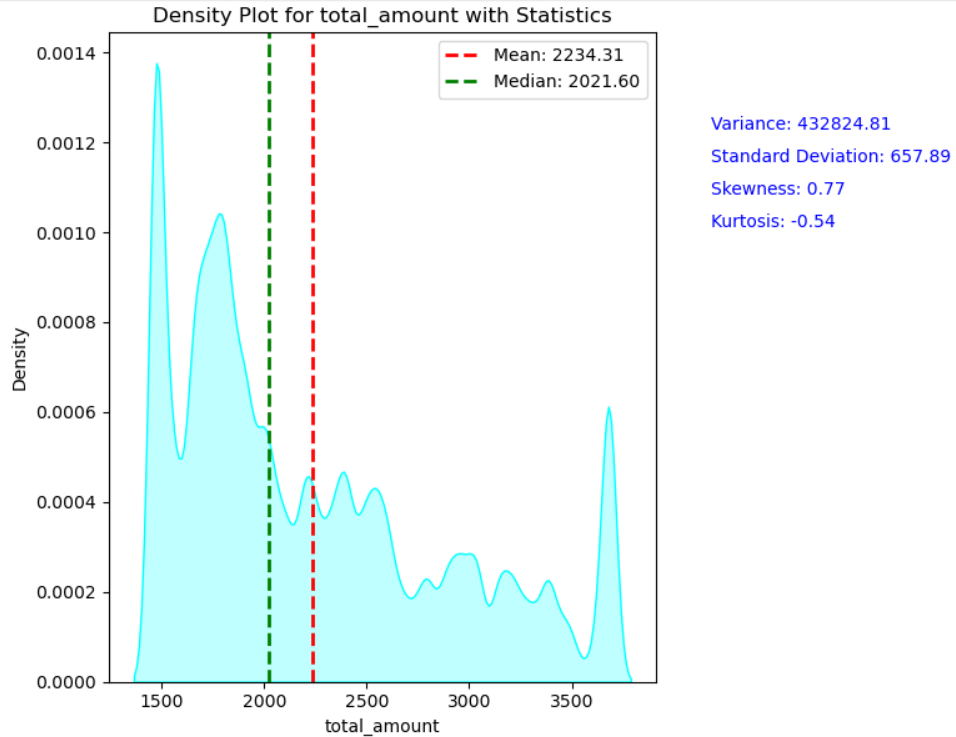
plt.tight\_layout()

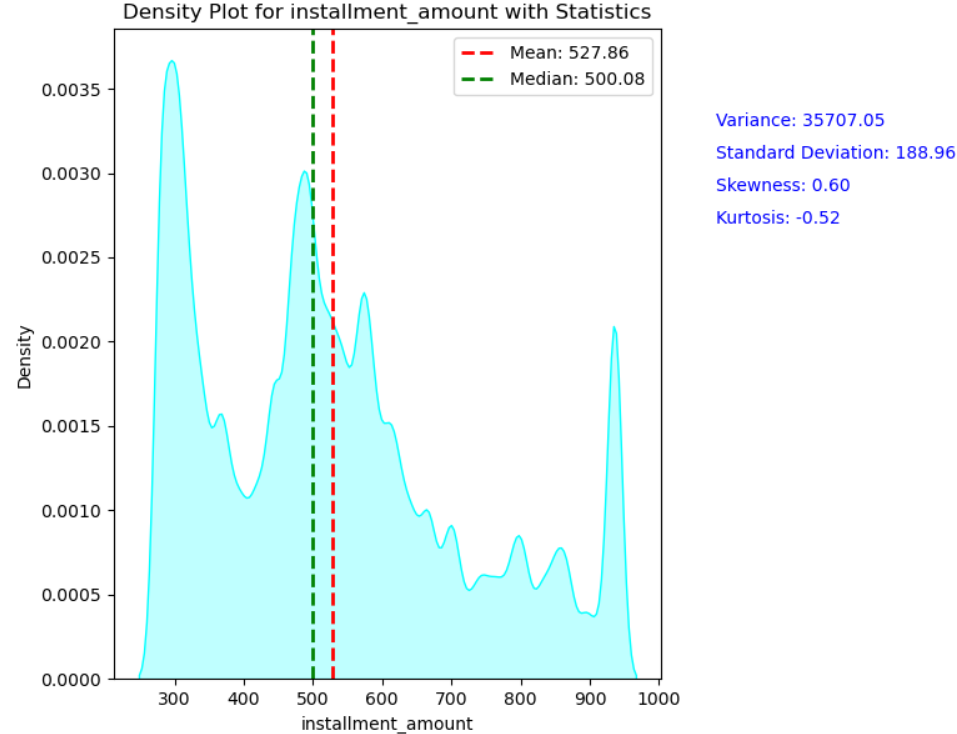
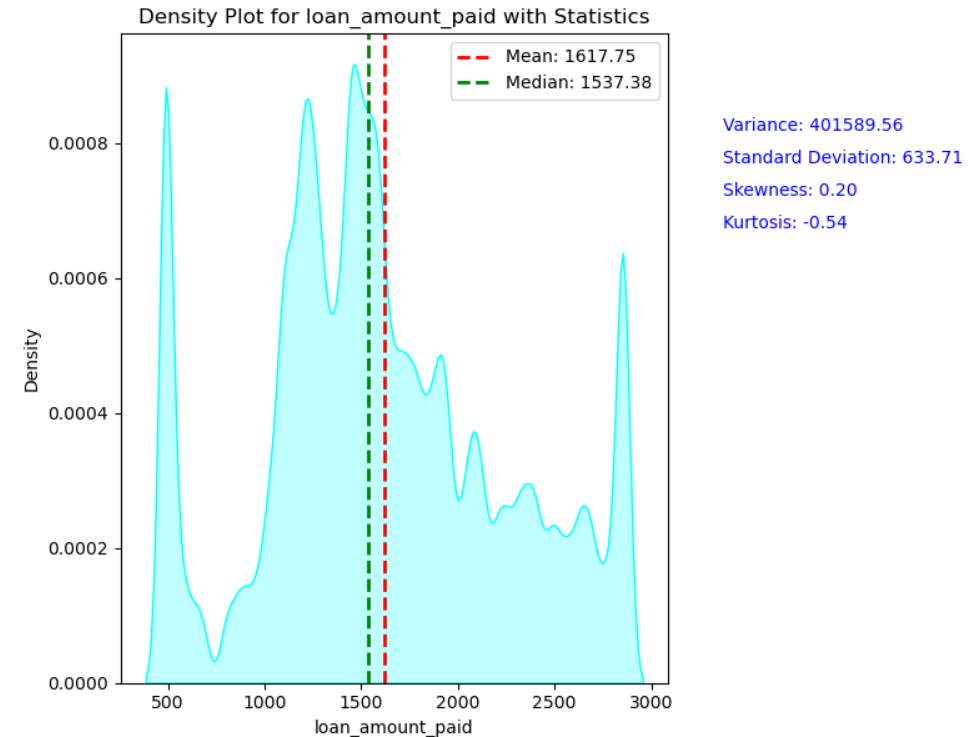
plt.show()

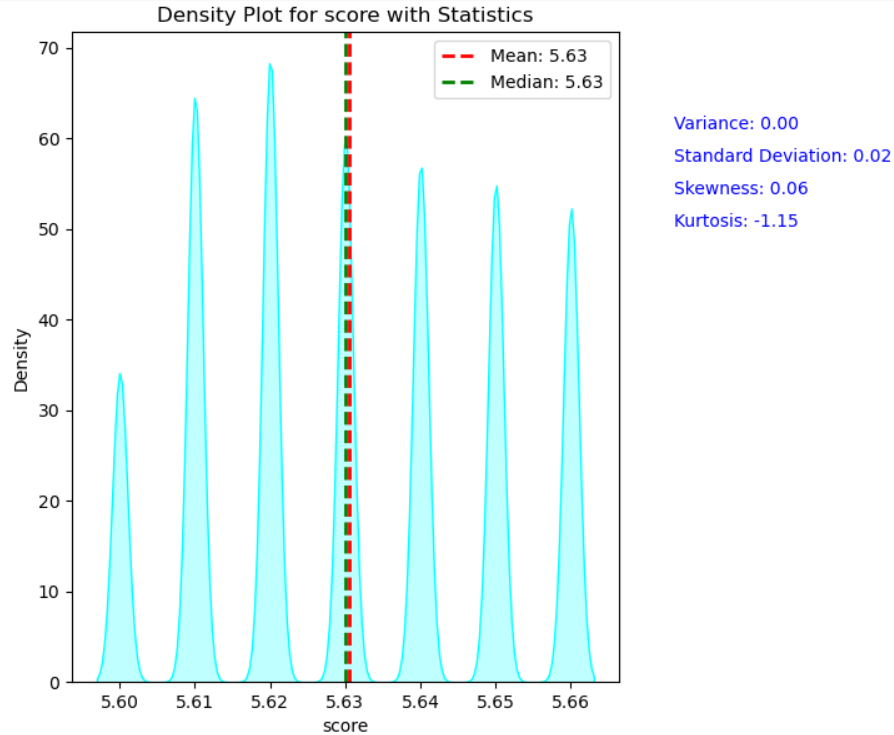
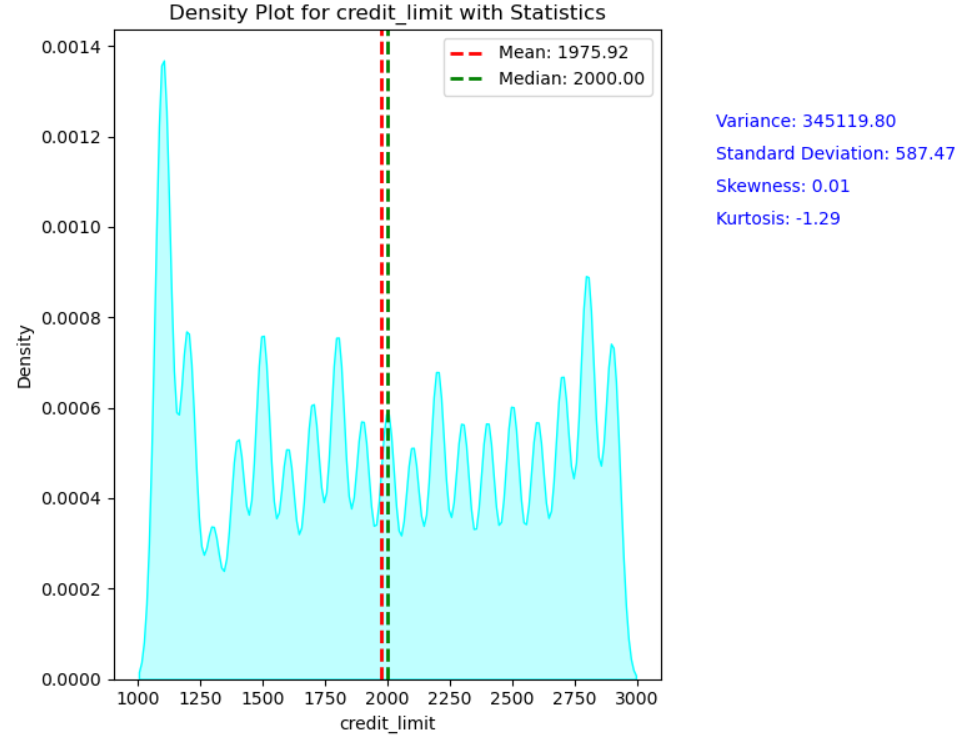
**Explanation of Code:**

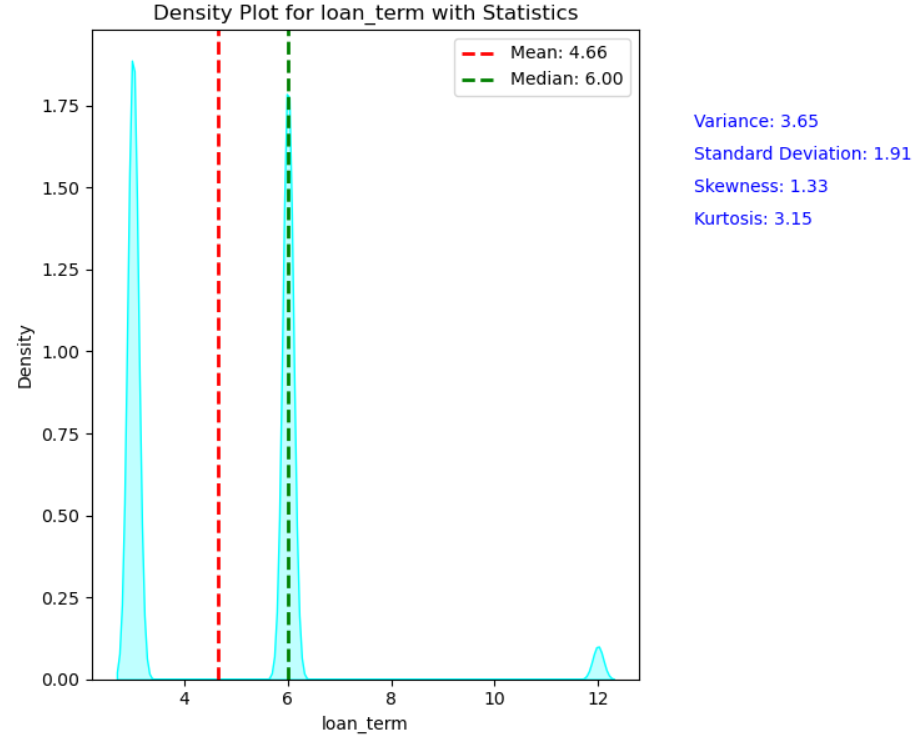
* data = data ensures the variable references your dataset.
* columns = numerical\_columns specifies which columns will be used for the density plots.
* **Subplot Creation:** fig, axes = plt.subplots(...) creates a figure and axes for plotting multiple density plots. The number of subplots (nrows) corresponds to the number of numerical columns you're analyzing.
* **Loop through Columns:** The for i, column in enumerate(columns) loop goes through each column in numerical\_columns and generates the plot for it.
* data\_column = data[column].dropna() removes any missing values in the column to avoid errors during plotting.
* **Calculate Statistics:** mean\_val, median\_val, variance\_val, std\_dev\_val, skewness\_val, and kurtosis\_val are calculated for each column to display relevant statistics:
* **Mean:** Average value.
* **Median:** Middle value.
* **Variance:** Measure of data dispersion.
* **Standard Deviation:** Square root of variance.
* **Skewness:** Indicates asymmetry (positive means right skewed, negative means left skewed).
* **Kurtosis:** Indicates the "tailedness" of the data distribution (positive means heavier tails).
* **Plotting the KDE (Kernel Density Estimate):** sns.kdeplot(data\_column, fill=True, ax=axes[i], color='cyan') creates a smooth density plot to visualize the distribution of the column.
* **Adding Mean and Median:** Two vertical dashed lines are added to show where the mean (red) and median (green) values are located on the plot.
* **Annotating the Plots:** The text for variance, standard deviation, skewness, and kurtosis is placed on the right side of each plot using axes[i].text(...).
* **Legend and Layout:** A legend is included for the mean and median, and plt.tight\_layout() ensures there is no overlap between the plots.

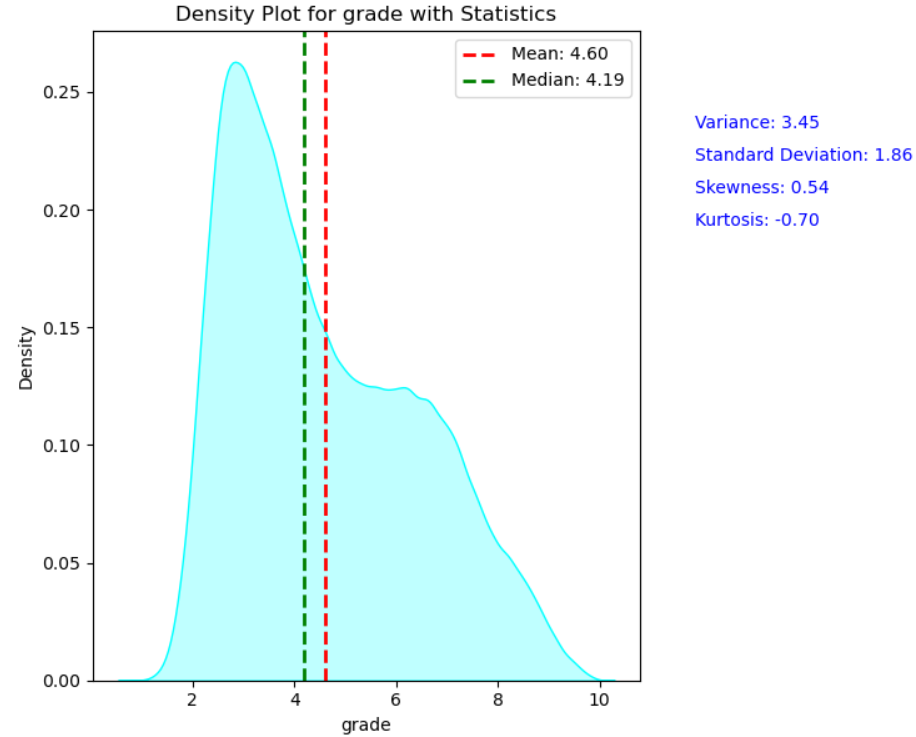
**Result: Executed (Output of the screenshot is attached below)**

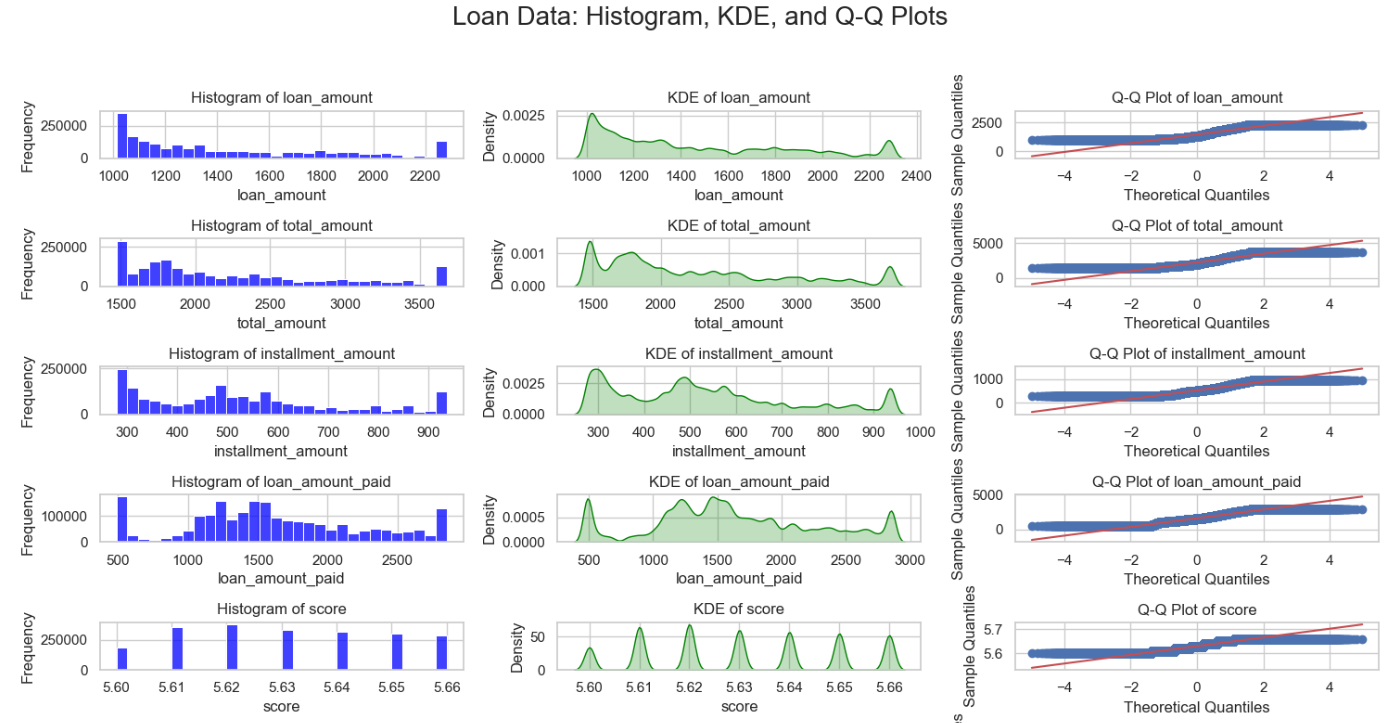
 

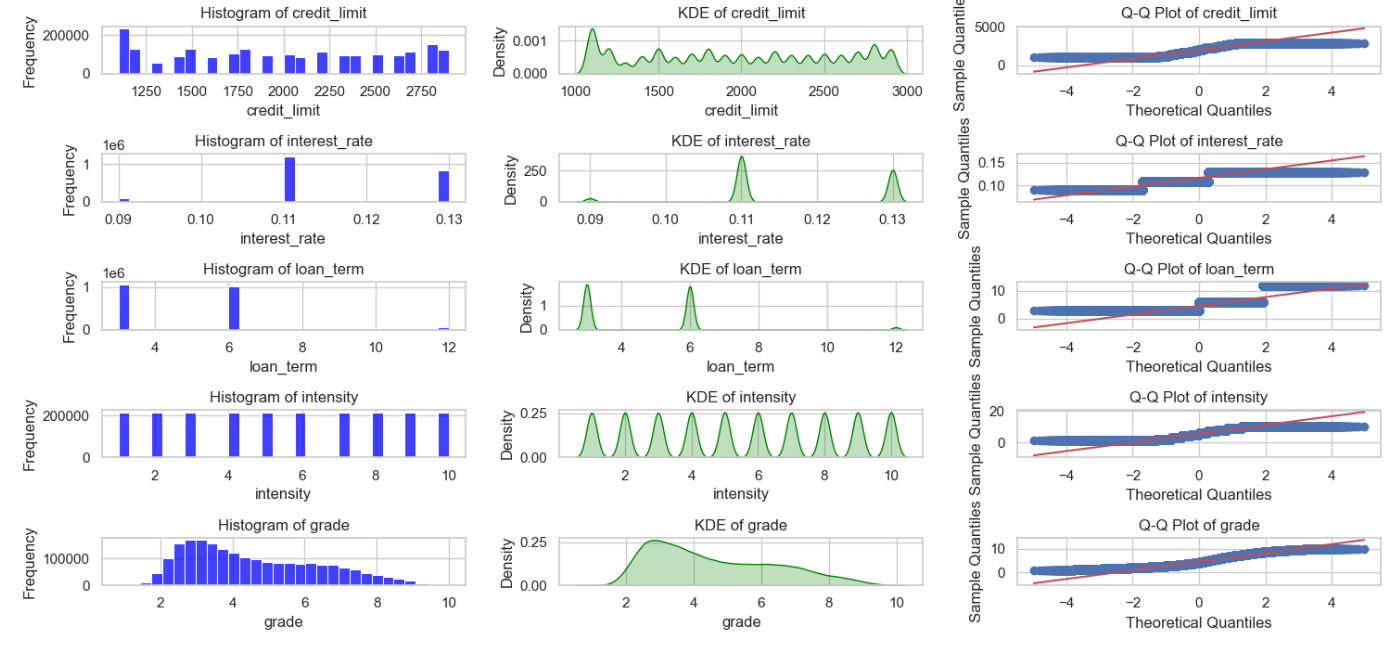
 

**Analysis:** Here as I mentioned earlier, we need to transform the data to make it distribute normally. We used this graphical representation for better understanding of distribution of data.  
  
**Output: As I mentioned earlier, I used three plots to know distribution correctly. I added all the 3 graphs in single code.**





**Step 8: Explanation of Score, Intensity, Loan Term & Interest Rate Abnormal Distribution:**

**Analysis:** In above graph we can clearly visualize that the column Score, Intensity, Loan Term and Interest rate are very abnormal. This is because the data in that column has less unique values & more repeated values. I used the below code to find the unique values.

**Implemented Code:**

score\_value\_counts = data['score'].value\_counts()

intensity\_value\_counts = data['intensity'].value\_counts()

loan\_term\_value\_counts = data['loan\_term'].value\_counts()

interest\_rate\_value\_counts = data['interest\_rate'].value\_counts()

print("Top repeated values for 'score':")

print(score\_value\_counts.head(10))

print("\nTop repeated values for 'intensity':")

print(intensity\_value\_counts.head(10))

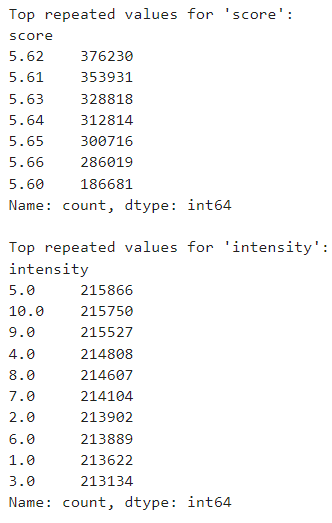
print("\nTop repeated values for 'loan\_term':")

print(loan\_term\_value\_counts.head(10))

print("\nTop repeated values for 'interest\_rate':")

print(interest\_rate\_value\_counts.head(10))  
  
**Explanation of Code:**

value\_counts(); It counts the how many unique values are present in the column.  
  
**Result: Executed (Output of the screenshot is attached below)**

**Analysis:**

* In score we have 7 unique values.  
  In intensity we have 10 unique values.
* In Loan Term we have only 3 values.
* In Interest Rate we have only 3 values.

Hence, we can conclude that when we have less values obviously, we will be having repeated values. So, when the repeated values will be there graph looks abnormally.

**Step 9: Transforming Data to Normal Distribution**

**Description:** This code aims to apply several **data transformation techniques** to ensure that numerical features in the dataset follow a more **normal distribution**. By transforming these features, we can enhance model performance in machine learning tasks, especially for models that assume normality.

**Implementation Code:**

import numpy as np

import pandas as pd

from scipy import stats

from sklearn.preprocessing import PowerTransformer, StandardScaler

import matplotlib.pyplot as plt

import seaborn as sns

data\_transformed = data.copy()

columns\_log = ['loan\_amount', 'total\_amount', 'installment\_amount', 'loan\_amount\_paid', 'credit\_limit']

data\_transformed[columns\_log] = data\_transformed[columns\_log].apply(lambda x: np.log1p(x)) # log1p = log(x + 1)

columns\_sqrt = ['intensity', 'grade']

data\_transformed[columns\_sqrt] = data\_transformed[columns\_sqrt].apply(lambda x: np.sqrt(x))

columns\_boxcox = ['loan\_amount', 'total\_amount', 'credit\_limit', 'grade'] # Use positive columns

for col in columns\_boxcox:

data\_transformed[col], \_ = stats.boxcox(data\_transformed[col] + 1) # +1 to avoid zero values

columns\_yeo\_johnson = ['loan\_amount\_paid', 'installment\_amount'] # Example columns

pt = PowerTransformer(method='yeo-johnson') # Initialize the transformer

data\_transformed[columns\_yeo\_johnson] = pt.fit\_transform(data\_transformed[columns\_yeo\_johnson])

columns\_standardize = ['score', 'interest\_rate', 'loan\_term'] # Additional columns for standardization

scaler = StandardScaler()

data\_transformed[columns\_standardize] = scaler.fit\_transform(data\_transformed[columns\_standardize])

print(data\_transformed.head())

def plot\_comparison(original, transformed, columns):

for col in columns:

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

sns.histplot(original[col], kde=True)

plt.title(f'Original {col}')

plt.subplot(1, 2, 2)

sns.histplot(transformed[col], kde=True)

plt.title(f'Transformed {col}')

plt.show()

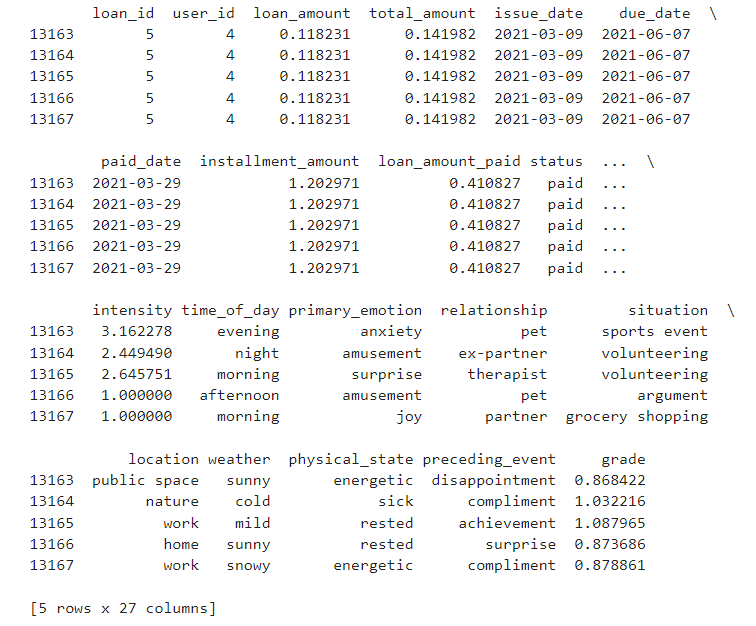
columns\_to\_check = ['loan\_amount', 'total\_amount', 'installment\_amount', 'loan\_amount\_paid', 'score', 'credit\_limit', 'interest\_rate', 'loan\_term', 'intensity', 'grade']

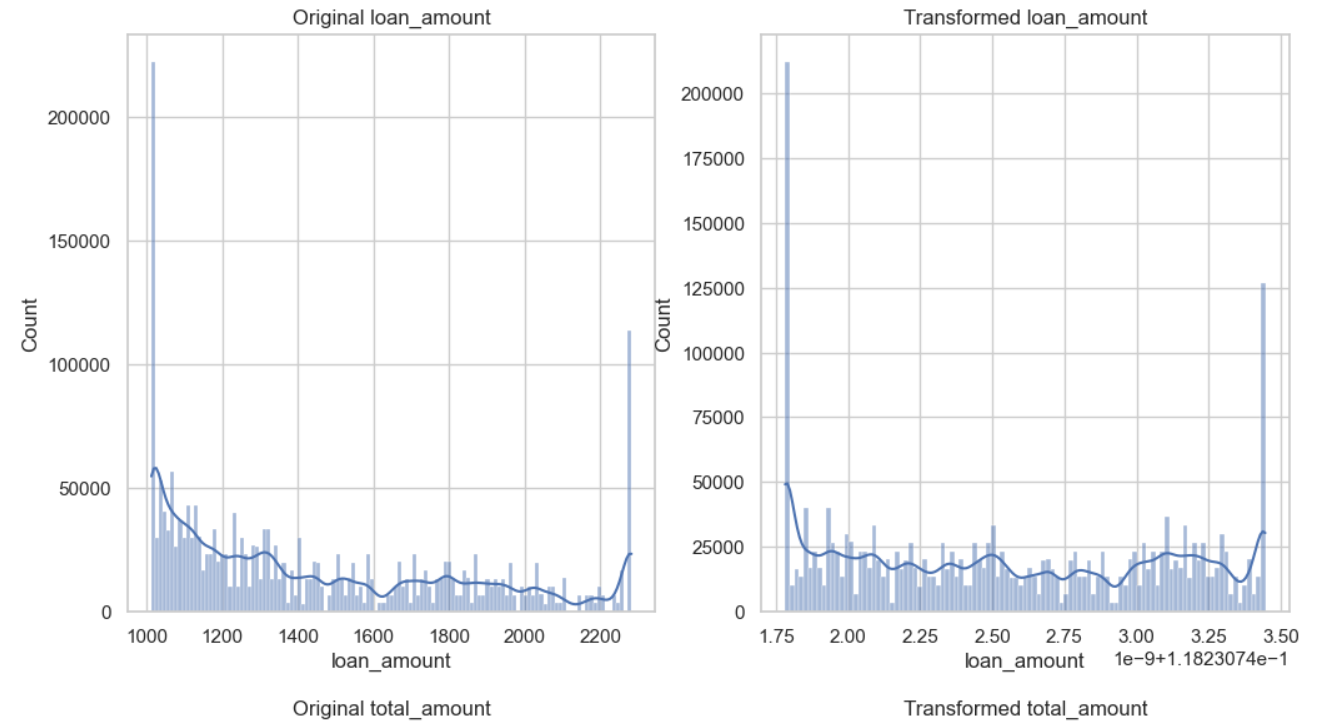
plot\_comparison(data, data\_transformed, columns\_to\_check)

**Explanation of Code:**

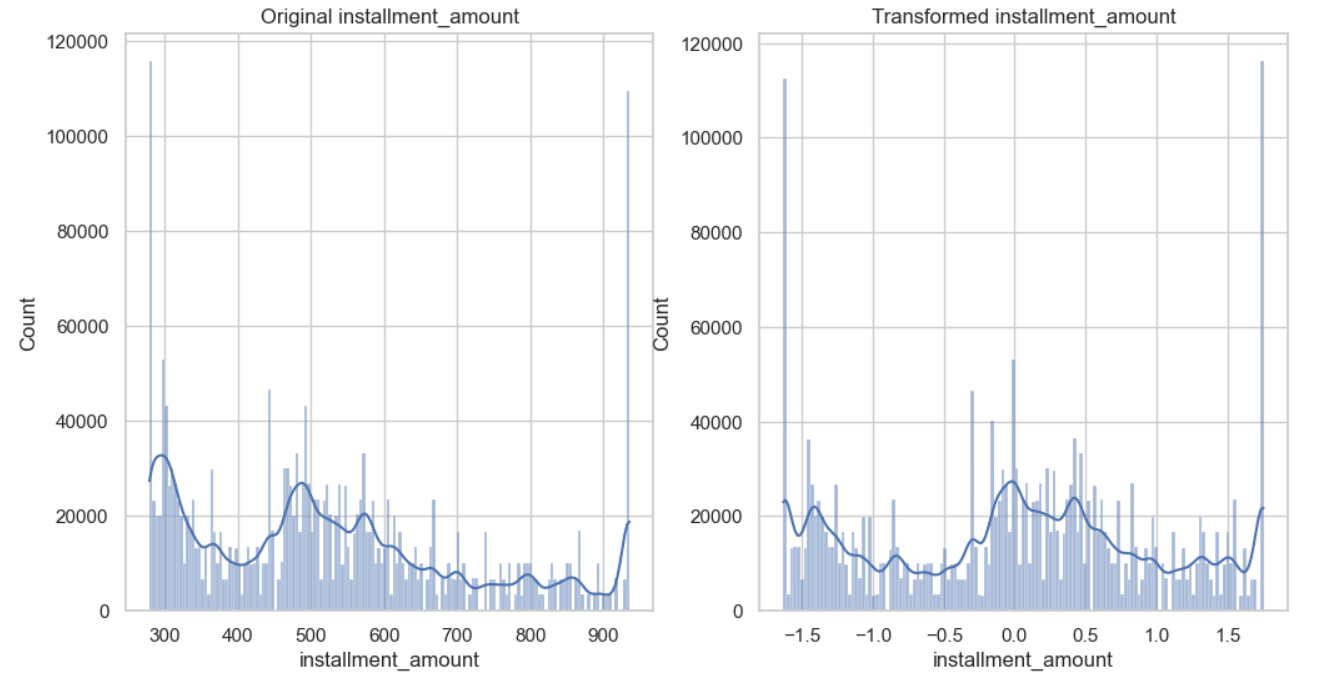
* **numpy, pandas:** Essential libraries for data manipulation.
* **scipy.stats:** Used for statistical functions like Box-Cox transformation.
* **sklearn.preprocessing**: Provides PowerTransformer and StandardScaler for Yeo-Johnson transformation and standardization.
* **matplotlib.pyplot, seaborn**: For plotting histograms to compare original vs. transformed data.
* **Square Root Transformation:** Applies the square root to the intensity and grade columns. This is useful for reducing the skewness of moderately skewed data.
* **Box-Cox Transformation:** This transformation is applied to columns that contain only positive values. The +1 ensures that the transformation can handle zero values. Box-Cox aims to make data more normally distributed.
* **Yeo-Johnson Transformation:** Unlike Box-Cox, this method works for both positive and negative data. It’s applied to the loan\_amount\_paid and installment\_amount columns.
* **StandardScaler:** This scales the selected columns (score, interest\_rate, loan\_term) to have a mean of 0 and a standard deviation of 1.
* **Visualization:** A function plot\_comparison is defined to display side-by-side histograms for each column before and after transformation.
* **KDE (Kernel Density Estimate):** This visualizes the probability density of the data. By comparing original vs. transformed data, we can see if the transformation successfully made the data more normally distributed.

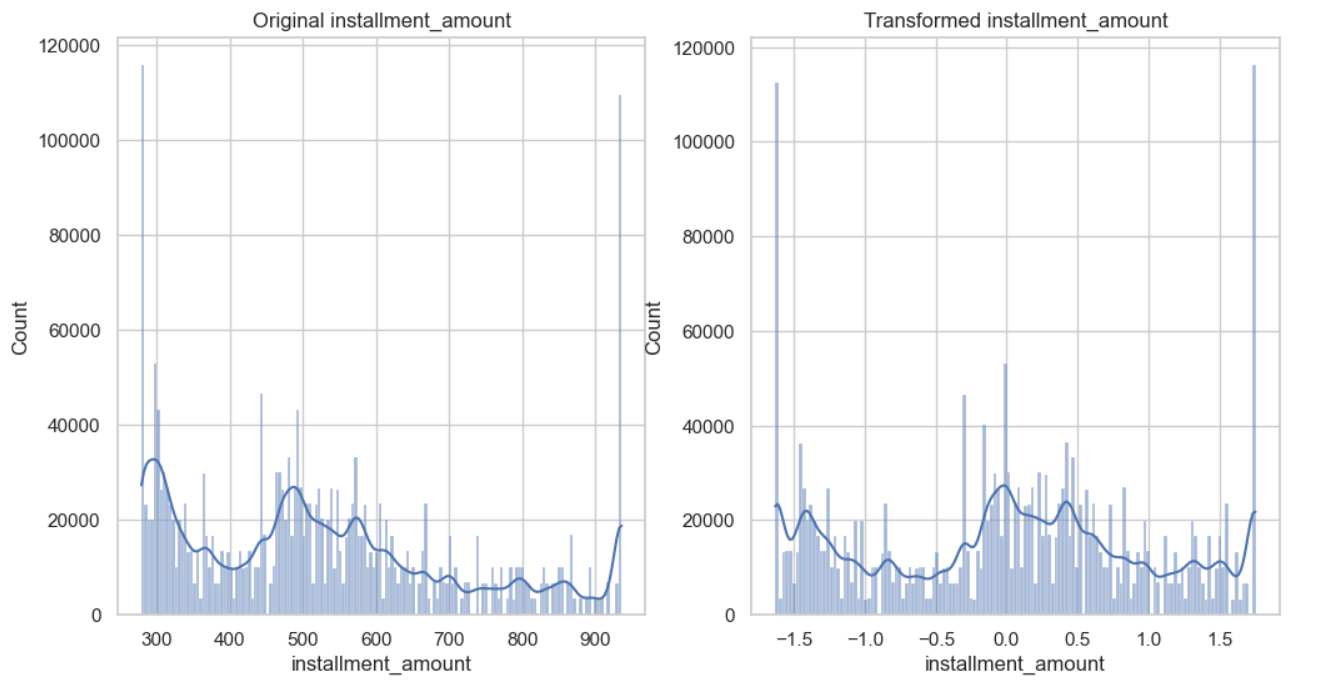
**Result: Executed (Output of the screenshot is attached below)**

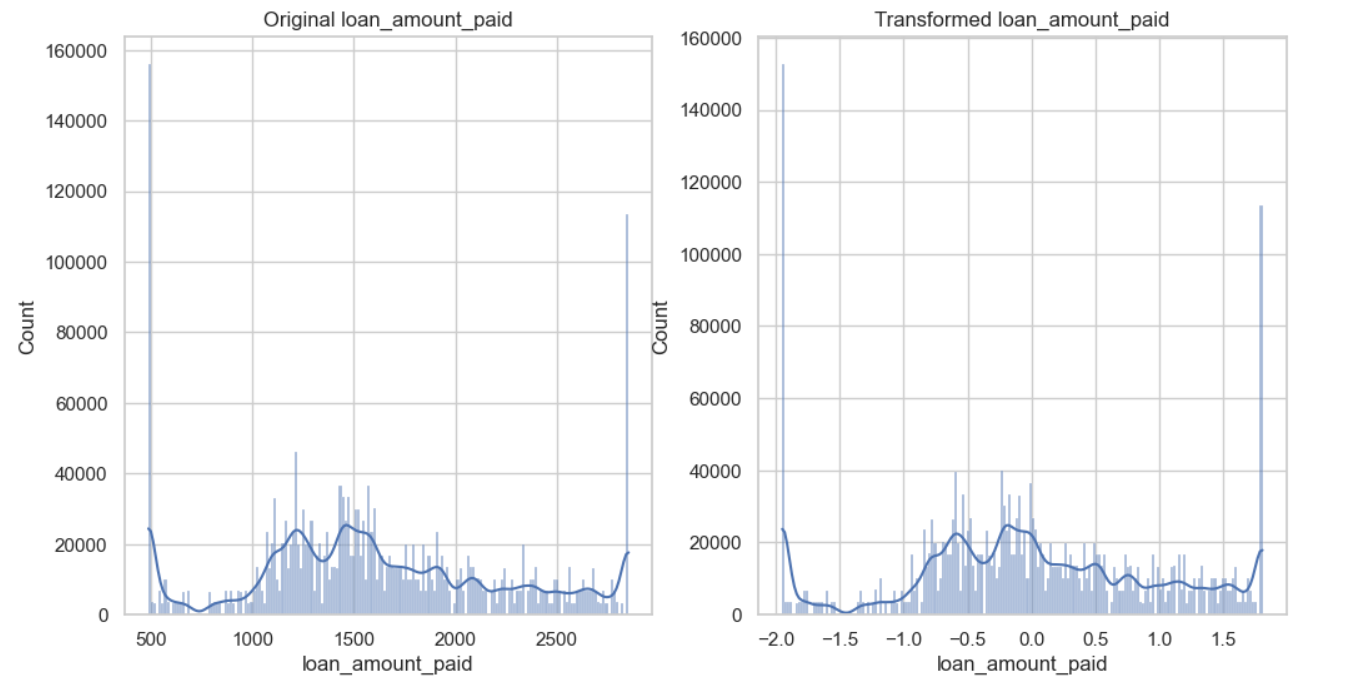


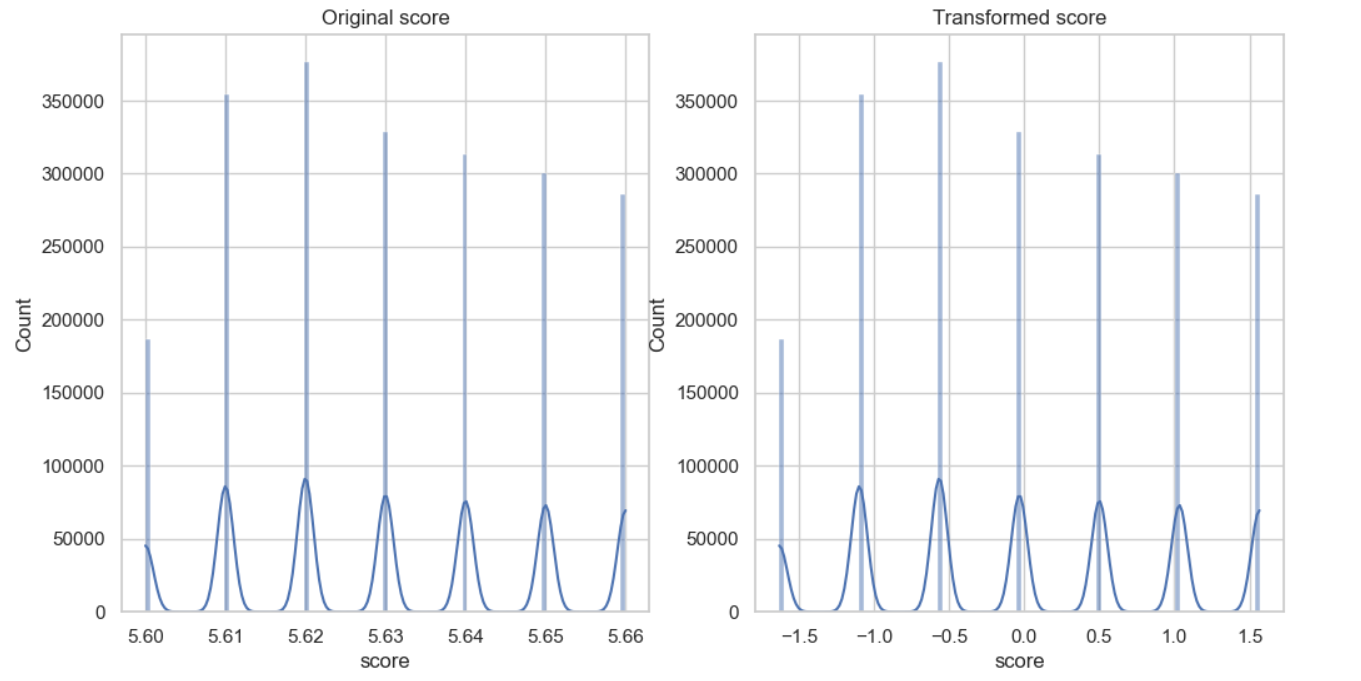


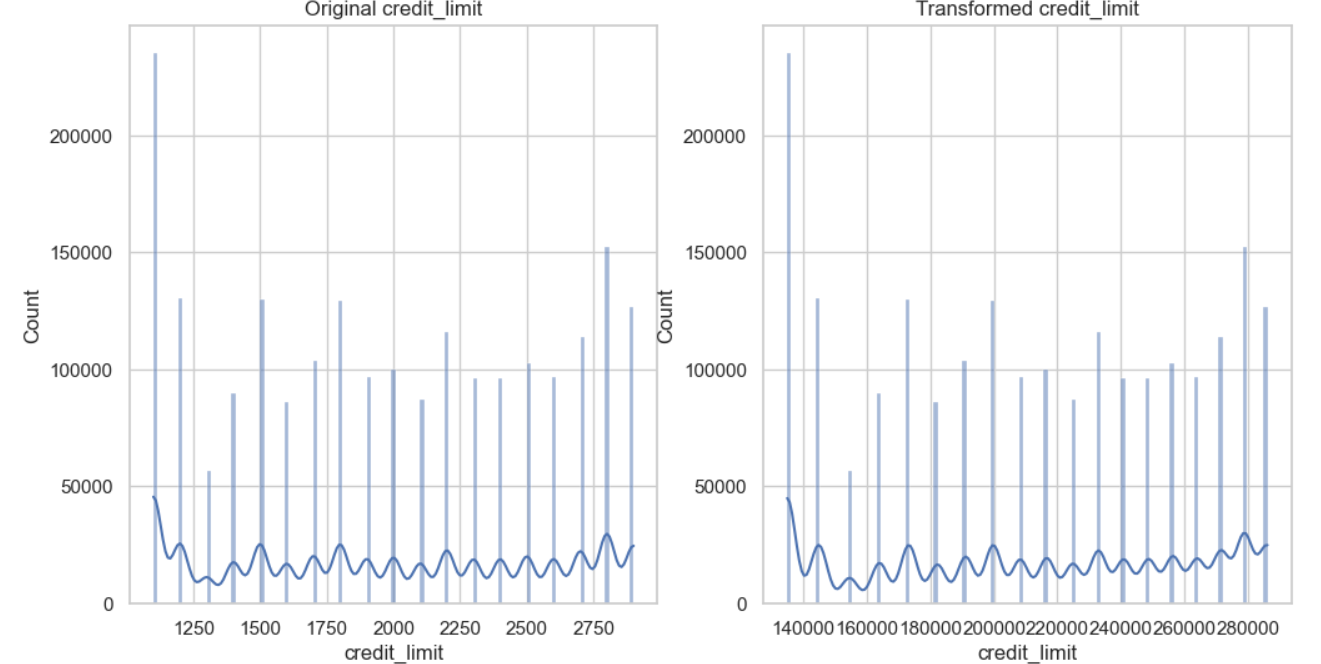




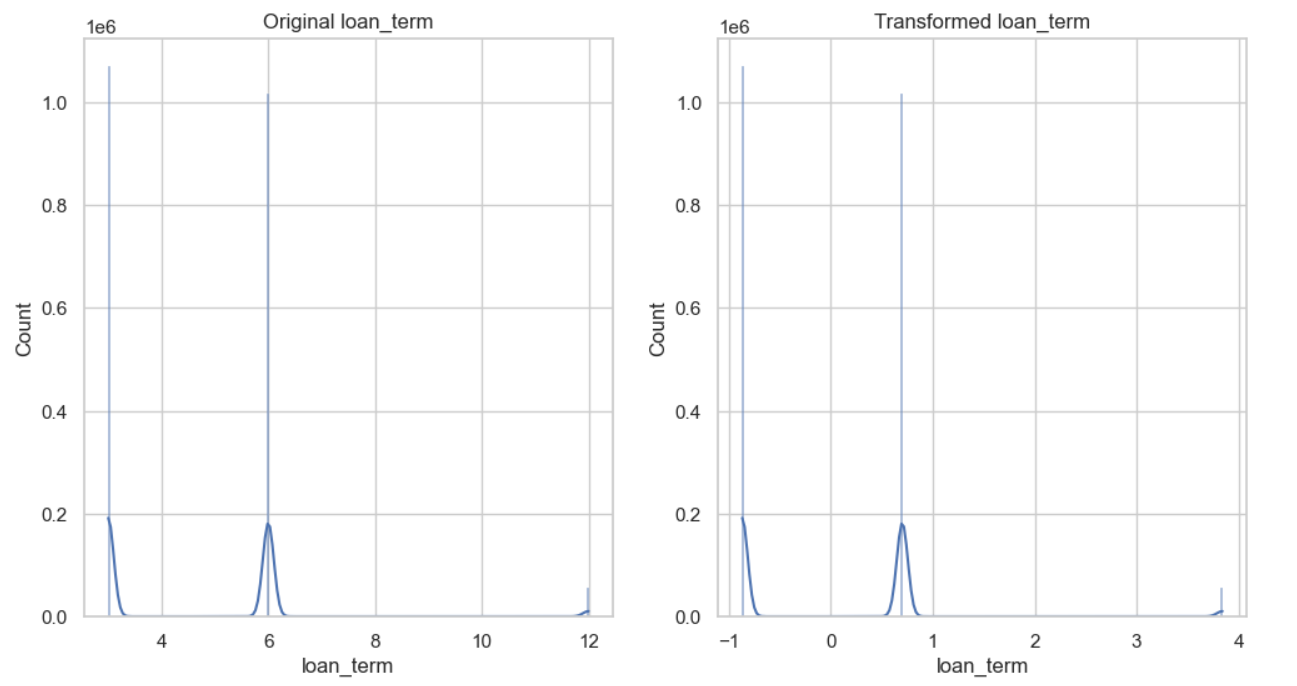


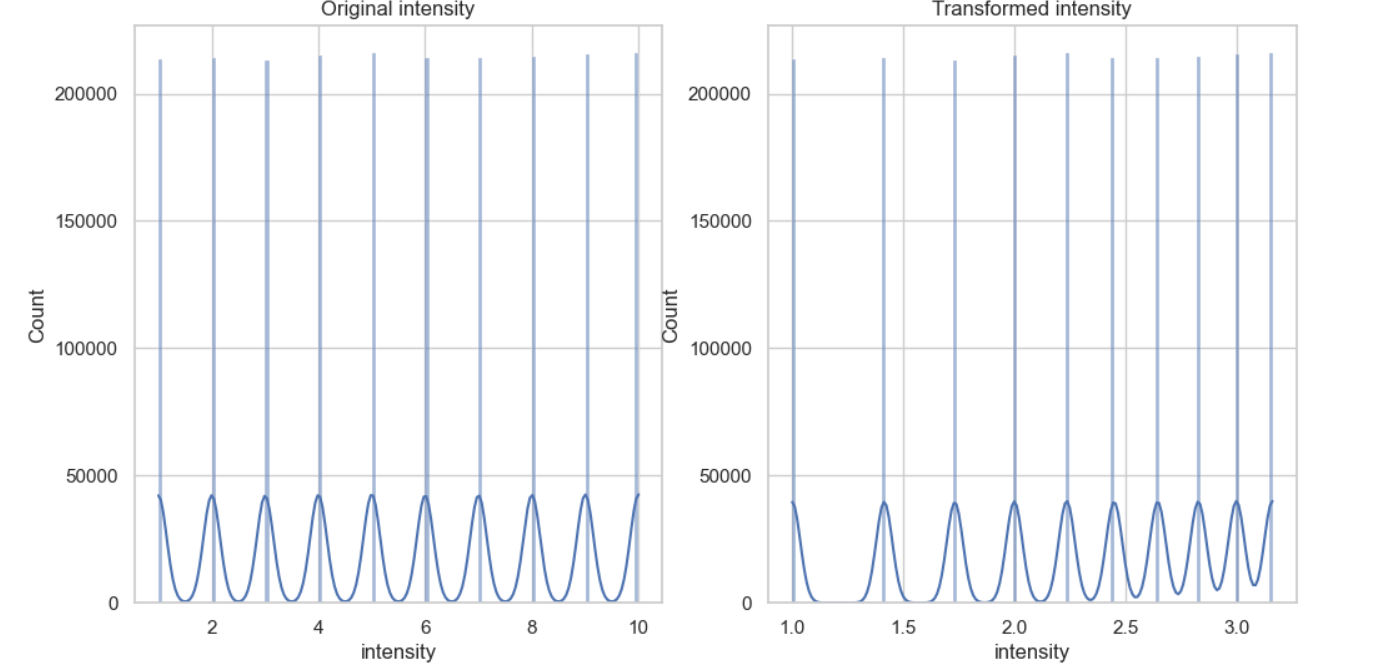


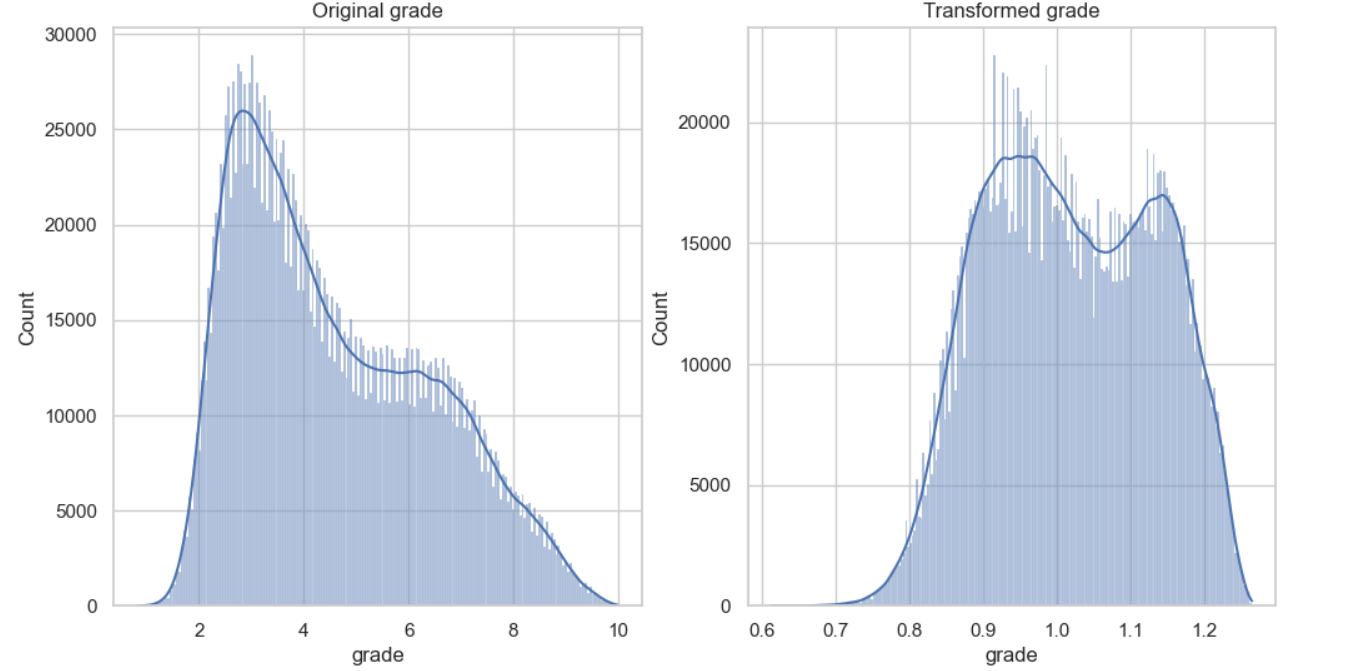












**Analysis:** Each transformation method is applied to columns that would benefit from it, followed by side-by-side visualizations for comparison. Here we can clearly see that all the columns were almost transformed to normal distribution except 4 columns (Interest Rate, Loan Term, Intensity and Score) as mentioned it is due to repeated numbers.

**Step 10: Emotional Pattern Analysis**

**Description:** This code effectively analyzes and visualizes emotional patterns over time and their correlation with various contexts. The use of grouping, aggregation, and visualization techniques provides a comprehensive overview of how emotions vary across different dimensions, aiding in understanding user sentiment in relation to loans and their emotional states.

**Implemented Code:**

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

data['timestamp'] = pd.to\_datetime(data['timestamp'])

dominant\_emotions = data.groupby(['user\_id', data['timestamp'].dt.date])['primary\_emotion'].agg(lambda x: x.value\_counts().index[0]).reset\_index()

dominant\_emotions.rename(columns={'timestamp': 'date'}, inplace=True) # Rename for clarity

emotion\_counts = dominant\_emotions['primary\_emotion'].value\_counts().reset\_index()

emotion\_counts.columns = ['primary\_emotion', 'count']

print("Dominant Emotions for Users Over Time:")

print(emotion\_counts)

# Visualize dominant emotions over time

plt.figure(figsize=(12, 6))

sns.barplot(data=emotion\_counts, x='primary\_emotion', y='count', palette='viridis')

plt.title('Dominant Emotions Count Over Time')

plt.xlabel('Primary Emotion')

plt.ylabel('Count')

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

# Grouping by user and time of day

emotional\_pattern = data.groupby(['user\_id', 'time\_of\_day', 'primary\_emotion'])['intensity'].mean().reset\_index()

print("\nNumerical Data for Emotional Patterns by Time of Day:")

print(emotional\_pattern)

plt.figure(figsize=(12, 6))

sns.lineplot(data=emotional\_pattern, x='time\_of\_day', y='intensity', hue='primary\_emotion', marker='o')

plt.title('Emotional Patterns by Time of Day')

plt.xlabel('Time of Day')

plt.ylabel('Average Emotional Intensity')

plt.legend(title='Primary Emotion')

plt.show()

context\_correlation = data.groupby(['primary\_emotion', 'relationship', 'situation', 'time\_of\_day']).size().reset\_index(name='counts')

print("\nEmotion Correlation with Contexts:")

print(context\_correlation)

# Visualize correlation of emotions with different contexts

plt.figure(figsize=(14, 7))

sns.heatmap(context\_correlation.pivot\_table(index='primary\_emotion', columns='relationship', values='counts', fill\_value=0), cmap='Blues')

plt.title('Emotion Correlation with Relationships')

plt.show()

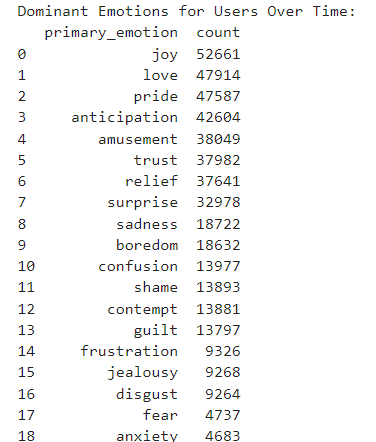
**Explanation of Code:**

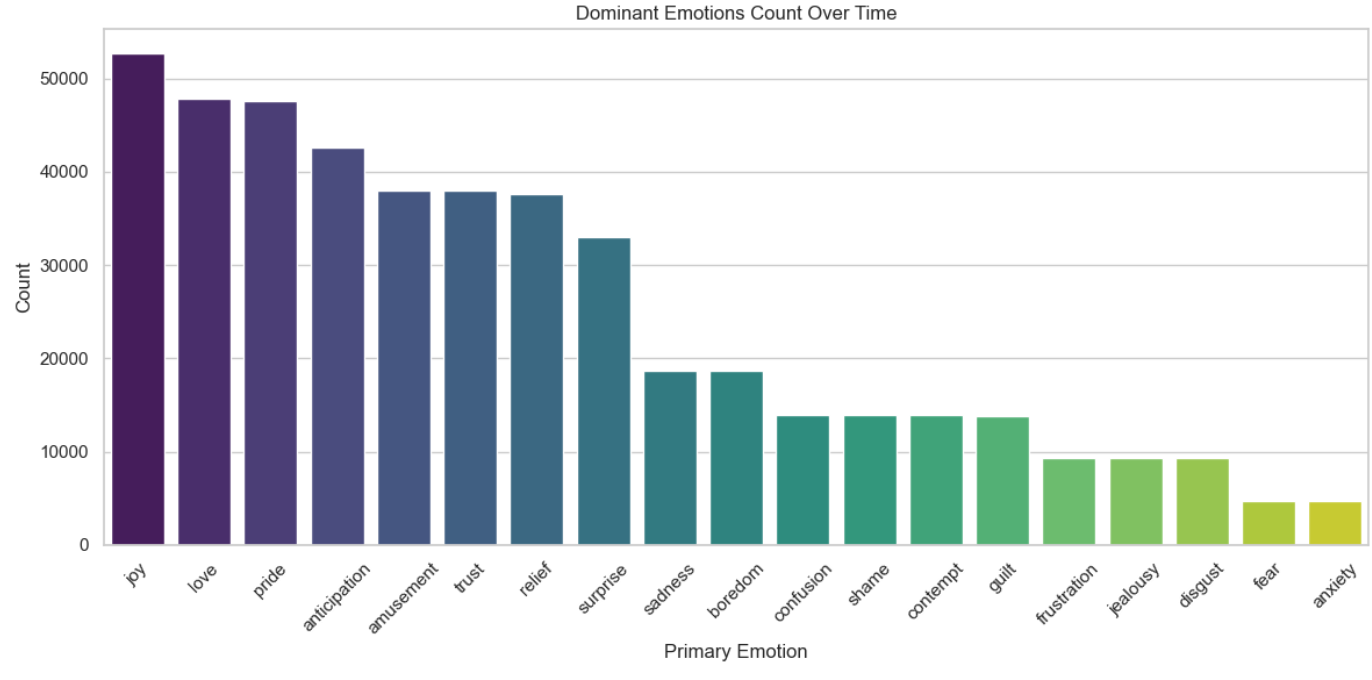
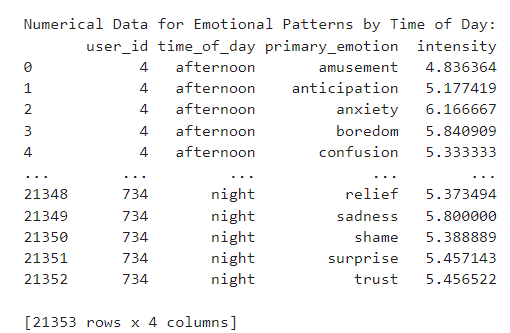
* **Timestamp:** timestamp column from a string format to a datetime object, enabling easier manipulation of date and time data.
* **Grouping by User and Date:** The data is grouped by user\_id and the date extracted from timestamp, then the most frequent primary\_emotion for each user on that date is calculated.
* **Aggregation:** The aggregation function value\_counts() counts the occurrences of each emotion, and index[0] selects the most common one.
* **emotion\_count:** counts how many times each dominant emotion appears across all users, creating a new DataFrame that stores the emotions and their respective counts.
* **Grouping:** The dataset is grouped by user\_id, time\_of\_day, and primary\_emotion, calculating the average intensity of emotions.This provides a new DataFrame that reflects how emotional intensity varies by time of day for each user and emotion.

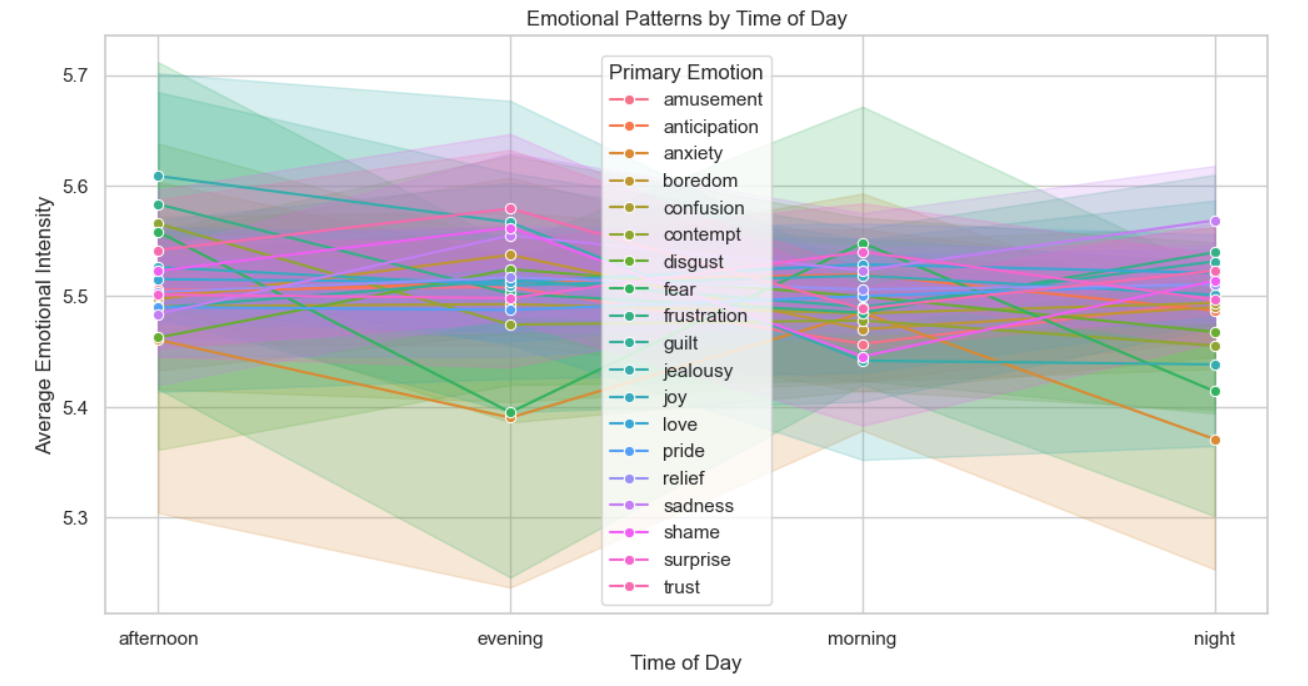
groups the data by primary\_emotion, relationship, situation, and time\_of\_day, counting occurrences in each group, which helps analyze how emotions correlate with these contexts.

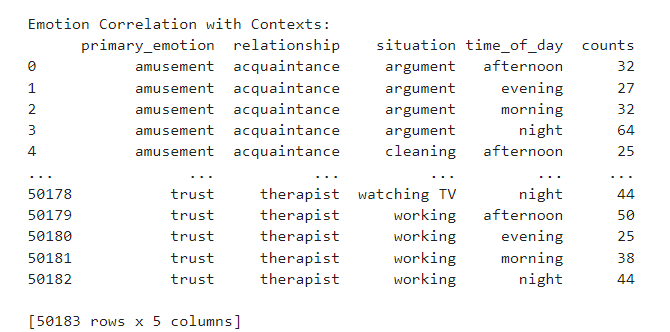
* **Heatmap Visualization:** A heatmap is generated to visualize the correlation between emotions and relationships. This provides a clear view of how different emotions are related to various relationships.
* **Pivot Table:** The data is reshaped into a pivot table to create the heatmap, with primary\_emotion as rows and relationship as columns.

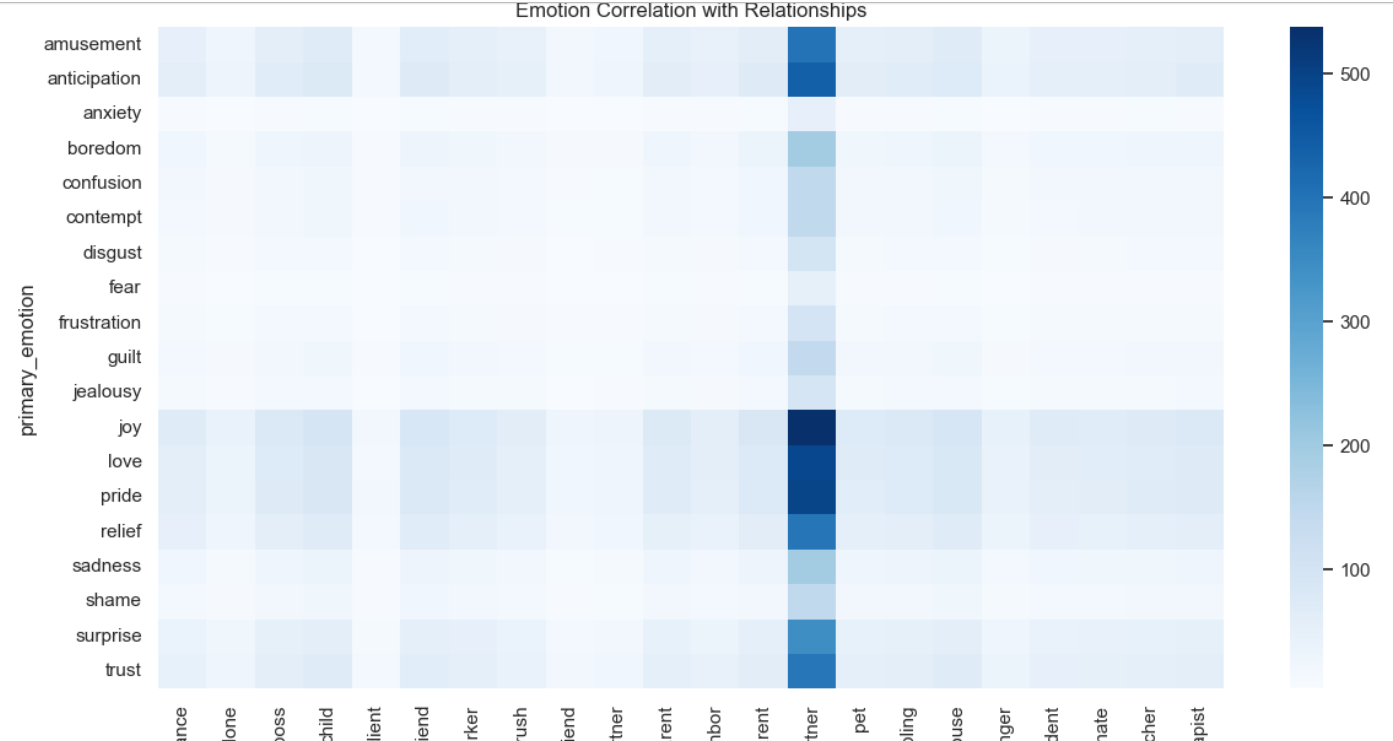
**Result: Executed (Output of the screenshot is attached below)**









**Analysis:**

* In dominant emotions counts over time graph we can see that Joy is the dominant emotion and anxiety is less dominant emotion over time.
* In emotion pattern by time of the day also we can conclude that joy is the most dominant emotion in morning, evening and night. Anxiety is less dominant emotion out of 4 time of the day i.e., (afternoon, evening, night).

**Step 11: Loan – Emotion Correlation**

**Description:** This section of code is designed to analyze the relationship between emotional states and loan characteristics within the dataset. Specifically, it focuses on how different primary emotions are associated with average loan amounts, interest rates, and emotional intensity. By visualizing these relationships, the code helps in understanding the impact of emotional factors on lending practices.

**Implementation Code:**

# Loan-Emotion Correlation

emotion\_loan\_correlation = data.groupby('primary\_emotion').agg(

avg\_loan\_amount=('loan\_amount', 'mean'),

avg\_interest\_rate=('interest\_rate', 'mean'),

avg\_intensity=('intensity', 'mean')

).reset\_index()

print("\nNumerical Data: Average Loan Characteristics by Primary Emotion:")

print(emotion\_loan\_correlation)

plt.figure(figsize=(12, 6))

sns.barplot(x='primary\_emotion', y='avg\_loan\_amount', data=emotion\_loan\_correlation, palette='coolwarm')

plt.title('Average Loan Amount by Primary Emotion')

plt.xlabel('Primary Emotion')

plt.ylabel('Average Loan Amount')

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

plt.figure(figsize=(12, 6))

sns.lmplot(data=data, x='loan\_amount', y='intensity', hue='primary\_emotion', height=6, aspect=1.5, scatter\_kws={'alpha':0.5}, markers='o')

plt.title('Emotional Intensity vs Loan Amount (with Regression Line)')

plt.xlabel('Loan Amount')

plt.ylabel('Emotional Intensity')

plt.legend(title='Primary Emotion', bbox\_to\_anchor=(1.05, 1), loc='upper left')

plt.tight\_layout()

plt.show()

plt.figure(figsize=(12, 6))

plt.hexbin(data['loan\_amount'], data['intensity'], gridsize=30, cmap='coolwarm', mincnt=1)

plt.colorbar(label='Count of Points')

plt.title('Hexbin: Emotional Intensity vs Loan Amount')

plt.xlabel('Loan Amount')

plt.ylabel('Emotional Intensity')

plt.tight\_layout()

plt.show()

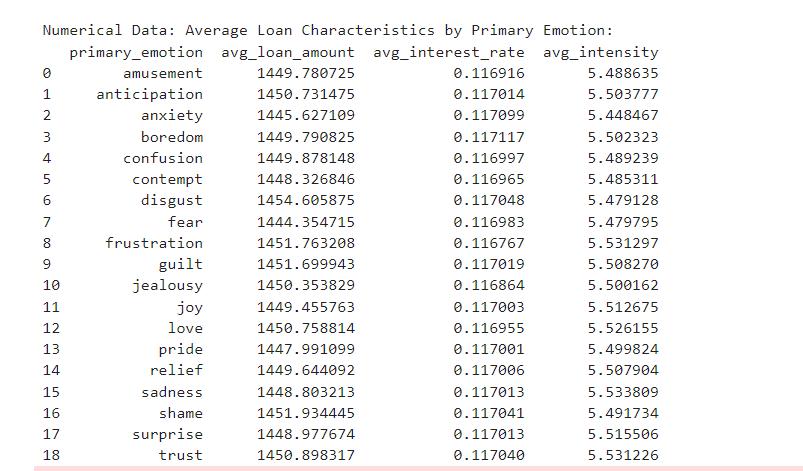
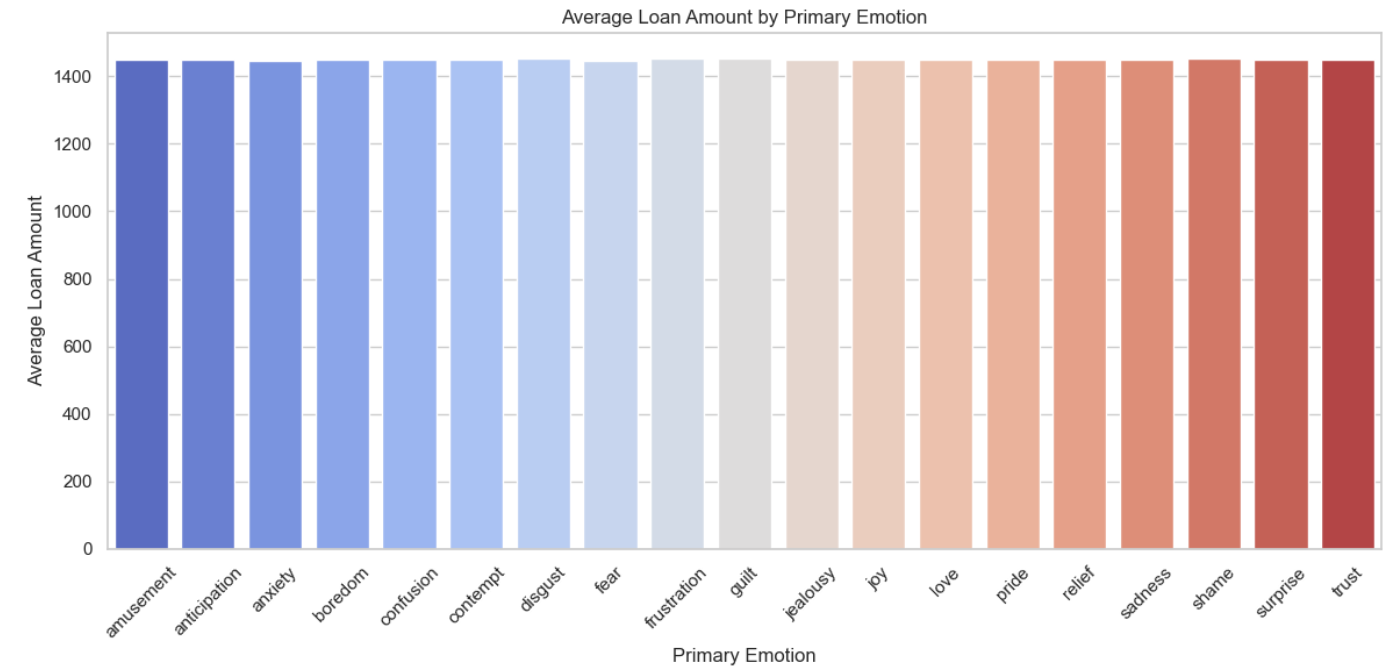
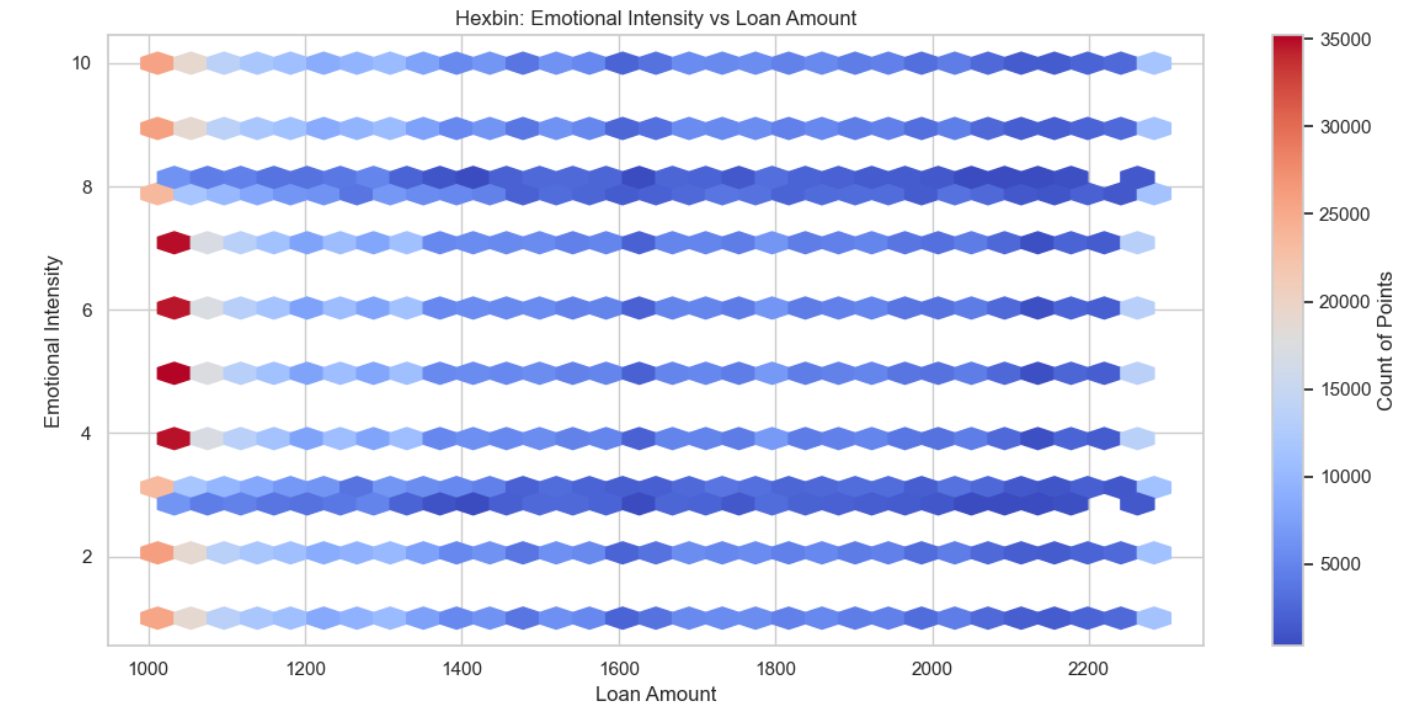
correlation = data[['intensity', 'loan\_amount', 'interest\_rate']].corr()

print("\nCorrelation between Emotional Intensity and Loan Terms (Loan Amount, Interest Rate):\n", correlation)

**Code Explanation:**

* **Grouping by Primary Emotion:**
* **Aggregation:** The line emotion\_loan\_correlation = data.groupby('primary\_emotion').agg(...) groups the data by the primary\_emotion column and calculates the average loan **characteristics for each emotion:**
* **Average Loan Amount:** Calculated using mean() on the loan\_amount column.
* **Average Interest Rate:** Calculated using mean() on the interest\_rate column.
* **Average Intensity:** Calculated using mean() on the intensity column.
* **Dataframe Reset:** .reset\_index() ensures that the results are formatted as a standard DataFrame for easier manipulation and visualization.
* **Displaying Numerical Data:**
* The line print("\nNumerical Data: Average Loan Characteristics by Primary Emotion:") is followed by printing the aggregated results stored in emotion\_loan\_correlation.
* This output provides a clear view of the average loan characteristics categorized by emotional states, which can be crucial for further analysis.
* Visualization of Average Loan Amount by Primary Emotion:
* **Bar Plot Creation:** The code sns.barplot(...) generates a bar plot to visualize the average loan amounts for each primary emotion.
* **X-axis:** Set to primary\_emotion, representing different emotional states.
* **Y-axis:** Set to avg\_loan\_amount, showing the average loan amounts associated with each emotion.
* **Color Palette:** The palette='coolwarm' parameter gives a visually appealing gradient to the bars, enhancing interpretability.
* **Plot Customization:** Additional settings such as titles, labels, and layout adjustments (plt.tight\_layout()) are applied to improve the presentation of the plot.
* Emotional Intensity vs Loan Amount:
* **Scatter Plot with Regression Line:** The sns.lmplot(...) function creates a scatter plot displaying the relationship between loan amounts and emotional intensity.
* **X-axis:** Represents the loan\_amount.
* **Y-axis:** Represents the intensity.
* **Hue:** The primary\_emotion differentiates points based on their associated emotions, allowing for color-coded insights into the distribution of data.
* **Regression Line:** The inclusion of a regression line (fit) provides a visual cue about the correlation between the two variables.
* **Visual Customization:** Titles and labels are added, along with a legend for easy interpretation.
* **Hexbin Plot of Emotional Intensity vs Loan Amount:**
* **Hexbin Visualization:** The plt.hexbin(...) function generates a hexbin plot, which is useful for visualizing the density of data points in relation to loan amounts and emotional intensity.
* **X-axis:** Represents the loan\_amount.
* **Y-axis:** Represents the intensity.
* **Color Legend:** The color of the hexagons indicates the count of points within each hexbin, providing insight into where data points are concentrated.
* **Colorbar Addition:** A colorbar is added to the plot, indicating the count of points, which aids in understanding the density of observations in specific ranges.
* **Correlation Calculation:** The code correlation = data[['intensity', 'loan\_amount', 'interest\_rate']].corr() computes the correlation matrix for the intensity, loan\_amount, and interest\_rate columns.This matrix quantifies the strength and direction of the linear relationships between these variables, helping to identify which factors are positively or negatively correlated.Finally, the correlation results are printed to provide insight into how emotional intensity relates to loan amounts and interest rates.
* **Insights into Loan Characteristics:** By analyzing the average loan amounts alongside emotional states, stakeholders can gain insights into how emotional factors may influence lending decisions.
* **Relationship Understanding:** The scatter plot with regression and hexbin plots allow for a deeper understanding of the correlations and interactions between emotional intensity and loan amounts.
* **Quantitative Analysis:** The correlation matrix provides a numerical assessment of relationships, allowing for data-driven decision-making and strategy formulation.

**Result: Executed (Output of the screenshot is attached below)**

**Analysis: The above graph shows us the importance of emotions to pay the loan and interest amount.**

**Step 12: Lending Operation Assessment**

**Description:** The code is designed to assess the lending operations by calculating key performance indicators such as the default rate, growth in lending over the years, profitability, and trends in defaulted loans. This information can help in making informed decisions about future lending strategies.

**Implemented Code:**

default\_rate = (data['status'] == 'unpaid').mean() \* 100

print(f"Default Rate: {default\_rate:.2f}%")

data['issue\_year'] = pd.to\_datetime(data['issue\_date']).dt.year

annual\_performance = data.groupby('issue\_year').agg(

total\_loans=('loan\_amount', 'count'),

total\_disbursed=('loan\_amount', 'sum'),

total\_revenue=('total\_amount', 'sum'),

avg\_interest\_rate=('interest\_rate', 'mean'),

defaulted\_loans=('status', lambda x: (x == 'unpaid').sum())

).reset\_index()

annual\_performance['profitability'] = annual\_performance['total\_revenue'] - annual\_performance['total\_disbursed']

print("\nAnnual Performance Summary:")

print(annual\_performance)

plt.figure(figsize=(14, 6))

sns.lineplot(x='issue\_year', y='total\_disbursed', data=annual\_performance, marker='o', label='Total Disbursed')

sns.lineplot(x='issue\_year', y='profitability', data=annual\_performance, marker='o', color='orange', label='Profitability')

plt.title('Total Loans Disbursed and Profitability Over Time')

plt.xlabel('Year')

plt.ylabel('Amount')

plt.legend()

plt.tight\_layout()

plt.show()

plt.figure(figsize=(14, 6))

sns.barplot(x='issue\_year', y='defaulted\_loans', data=annual\_performance, color='salmon')

plt.title('Number of Defaulted Loans Over Time')

plt.xlabel('Year')

plt.ylabel('Number of Defaulted Loans')

plt.tight\_layout()

plt.show()

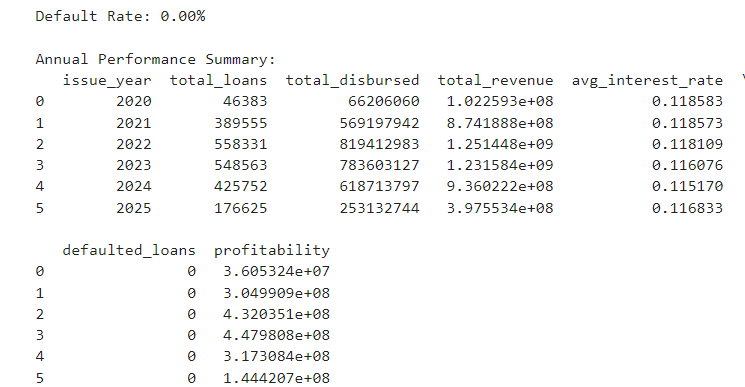
print("\nDetailed Annual Performance with Defaulted Loans:")

print(annual\_performance[['issue\_year', 'defaulted\_loans']])

**Explanation of Code:**

* **Default Rate Calculation**: The code calculates the percentage of loans that are in an "unpaid" status. This is done by checking how many loan statuses are marked as "unpaid" and averaging them. This is critical for understanding the risk profile of the lending portfolio and helps in risk management.
* **Annual Performance Calculation:**The code first extracts the year from the issue\_date to group the data accordingly.
* **It aggregates the data by year to calculate:**
* **Total Loans:** Count of loans issued in that year.
* Total Disbursed: Total amount of money loaned out.
* **Total Revenue:** Total amount collected, including principal and interest.
* **Average Interest Rate:** Average interest charged on loans for that year.
* **Defaulted Loans:** The number of loans that have gone unpaid.
* **Profitability Calculation:** Profitability is calculated by subtracting total disbursed amounts from total revenue. This helps in understanding how much profit is generated from the loans issued.
* **Visualization of Trends:** This section generates a line plot showing the trend of total loans disbursed and profitability over time. The use of different colors helps differentiate between the two metrics, making it easy to see how they evolve in relation to each other.
* **Bar Chart for Defaulted Loans:** A bar chart is created to visualize the number of defaulted loans per year. This visualization is important for assessing the risk trends in lending operations.

**Result: Executed (Output of the screenshot is attached below)**





**Analysis:**

* **Default Rate:** The default rate provides insight into how many loans are failing, which can help in adjusting lending strategies or interest rates.
* Annual Performance: By observing total loans disbursed alongside profitability, organizations can make better lending decisions to ensure sustainable growth.
* **Default Trends:** The trends in defaulted loans can indicate economic conditions or specific issues with borrowers in certain years.
* In the total disbursed and profitability over time graph we can see that in the year of 2022 total.
* Disbursed rate is very low and very high is in the year 2022. And profitability was very high in the year 2023 and very low in 2020.

**Step 13: Loan Disbursement**

**Description:** The purpose of this code is to analyze and visualize the trends in loan disbursements over time. By tracking the total amount of loans disbursed each month, it provides insights into lending patterns, seasonality, and the overall performance of the lending operations.

**Implemented Code:**

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

data['issue\_date'] = pd.to\_datetime(data['issue\_date'], errors='coerce')

data['total\_amount'] = pd.to\_numeric(data['total\_amount'], errors='coerce')

loan\_disbursement\_trends = data.groupby(data['issue\_date'].dt.to\_period('M'))['total\_amount'].sum().reset\_index()

loan\_disbursement\_trends['issue\_date'] = loan\_disbursement\_trends['issue\_date'].dt.to\_timestamp()

loan\_disbursement\_trends = loan\_disbursement\_trends.dropna()

plt.figure(figsize=(10, 6))

sns.lineplot(x='issue\_date', y='total\_amount', data=loan\_disbursement\_trends, marker='o', color='blue')

plt.title('Loan Disbursement Trends Over Time')

plt.xlabel('Issue Date (Month)')

plt.ylabel('Total Disbursed Amount')

plt.xticks(rotation=45)

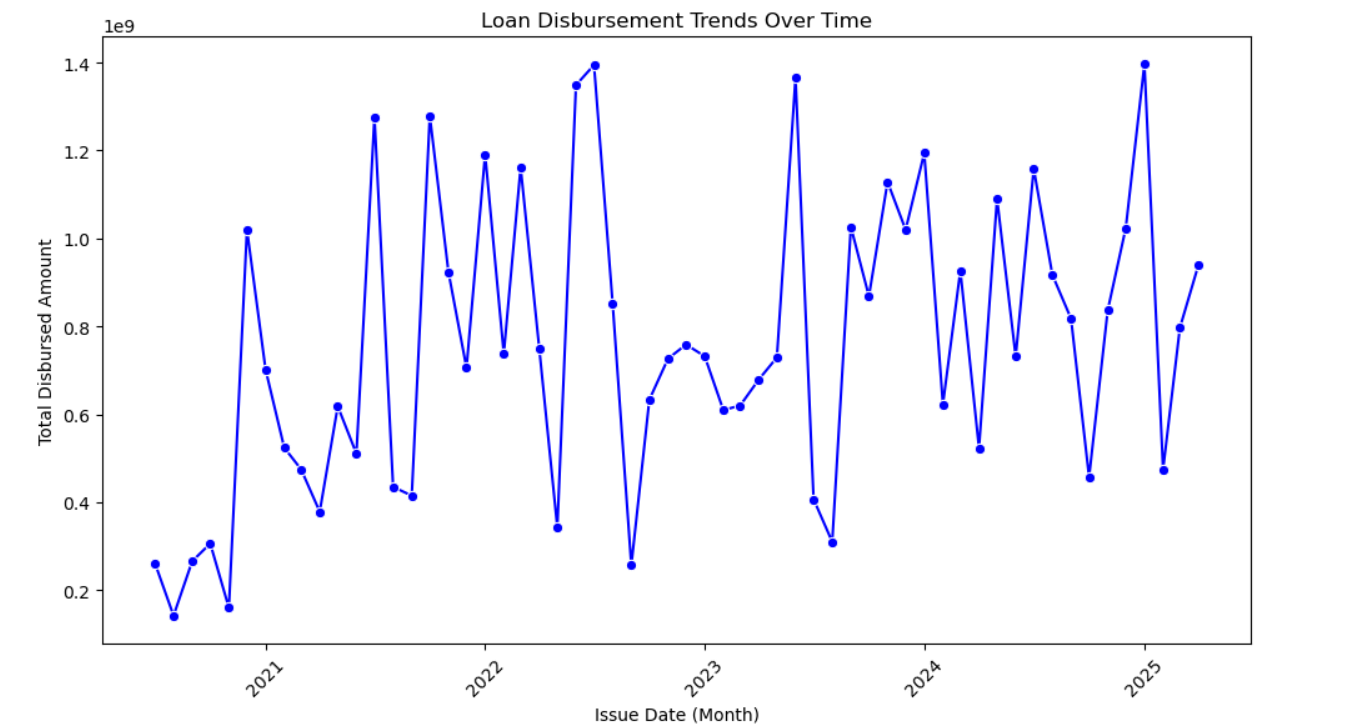
plt.tight\_layout()

plt.show()

**Explanation of Code:**

* **Convert to Datetime:** This line converts the issue\_date column into a datetime format, ensuring that any date operations performed later on will work correctly. The errors='coerce' argument will convert any non-convertible values to NaT (Not a Time), effectively handling any invalid date entries.
* **Convert to Numeric:** Similar to the previous step, this line converts the total\_amount column to a numeric type, which is necessary for performing numerical operations and aggregations.
* **Grouping by Month:** The code groups the data by the month of the issue\_date and calculates the total amount disbursed (total\_amount) for each month.
* **Resetting Index:** The reset\_index() method is used to convert the groupby object back to a DataFrame.
* **Convert Period to Timestamp:** This step converts the monthly period back to a timestamp format suitable for plotting on the x-axis.
* **Drop Missing Values:** The code checks for any missing values in the loan\_disbursement\_trends DataFrame and removes them to ensure a clean dataset for visualization.
* **Creating the Plot:** This section creates a line plot using Seaborn's lineplot function, where:
* X-axis: Represents the issue\_date (grouped by month).
* Y-axis: Represents the total\_amount (total disbursed amount).
* **Markers:** Circles are added to the data points for better visibility.
* **Titles and Labels:** The code sets the title and labels for the axes to make the plot more informative.
* **Rotating X-ticks:** The x-ticks are rotated for better readability, especially when there are many data points.
* **Displaying the Plot:** Finally, the plot is displayed using plt.show()

**Result: Executed (Output screenshot is attached below)**



**Analysis:**

* Trends Over Time: The line plot will allow to see how loan disbursements change over the months.
* **Volume of Loans**: Understanding the total amounts disbursed helps assess the lending strategy's effectiveness and the market demand for loans.
* In the year 2022 (June month) loan disbursed was very high and in the year 2020(September month) loan disbursed was very less.

**Step 12: Machine Learning model to predict loan terms based on emotional and contextual data & identifying important features**

**Description:** The main goal of this code is to create a predictive model using the Random Forest algorithm to estimate the loan term based on several features such as credit score, credit limit, interest rate, loan amount, and emotional intensity. The model also evaluates its performance and determines the importance of each feature in predicting the loan term.

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.inspection import permutation\_importance

import seaborn as sns

import matplotlib.pyplot as plt

X = data[['score', 'credit\_limit', 'interest\_rate', 'loan\_amount', 'intensity']] # Include relevant features

y = data['loan\_term'] # Target variable

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

model = RandomForestRegressor(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f'Mean Squared Error: {mse:.2f}')

print(f'R-squared: {r2:.2f}')

result = permutation\_importance(model, X\_test, y\_test, n\_repeats=30, random\_state=42)

feature\_importance = pd.DataFrame({

'Feature': X.columns,

'Importance': result.importances\_mean

}).sort\_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))

sns.barplot(x='Importance', y='Feature', data=feature\_importance)

plt.title('Feature Importance (Permutation Importance)')

plt.xlabel('Importance Score')

plt.ylabel('Feature')

plt.show()

def predict\_loan\_term(input\_data):

"""

Predicts the loan term based on the input features.

input\_data (dict): Dictionary containing input features.

Returns:

float: Predicted loan term.

"""

input\_df = pd.DataFrame([input\_data])

predicted\_term = model.predict(input\_df)

return predicted\_term[0]

input\_example = {

'score': 700,

'credit\_limit': 15000,

'interest\_rate': 5.5,

'loan\_amount': 10000,

'intensity': 0.8

}

predicted\_term = predict\_loan\_term(input\_example)

print(f'Predicted Loan Term: {predicted\_term:.2f}')

**Explanation of Code:**

* **Feature Selection:** Here, X is defined as the DataFrame containing the predictor variables, while y is the target variable (loan term) you want to predict. This selection is crucial as it directly impacts the model's performance.
* **Train-Test Split:** The dataset is split into training and testing sets using an 80-20 ratio. This allows for model training on one subset and evaluation on another, which helps prevent overfitting.
* **Random Forest Regressor:** A Random Forest model is instantiated with 100 trees (estimators) for robust predictions. The model is then trained on the training data.
* **Predicting Loan Terms:** The model predicts loan terms using the test set features.
* **Performance Metrics:** The Mean Squared Error (MSE) and R-squared (R²) score are calculated to evaluate the model's accuracy.
* **MSE:** Indicates the average squared difference between predicted and actual values (lower is better).
* **R²:** Represents the proportion of variance in the dependent variable predictable from the independent variables (values close to 1 indicate better fit).
* **Permutation Importance:** This method measures the effect of shuffling feature values on the model's accuracy. The larger the drop in accuracy, the more important the feature is.
* **Visualization:** The importance scores are plotted using a bar chart, showing how each feature influences the prediction of loan terms.
* **Custom Prediction Function:** This function takes input features as a dictionary, converts it to a DataFrame, and uses the trained model to predict the loan term.
* **Example Usage:** A sample input is provided to demonstrate how to use the function, showcasing the flexibility to predict loan terms for new data.

**Result: Executed (Output screenshot is attached below)**



**Analysis:**

* The Mean Squared Error and R-squared values provide insights into the model's performance, indicating how well the features correlate with the loan term.
* The feature importance plot helps identify which factors have the most influence on predicting the loan term, guiding future lending strategies and assessments.
* The important feature for loan prediction is the score as we get the output using the Random Forest technique.
* Loan Term is of 3 types 3, 6 and 12 years out of which the prediction score of loan term is average 4.54 years.