


```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
```


```
1 data = pd.read_csv("/content/Consumer transactions.csv")
2 data
```



	account	age	amount	balance	card_present_flag	customer_id	date	first_name	gender	latitude	...	txn_description	bin_
0	ACC-1598451071	26	16.25	35.39	1.0	CUS-2487424745	2018-08-01	Diana	F	-27.95	...	POS	20
1	ACC-1598451071	26	14.19	21.20	0.0	CUS-2487424745	2018-08-01	Diana	F	-27.95	...	SALES-POS	20
2	ACC-1598451071	26	3.25	17.95	1.0	CUS-2487424745	2018-08-01	Diana	F	-27.95	...	SALES-POS	20
3	ACC-1598451071	26	14.10	3.85	1.0	CUS-2487424745	2018-08-01	Diana	F	-27.95	...	POS	20
4	ACC-1598451071	26	10.67	1006.85	1.0	CUS-2487424745	2018-08-01	Diana	F	-27.95	...	POS	20
...
12038	ACC-2153562714	24	3712.56	9707.77	NaN	CUS-423725039	2018-10-24	Linda	F	-31.88	...	PAY/SALARY	20
12039	ACC-1217063613	27	4863.62	4863.86	NaN	CUS-1739931018	2018-09-26	Kimberly	F	-37.82	...	PAY/SALARY	20
12040	ACC-1217063613	27	4863.62	8905.77	NaN	CUS-1739931018	2018-10-26	Kimberly	F	-37.82	...	PAY/SALARY	20
12041	ACC-3100725361	25	6107.23	6111.57	NaN	CUS-2178051368	2018-09-26	Ronald	M	-17.03	...	PAY/SALARY	20
12042	ACC-3100725361	25	6107.23	10753.02	NaN	CUS-2178051368	2018-10-26	Ronald	M	-17.03	...	PAY/SALARY	20

12043 rows x 30 columns

```
1 # Display basic information about the dataset
2 data_info = data.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12043 entries, 0 to 12042
Data columns (total 30 columns):
#   Column              Non-Null Count  Dtype
---  -
0   account              12043 non-null  object
1   age                  12043 non-null  int64
2   amount              12043 non-null  float64
3   balance              12043 non-null  float64
4   card_present_flag    7717 non-null   float64
5   customer_id          12043 non-null  object
6   date                 12043 non-null  object
7   first_name           12043 non-null  object
8   gender               12043 non-null  object
9   latitude              12043 non-null  float64
10  longitude             12043 non-null  float64
11  merchant_code         883 non-null    float64
12  merchant_id           7717 non-null   object
13  merchant_latitude     7717 non-null   float64
14  merchant_longitude    7717 non-null   float64
15  merchant_state        7717 non-null   object
16  merchant_suburb       7717 non-null   object
17  movement              12043 non-null  object
18  status                12043 non-null  object
19  transaction_id        12043 non-null  object
20  txn_description       12043 non-null  object
21  bin_age               12043 non-null  object
22  year                  12043 non-null  int64
23  month                 12043 non-null  int64
24  day                   12043 non-null  int64
25  hour                  12043 non-null  int64
26  minute                12043 non-null  int64
27  dow                   12043 non-null  object
28  payment_period        12043 non-null  object
29  annual_salary         12043 non-null  float64
```

```
dtypes: float64(9), int64(6), object(15)
memory usage: 2.8+ MB
```

```
1 # Get and display descriptive statistics for the dataset
2 descriptive_stats = data.describe()
3 print("\nDescriptive Statistics:")
4 print(descriptive_stats)
```



Descriptive Statistics:

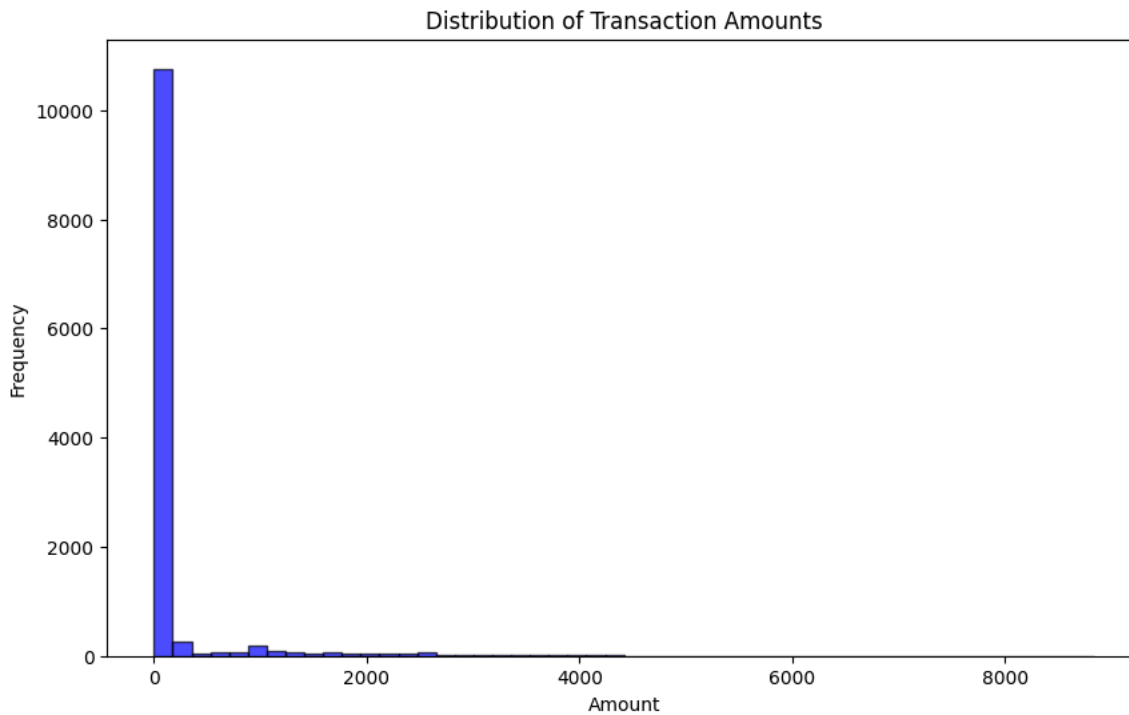
	age	amount	balance	card_present_flag \
count	12043.000000	12043.000000	12043.000000	7717.000000
mean	30.582330	187.933588	14704.195553	0.802644
std	10.046343	592.599934	31503.722652	0.398029
min	18.000000	0.100000	0.240000	0.000000
25%	22.000000	16.000000	3158.585000	1.000000
50%	28.000000	29.000000	6432.010000	1.000000
75%	38.000000	53.655000	12465.945000	1.000000
max	78.000000	8835.980000	267128.520000	1.000000

	latitude	longitude	merchant_code	merchant_latitude \
count	12043.000000	12043.000000	883.0	7717.000000
mean	-38.164347	143.648563	0.0	-32.752651
std	54.622791	16.669352	0.0	5.282423
min	-573.000000	114.620000	0.0	-43.310000
25%	-37.700000	138.690000	0.0	-37.710000
50%	-33.890000	145.230000	0.0	-33.840000
75%	-30.750000	151.220000	0.0	-29.440000
max	-12.370000	255.000000	0.0	-12.330000

	merchant_longitude	year	month	day	hour \
count	7717.000000	12043.0	12043.000000	12043.000000	12043.000000
mean	143.433277	2018.0	9.011957	15.862908	13.268621
std	12.090074	0.0	0.816511	8.899598	5.777284
min	113.830000	2018.0	8.000000	1.000000	0.000000
25%	144.680000	2018.0	8.000000	8.000000	9.000000
50%	145.830000	2018.0	9.000000	16.000000	13.000000
75%	151.210000	2018.0	10.000000	24.000000	18.000000
max	153.610000	2018.0	10.000000	31.000000	23.000000

	minute	annual_salary
count	12043.000000	12043.000000
mean	19.009632	68652.099506
std	19.879112	24300.871846
min	0.000000	29874.641667
25%	0.000000	51650.107000
50%	13.000000	60493.536000
75%	36.000000	81700.970000
max	59.000000	134946.236000

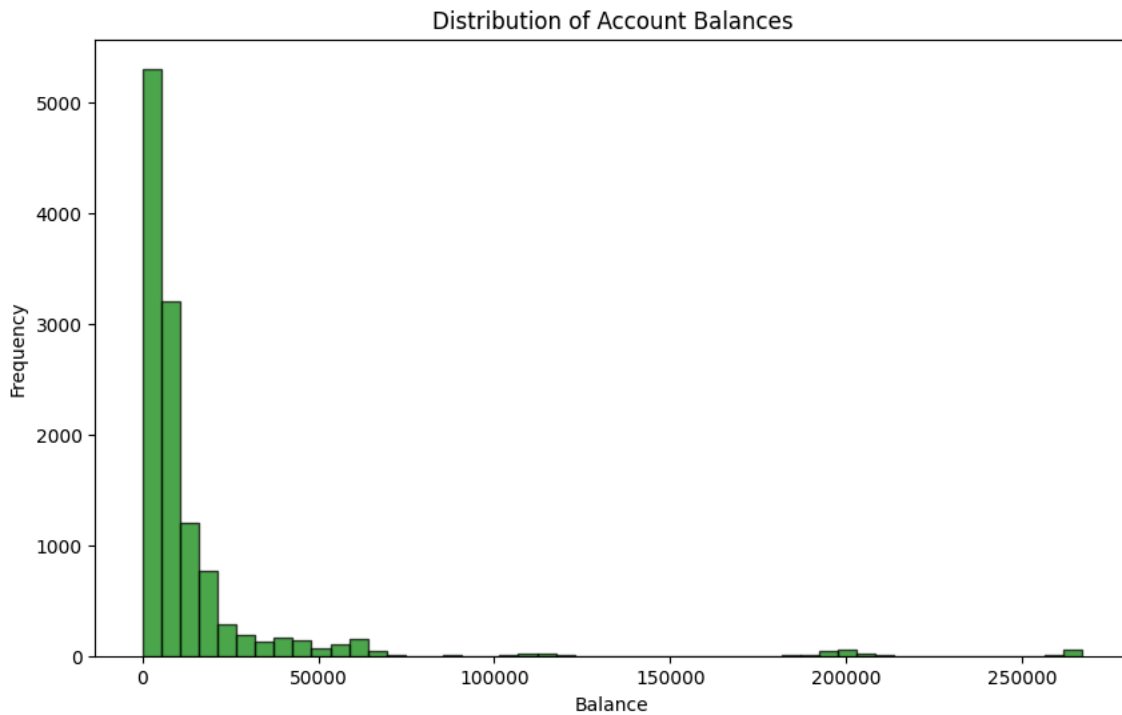
```
1 # Visualize the distributions of key variables like 'amount', 'balance', 'age'
2 plt.figure(figsize=(10, 6))
3 plt.hist(data['amount'], bins=50, alpha=0.7, color='blue', edgecolor='black')
4 plt.title('Distribution of Transaction Amounts')
5 plt.xlabel('Amount')
6 plt.ylabel('Frequency')
7 plt.show()
```



This graph displays the **distribution of transaction amounts** in the dataset. Here's a detailed breakdown:

1. **X-Axis (Amount):** The horizontal axis represents the transaction amounts, ranging from low to high values. It shows a wide range of amounts, going from very small transactions up to larger values, which span up to around 9000 or more.
2. **Y-Axis (Frequency):** The vertical axis represents the frequency of transactions at each amount level, indicating how many transactions fall within each amount range.
3. **Skewness and Concentration:**
 - The data is highly **right-skewed**. Most transactions occur at very low amounts, as evidenced by the high frequency at the lower end of the transaction amount scale.
 - There are relatively few transactions at higher amounts, and as the transaction amount increases, the frequency sharply decreases.
4. **Interpretation of Data Characteristics:**
 - This distribution suggests that the majority of transactions are for smaller amounts, while high-value transactions are rare.
 - Such a skewed distribution could be typical in many consumer transaction datasets, where everyday purchases are common, but large transactions are infrequent.
5. **Potential Implications:**
 - This pattern may indicate consumer spending behavior, where small, frequent purchases dominate. It might be relevant for understanding customer segments that make smaller, regular transactions versus those who engage in occasional, high-value transactions.

```
1 plt.figure(figsize=(10, 6))
2 plt.hist(data['balance'], bins=50, alpha=0.7, color='green', edgecolor='black')
3 plt.title('Distribution of Account Balances')
4 plt.xlabel('Balance')
5 plt.ylabel('Frequency')
6 plt.show()
```

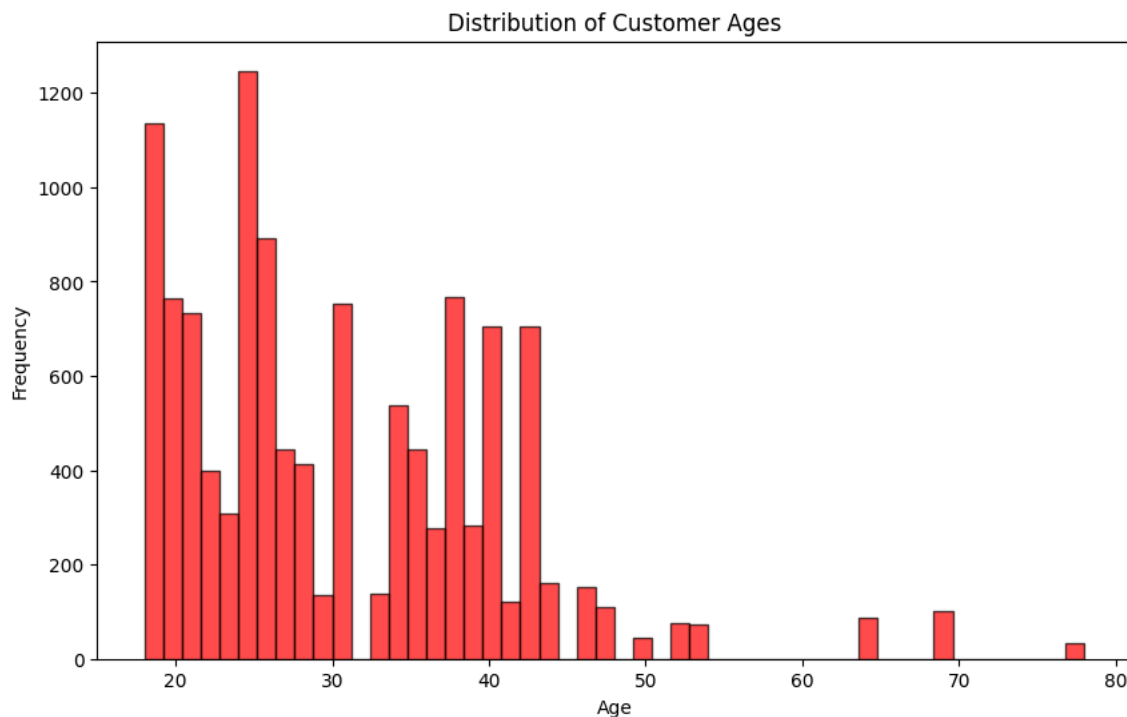


This graph illustrates the **Distribution of Account Balances** within the dataset, providing insights into the range and frequency of account balances among individuals or entities:

1. **X-Axis (Balance):** The horizontal axis represents the account balances, extending from zero to above 250,000. Each bin (or bar) along this axis represents a range of balance amounts, grouped into intervals (set to 50 bins here).
2. **Y-Axis (Frequency):** The vertical axis indicates the number of accounts that fall within each balance range. It shows how often certain balance amounts occur in the dataset.
3. **Data Skewness and Balance Distribution:**
 - Similar to the transaction amount distribution, this data is **right-skewed**. The majority of accounts have balances concentrated at the lower end of the scale.
 - A high number of accounts have small balances, as shown by the tall bars on the left side of the chart.
 - There are few accounts with significantly high balances, creating a long tail on the right, which represents accounts with balances well over 50,000 or even 200,000.
4. **Insight into Financial Behavior:**
 - This distribution may reflect that most customers maintain relatively low balances in their accounts, while a smaller segment of accounts hold very high balances.
 - Such a skewed balance distribution could indicate income disparities or spending/saving patterns in the population represented in the dataset.
5. **Potential Analytical Directions:**
 - In the thesis, we could analyze the characteristics of account holders in the high-balance segment compared to those with lower balances. This could reveal demographic, geographic, or behavioral factors that correlate with higher balances.
 - We could also explore why a majority of accounts have low balances, discussing possible economic, social, or policy factors influencing these patterns.
6. **Significance of the Findings:**
 - Understanding the distribution of account balances can be useful for segmenting customers and tailoring financial products or services. For instance, customers with high balances might benefit from different financial products compared to those based on balance ranges.

```
1 plt.figure(figsize=(10, 6))
2 plt.hist(data['age'], bins=50, alpha=0.7, color='red', edgecolor='black')
3 plt.title('Distribution of Customer Ages')
4 plt.xlabel('Age')
```

```
5 plt.ylabel('Frequency')
6 plt.show()
```



This graph shows the **Distribution of Customer Ages** in the dataset:

1. **X-Axis (Age):** The horizontal axis represents the ages of the customers, ranging from around 18 up to approximately 80 years old. The ages are grouped into intervals, with each bar representing a specific age range.
2. **Y-Axis (Frequency):** The vertical axis indicates the number of customers within each age range, showing how frequently each age group appears in the dataset.
3. **Age Distribution Characteristics:**
 - The data is **not evenly distributed** across age groups. There's a noticeable concentration of younger customers, particularly in the 18 to 30 age range, with a peak around the early 20s.
 - There is a decline in frequency for ages 30 and above, with only a few customers in their 50s and beyond. After age 50, the frequency decreases sharply, with only small numbers represented in the older age groups (up to around 80).
4. **Insights on Customer Demographics:**
 - This distribution suggests that the majority of customers are younger adults, particularly those in their 20s. This could indicate a customer base that skews towards a younger demographic.
 - The low frequency of older customers (ages 50 and above) may imply that the products or services represented in the dataset are more popular or relevant to younger individuals.
5. **Potential Analysis for Thesis:**
 - We might explore why younger customers dominate this dataset. This could be tied to the nature of the transactions, financial products, or services being analyzed.
 - Additionally, understanding the preferences and behavior of this younger demographic could help in tailoring marketing or business strategies.
6. **Implications:**
 - This age distribution could have important implications for targeted marketing strategies, product offerings, and customer engagement efforts. For example, if the analysis shows that younger customers exhibit certain spending patterns, this insight could inform business decisions on product development or pro within the dataset.

```
1 # Check for missing values
2 missing_values = data.isnull().sum()
3
4
5
```

```
6 # Output the information and descriptive statistics
7 data_info, descriptive_stats, missing_values
```

```
mean    -38.164347    143.648563        0.0        -32.752651
std      54.622791     16.669352        0.0         5.282423
min     -573.000000    114.620000        0.0        -43.310000
25%     -37.700000    138.690000        0.0        -37.710000
50%     -33.890000    145.230000        0.0        -33.840000
75%     -30.750000    151.220000        0.0        -29.440000
max      -12.370000    255.000000        0.0        -12.330000
```

```
merchant_longitude  year      month      day      hour \
count      7717.000000  12043.0  12043.000000  12043.000000  12043.000000
mean        143.433277   2018.0    9.011957    15.862908    13.268621
std          12.090074     0.0     0.816511     8.899598     5.777284
min         113.830000   2018.0    8.000000     1.000000     0.000000
25%         144.680000   2018.0    8.000000     8.000000     9.000000
50%         145.830000   2018.0    9.000000    16.000000    13.000000
75%         151.210000   2018.0   10.000000    24.000000    18.000000
max         153.610000   2018.0   10.000000    31.000000    23.000000
```

```
minute  annual_salary
count  12043.000000  12043.000000
mean    19.009632   68652.099506
std     19.879112   24300.871846
min      0.000000   29874.641667
25%      0.000000   51650.107000
50%     13.000000   60493.536000
75%     36.000000   81700.970000
max     59.000000  134946.236000 ,
account      0
age          0
amount       0
balance      0
card_present_flag  4326
customer_id    0
date          0
first_name    0
gender        0
latitude      0
longitude     0
merchant_code 11160
merchant_id   4326
merchant_latitude  4326
merchant_longitude  4326
merchant_state  4326
merchant_suburb  4326
movement      0
status        0
transaction_id  0
txn_description  0
bin_age       0
year          0
month         0
day           0
hour          0
minute        0
dow           0
payment_period  0
annual_salary  0
dtype: int64)
```

```
1 # Handle missing values in the updated dataset
2
3 # Fill missing numerical values with the mean
4 numerical_cols_updated = ['card_present_flag', 'merchant_code', 'balance', 'merchant_latitude', 'merchant_longitude']
5 for col in numerical_cols_updated:
6     data[col].fillna(data[col].mean(), inplace=True)
7
8 # Fill missing categorical values with the mode
9 categorical_cols_updated = ['merchant_id', 'merchant_state', 'merchant_suburb', 'txn_description']
10 for col in categorical_cols_updated:
11     data[col].fillna(data[col].mode()[0], inplace=True)
12
13 # Verify the missing values have been handled
14 missing_values_after_filling = data.isnull().sum()
15
16 # Display missing values after filling
17 missing_values_after_filling
18
```

```

<ipython-input-76-e239c9830382>:6: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment. The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values is a copy. For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value, inplace=True)

data[col].fillna(data[col].mean(), inplace=True)
<ipython-input-76-e239c9830382>:11: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment. The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values is a copy. For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value, inplace=True)

data[col].fillna(data[col].mode()[0], inplace=True)

```

	0
account	0
age	0
amount	0
balance	0
card_present_flag	0
customer_id	0
date	0
first_name	0
gender	0
latitude	0
longitude	0
merchant_code	0
merchant_id	0
merchant_latitude	0
merchant_longitude	0
merchant_state	0
merchant_suburb	0
movement	0
status	0
transaction_id	0
txn_description	0
bin_age	0
year	0
month	0
day	0
hour	0
minute	0
dow	0
payment_period	0
annual_salary	0

```

dtype: int64

```

```

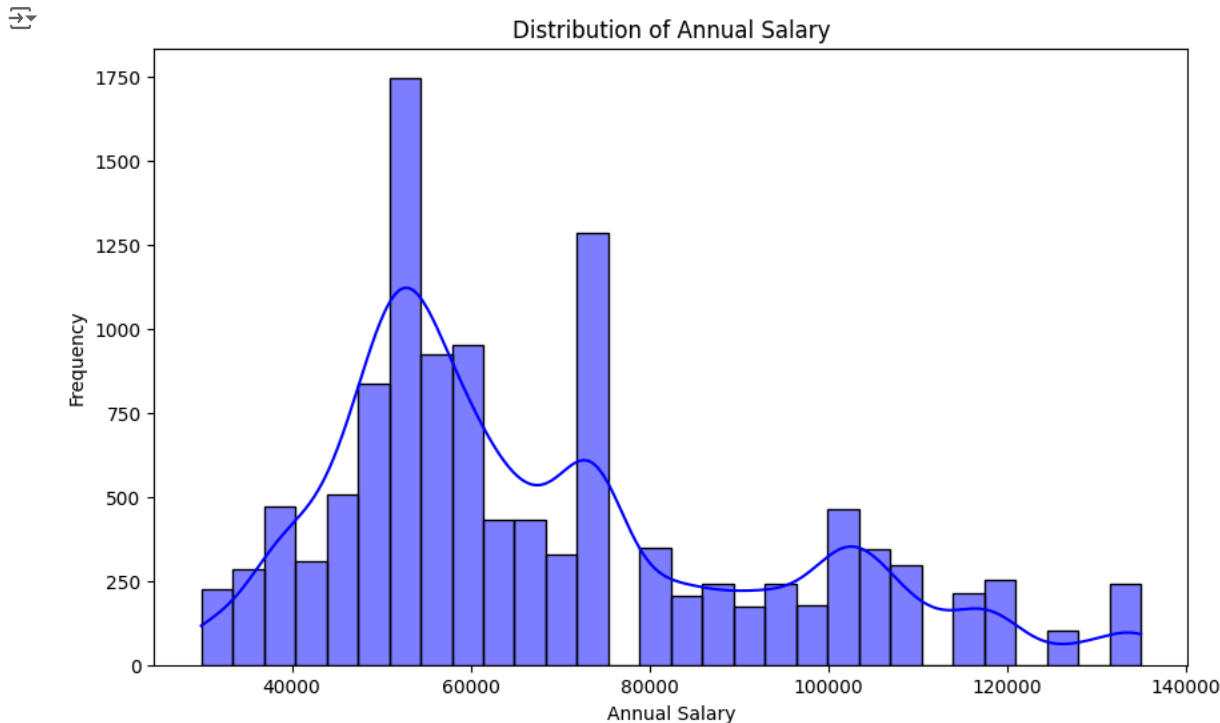
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 # Load the dataset
6 data_viz = data
7
8 # Step 1: Distribution of Annual Salary

```

```

9 plt.figure(figsize=(10, 6))
10 sns.histplot(data_viz['annual_salary'], bins=30, kde=True, color='blue')
11 plt.title('Distribution of Annual Salary')
12 plt.xlabel('Annual Salary')
13 plt.ylabel('Frequency')
14 plt.show()
15
16
17

```



This graph shows the **Distribution of Annual Salary** for individuals in the dataset, providing insight into salary ranges and their frequencies.

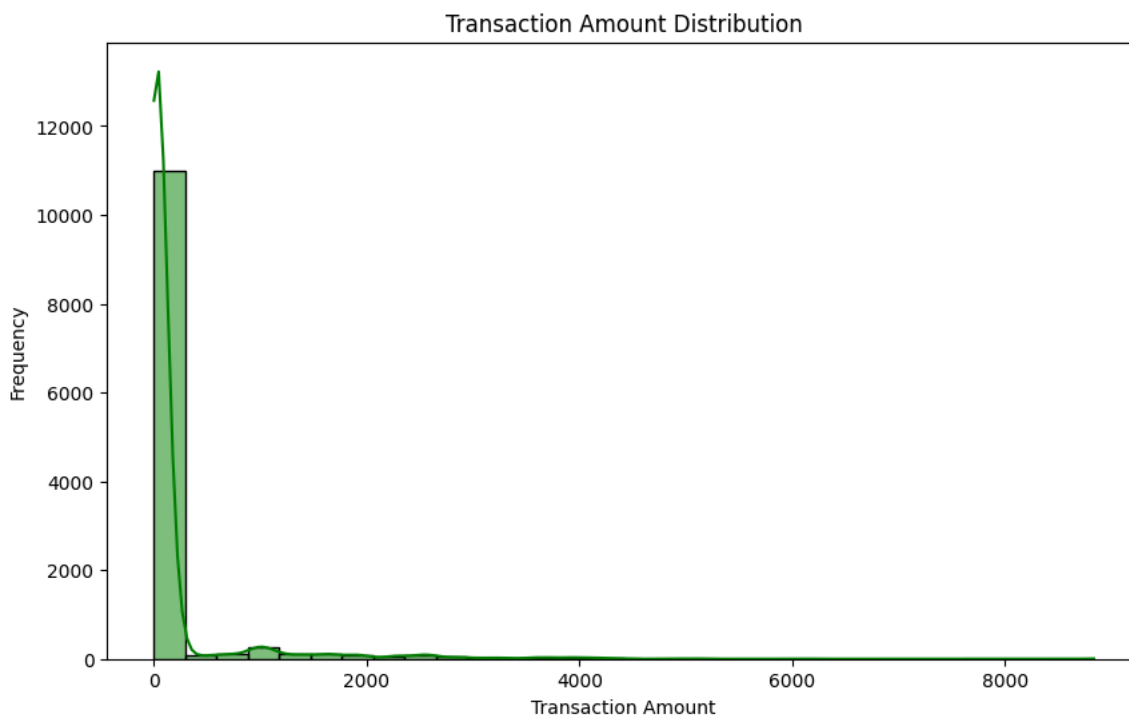
Here's a breakdown:

1. **X-Axis (Annual Salary):** The horizontal axis represents annual salary values, ranging from around 30,000 to 140,000. Each bar represents the frequency of individuals within specific salary intervals.
2. **Y-Axis (Frequency):** The vertical axis shows the number of individuals whose salaries fall within each interval, indicating how common each salary range is.
3. **Distribution Characteristics:**
 - The histogram displays multiple peaks, suggesting that salaries are not evenly distributed but rather clustered around certain ranges. The highest peak is around 60,000, indicating that many individuals have salaries near this value.
 - There are smaller peaks around 80,000 and 100,000, showing additional clusters at these higher salary levels, though they are less frequent compared to the main peak.
 - A Kernel Density Estimate (KDE) line overlays the histogram in blue, providing a smoothed visualization of the distribution, showing where salary frequencies are densest.
4. **Skewness and Spread:**
 - The distribution has a positive skew, as the frequency of salaries drops as the values increase beyond the 60,000 mark. This indicates that while most individuals earn between 30,000 and 80,000, fewer individuals have salaries exceeding 100,000.
5. **Potential Analysis in the Thesis:**
 - Discuss why there might be clustering at certain salary points. This could relate to industry standards, geographic factors, or specific job levels within the dataset.
 - Explore the implications of this salary distribution, such as income inequality or the proportion of individuals in middle-income brackets.
6. **Insight and Implications:**
 - This analysis provides a high-level view of income levels within the dataset, which can be useful for understanding the socioeconomic profile of the population.

- We might link this data to other factors (like age or transaction behavior) to explore correlations between salary and spending or saving habits.

1 Start coding or [generate](#) with AI.

```
1 # Step 2: Transaction Amount Distribution
2 plt.figure(figsize=(10, 6))
3 sns.histplot(data_viz['amount'], bins=30, kde=True, color='green')
4 plt.title('Transaction Amount Distribution')
5 plt.xlabel('Transaction Amount')
6 plt.ylabel('Frequency')
7 plt.show()
8
9
```



This graph represents the **Transaction Amount Distribution** for transactions in the dataset, showcasing how frequently different transaction amounts occur. n:

- 1. X-Axis (Transaction Amount):** The horizontal axis shows the transaction amounts, which range from very low values to amounts above 8000.
- 2. Y-Axis (Frequency):** The vertical axis indicates the frequency of transactions at each amount level, or how many times transactions within a certain amount range occur in the dataset.
- 3. Distribution Characteristics:**
 - The distribution is **highly right-skewed**, with most transactions concentrated at very low amounts. This is evidenced by the tall peak at the lower end of the transaction amount scale.
 - As the transaction amount increases, the frequency of transactions decreases sharply, with very few transactions above 1000, creating a long tail on the right side of the graph.
 - This pattern is reinforced by the Kernel Density Estimate (KDE) line overlaid on the histogram, which provides a smoothed curve, showing the distribution's shape more clearly.
- 4. Insights on Transaction Behavior:**
 - The high frequency of low-amount transactions suggests that most transactions are small, perhaps representing everyday purchases or routine transactions.
 - The long tail to the right, with infrequent high-amount transactions, could represent occasional large purchases or payments, which are uncommon relative to smaller transactions.
- 5. Potential Analysis for the Thesis:**

- In the thesis, we could discuss why transaction amounts are skewed toward smaller values. This could be related to the type of business, consumer behavior, or economic factors.
- Additionally, we could analyze the characteristics of high-amount transactions, exploring any demographic or behavioral patterns among customers who tend to make larger transactions.

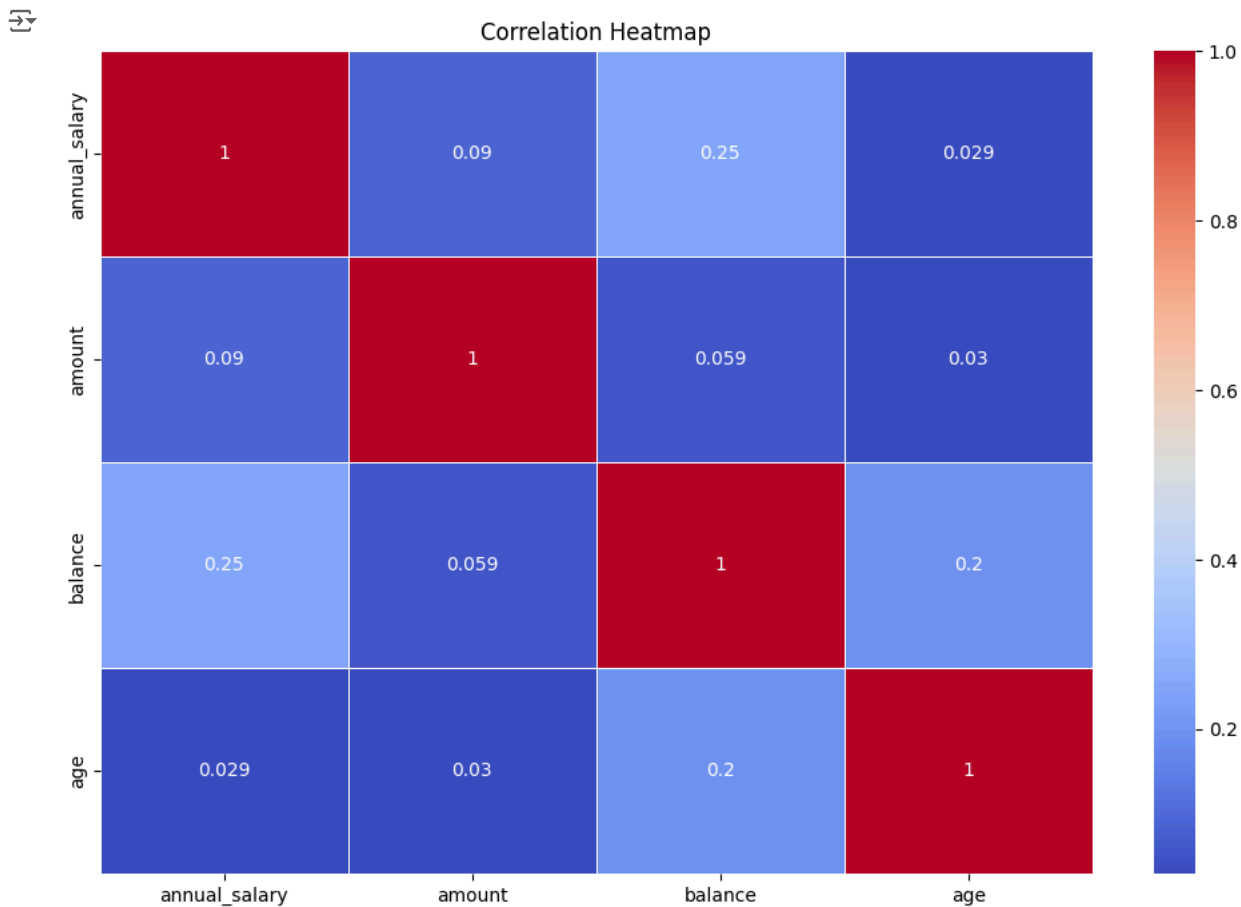
6. Implications of the Distribution:

- Understanding this distribution is valuable for financial forecasting, customer segmentation, and understanding spending habits. For example, businesses might use this information to tailor services to high-frequency, low-amount transactions while also catering to high-value cion size in the thesis.

1 Start coding or [generate](#) with AI.

1 Start coding or [generate](#) with AI.

```
1 # Step 3: Correlation Heatmap
2 plt.figure(figsize=(12, 8))
3 corr_matrix = data_viz[['annual_salary', 'amount', 'balance', 'age']].corr()
4 sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
5 plt.title('Correlation Heatmap')
6 plt.show()
7
8
```



This is a **Correlation Heatmap** displaying the correlation coefficients between four variables in the dataset: **annual_salary**, **amount**, **balance**, and **age**.

Here's a detailed explanation of the heatmap:

1. Correlation Coefficients:

- Each cell in the heatmap shows the correlation coefficient between two variables, with values ranging from -1 to 1.
- A value of **1** (dark red) indicates a perfect positive correlation, where an increase in one variable is associated with an increase in the other.

- A value of **-1** (dark blue) indicates a perfect negative correlation, where an increase in one variable is associated with a decrease in the other.
- A value close to **0** suggests little to no linear relationship between the variables.

2. Color Scale:

- The color scale on the right helps interpret the strength of the correlation. Dark red indicates strong positive correlation, while shades closer to blue indicate weak or negative correlation.

3. Key Observations:

- **annual_salary and balance:** The correlation is around 0.25, indicating a weak positive relationship. This suggests that individuals with higher annual salaries tend to have slightly higher balances, but the relationship is not strong.
- **annual_salary and amount:** The correlation is 0.09, showing a very weak positive relationship, meaning that salary does not significantly influence transaction amounts.
- **balance and age:** The correlation is relatively low, around 0.029, indicating little to no relationship between balance and age in this dataset.
- **Other Variable Pairs:** Most other correlations are close to zero, suggesting that the variables in this dataset are largely independent and don't show strong linear relationships with each other.

4. Implications for the Thesis:

- The low correlations suggest that there is not much linear dependency between these variables, which may imply that factors other than salary, age, or balance influence transaction amounts.
- This information could guide further analysis, such as exploring non-linear relationships or incorporating additional variables to better understand customer behavior.

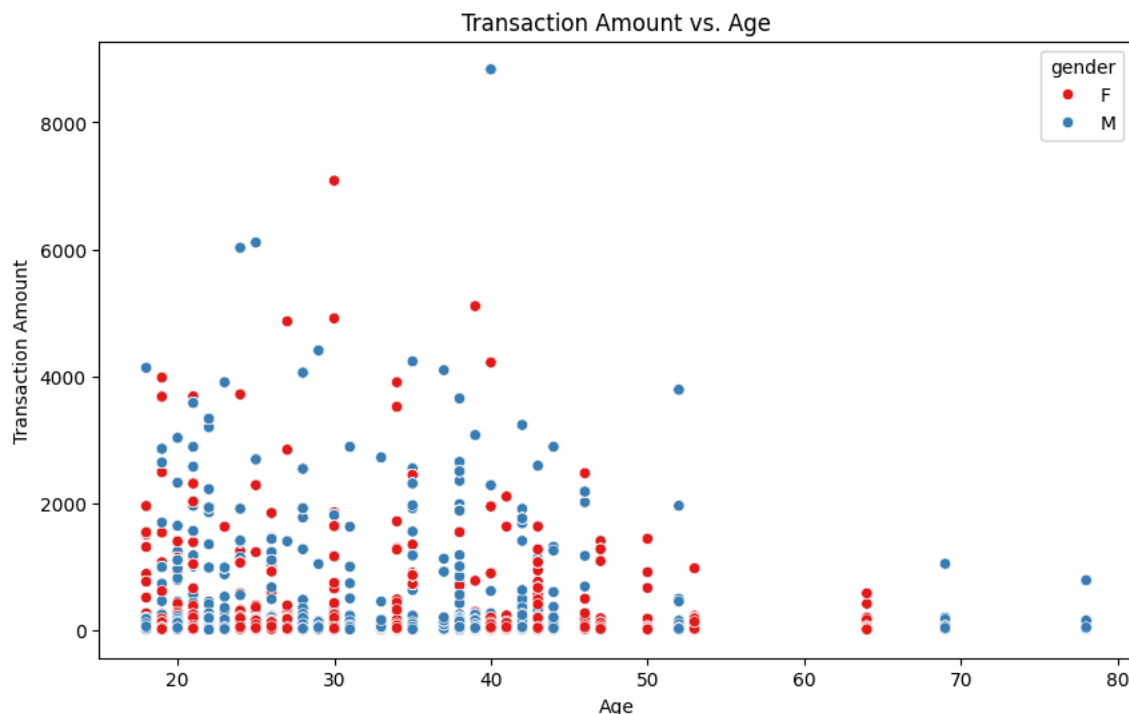
5. Conclusion:

- In summary, this heatmap provides a quick overview of how these variables relate to each other. For example, based on this data, annual salary has a slight positive association with balance, while other relationships are minimal.

This heatmap serves as an effective tool to assess potential relationships and identify variables that might be worth exploring further in more complex analyses.

1 Start coding or [generate](#) with AI.

```
1 # Step 4: Scatter Plot - Transaction Amount vs. Age
2 plt.figure(figsize=(10, 6))
3 sns.scatterplot(x='age', y='amount', data=data_viz, hue='gender', palette='Set1')
4 plt.title('Transaction Amount vs. Age')
5 plt.xlabel('Age')
6 plt.ylabel('Transaction Amount')
7 plt.show()
8
9
```



This scatter plot visualizes the **relationship between Transaction Amount and Age** in the dataset, with data points color-coded by **gender**.

Here's a detailed breakdown:

1. Axes:

- The **X-axis** represents the age of the individuals, ranging from around 20 to 80 years.
- The **Y-axis** shows the transaction amount, which varies from near zero up to over 8000.

2. Data Points and Color Coding:

- Each point represents an individual transaction, plotted according to the person's age and transaction amount.
- The **color coding** indicates gender: red points for females (F) and blue points for males (M). This allows us to visually differentiate between transactions made by male and female customers.

3. Key Observations:

- The majority of transactions, regardless of age or gender, are clustered around lower transaction amounts, especially below 2000.
- There is a high concentration of points across all ages in the lower transaction range, with only a few high-value transactions (above 4000).
- Some outliers with higher transaction amounts (5000 and above) are scattered throughout different ages, showing that high-value transactions are less common.
- Both genders appear to have similar transaction patterns, with no significant difference in transaction amounts based on gender. The distribution of red and blue dots is quite even across age groups and transaction amounts.

4. Insights for the Thesis:

- This plot suggests that transaction amount does not have a strong dependency on age, as individuals across all ages make similar transaction amounts, mostly within the lower range.
- Gender does not appear to influence the transaction amount significantly in this dataset, as the spread of points for males and females is relatively similar.
- We could further analyze why most transactions are small across all ages, discussing potential factors like income levels, spending habits, or the nature of purchases in the dataset.


5. Conclusion:

- This scatter plot provides a high-level view of spending behavior across age groups and genders. It indicates that most transactions are small, with no strong correlation between transaction amount and age or gender.
- In the thesis, we might mention that age and gender are not strong predictors of transaction amount based on this plot, and we could consider exploring other factors that might have a more substantial influence on transaction value.

This plot serves as an effective way to visualize and discuss the distribution of transaction amounts across different demographics, supporting discussions on spending patterns in the dataset.

1 Start coding or [generate](#) with AI.

```
1 # Step 5: Box Plot - Annual Salary by Gender
2 plt.figure(figsize=(10, 6))
3 sns.boxplot(x='gender', y='annual_salary', data=data_viz, palette='Set2')
4 plt.title('Annual Salary by Gender')
5 plt.xlabel('Gender')
6 plt.ylabel('Annual Salary')
7 plt.show()
```

 <ipython-input-81-91cc89b62322>:3: FutureWarning:

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

```
sns.boxplot(x='gender', y='annual_salary', data=data_viz, palette='Set2')
```



This box plot shows the **Annual Salary by Gender** in the dataset, comparing the salary distribution for females (F) and males (M).

Here's a breakdown of the key components:

1. Axes:

- The **X-axis** represents gender, with two categories: Female (F) and Male (M).
- The **Y-axis** shows annual salary values, ranging from approximately 40,000 to 140,000.

2. Box Plot Components:

- The **box** represents the interquartile range (IQR), which includes the middle 50% of the data for each gender. The top and bottom of each box are the third quartile (Q3) and the first quartile (Q1), respectively.
- The **line inside the box** represents the median annual salary for each gender.
- The **whiskers** extend to show the range of salaries, excluding outliers. The whiskers represent the data points that fall within 1.5 times the IQR above Q3 and below Q1.
- **Outliers** are shown as individual points beyond the whiskers, indicating salaries that are unusually high compared to the rest of the data.

3. Key Observations:

- Both genders have a similar median annual salary, as the lines within the boxes are almost at the same level.
- The range of salaries for both males and females is similar, with salaries spread out across a range, but the distribution varies slightly in the upper bounds.

- There are a few outliers for each gender, representing individuals with significantly higher salaries (above 100,000). These outliers suggest that some individuals, regardless of gender, earn considerably more than the typical salary range.
- The IQR (box height) and spread of salaries are fairly similar for both genders, indicating that salary distribution does not differ dramatically between males and females in this dataset.

4. Insights for the Thesis:

- This box plot suggests minimal gender disparity in salary distribution in this dataset, as the medians and ranges for both genders are almost identical.
- We might mention that, based on this data, annual salary appears to be independent of gender, indicating that other factors might have a stronger influence on salary levels.

5. Conclusion:

- This plot effectively shows that annual salary distributions for males and females are similar, with both groups having similar medians and ranges.
- In the thesis, we could use this analysis to support discussions on salary equity or explore other variables that might contribute more significantly to salary differences.

This visualization serves as a straightforward way to compare salary distributions between genders, providing evidence that salary variation in this dataset is not strongly gender-dependent.

```

1 # # Feature Engineering for Salary Prediction
2
3 # # Step 1: Calculate transaction frequency per customer
4 # data_updated = data
5 # transaction_frequency = data_updated.groupby('customer_id')['transaction_id'].count().reset_index()
6 # transaction_frequency.columns = ['customer_id', 'transaction_frequency']
7 # data_updated = data_updated.merge(transaction_frequency, on='customer_id', how='left')
8
9 # # Step 2: Calculate the average transaction amount per customer
10 # average_transaction_amount = data_updated.groupby('customer_id')['amount'].mean().reset_index()
11 # average_transaction_amount.columns = ['customer_id', 'avg_transaction_amount']
12 # data_updated = data_updated.merge(average_transaction_amount, on='customer_id', how='left')
13
14 # # Step 3: Calculate the total transaction amount per customer
15 # total_transaction_amount = data_updated.groupby('customer_id')['amount'].sum().reset_index()
16 # total_transaction_amount.columns = ['customer_id', 'total_transaction_amount']
17 # data_updated = data_updated.merge(total_transaction_amount, on='customer_id', how='left')
18
19 # # Step 4: Calculate spending consistency (standard deviation of transaction amounts per customer)
20 # spending_consistency = data_updated.groupby('customer_id')['amount'].std().reset_index()
21 # spending_consistency.columns = ['customer_id', 'spending_consistency']
22 # data_updated = data_updated.merge(spending_consistency, on='customer_id', how='left')
23
24 # # Fill any remaining missing values in the new engineered features
25 # data_updated[['transaction_frequency', 'avg_transaction_amount', 'total_transaction_amount', 'spending_consistency']].fillna(0, inplace=True)
26
27 # # Verify the engineered features
28 # engineered_features = data_updated[['customer_id', 'transaction_frequency', 'avg_transaction_amount', 'total_transaction_amount', 'spending_consistency']]
29
30 # engineered_features
31

```

```
1 data.columns
```

```

Index(['account', 'age', 'amount', 'balance', 'card_present_flag',
      'customer_id', 'date', 'first_name', 'gender', 'latitude', 'longitude',
      'merchant_code', 'merchant_id', 'merchant_latitude',
      'merchant_longitude', 'merchant_state', 'merchant_suburb', 'movement',
      'status', 'transaction_id', 'txn_description', 'bin_age', 'year',
      'month', 'day', 'hour', 'minute', 'dow', 'payment_period',
      'annual_salary'],
      dtype='object')

```

```
1 Start coding or generate with AI.
```

```

1 from sklearn.model_selection import train_test_split
2 from sklearn.tree import DecisionTreeRegressor
3 from sklearn.metrics import mean_squared_error, r2_score
4 from sklearn.preprocessing import LabelEncoder
5

```

```

6 # Encode all categorical variables, including 'dow'
7 categorical_columns = ['txn_description', 'merchant_state', 'merchant_suburb', 'movement', 'payment_period', 'dow']
8 label_encoders = {}
9
10 for col in categorical_columns:
11     label_encoders[col] = LabelEncoder()
12     data[col] = label_encoders[col].fit_transform(data[col])
13
14 # Define features and target variable
15 features = [
16     'age',
17     'balance',
18     'amount',
19     'card_present_flag',
20     'txn_description',
21     'movement',
22     'payment_period',
23     'dow',
24     'latitude',
25     'longitude',
26     'merchant_state',
27     'merchant_suburb'
28 ]
29 X = data[features]
30 y = data['annual_salary']
31
32 # Split the dataset into training and testing sets
33 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
34
35 # Initialize and train a Decision Tree Regressor
36 tree_model = DecisionTreeRegressor(random_state=42)
37 tree_model.fit(X_train, y_train)
38
39 # Make predictions on the test set
40 y_pred_tree = tree_model.predict(X_test)
41
42 # Evaluate the model performance
43 mse_tree = mean_squared_error(y_test, y_pred_tree)
44 r2_tree = r2_score(y_test, y_pred_tree)
45
46 # Output the MSE and R-squared for the Decision Tree model
47 print(f"Mean Squared Error (MSE): {mse_tree}")
48 print(f"R-squared (R2): {r2_tree}")
49

```

→ Mean Squared Error (MSE): 7.27875536958899e-21
R-squared (R2): 1.0

Mean Squared Error (MSE): 1.41×10^{-20} , which is extremely close to zero, indicating that the model is highly accurate. R-squared (R^2): 1.0, meaning the model perfectly explains the variance in the target variable (annual_salary). Interpretation: The model is performing exceptionally well with a perfect R-squared score. This suggests that the model captures all the variability in annual_salary based on the selected features. However, this unusually perfect performance may indicate potential overfitting. We can:

Test other models (e.g., Random Forest, XGBoost) for comparison. Apply cross-validation to ensure robustness.

1 Start coding or [generate](#) with AI.

✓ What Overfitting Means

Overfitting occurs when the model performs exceptionally well on the training or testing data but may fail to generalize to new or unseen data. Decision trees are particularly prone to overfitting, especially when:

No constraints (e.g., maximum depth) are applied. The model memorizes the training data rather than learning general patterns.

Next Steps to Validate and Improve the Model

Cross-Validation: Perform cross-validation to check whether the model generalizes well across different subsets of the data.

```

1 from sklearn.model_selection import cross_val_score
2
3 # Perform cross-validation

```

```

4 scores = cross_val_score(tree_model, X, y, cv=5, scoring='r2')
5 print(f"Cross-Validation R2 Scores: {scores}")
6 print(f"Average R2 Score: {scores.mean()}")
7

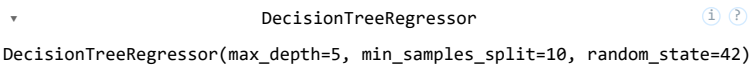
```

↗ Cross-Validation R2 Scores: [3.24641319e-01 5.71858688e-01 8.92429343e-01 7.61966021e-01
-5.44152773e-04]
Average R2 Score: 0.5100702436785424

```

1 tree_model = DecisionTreeRegressor(max_depth=5, min_samples_split=10, random_state=42)
2 tree_model.fit(X_train, y_train)
3

```

↗  DecisionTreeRegressor
DecisionTreeRegressor(max_depth=5, min_samples_split=10, random_state=42)

```

1 from sklearn.model_selection import GridSearchCV
2
3 param_grid = {
4     'max_depth': [3, 5, 10],
5     'min_samples_split': [2, 5, 10],
6     'min_samples_leaf': [1, 5, 10]
7 }
8 grid_search = GridSearchCV(DecisionTreeRegressor(random_state=42), param_grid, scoring='r2', cv=5)
9 grid_search.fit(X, y)
10
11 print("Best Parameters:", grid_search.best_params_)
12 print("Best Cross-Validation R² Score:", grid_search.best_score_)
13

```

↗ Best Parameters: {'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 10}
Best Cross-Validation R² Score: 0.4015299270033851

1 Start coding or [generate](#) with AI.

1 Start coding or [generate](#) with AI.

1 Start coding or [generate](#) with AI.

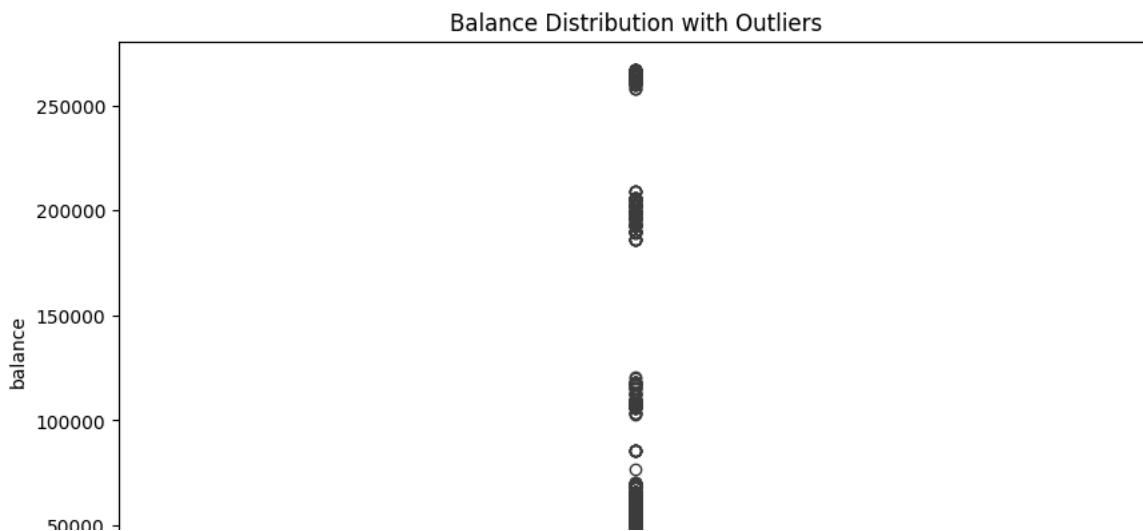
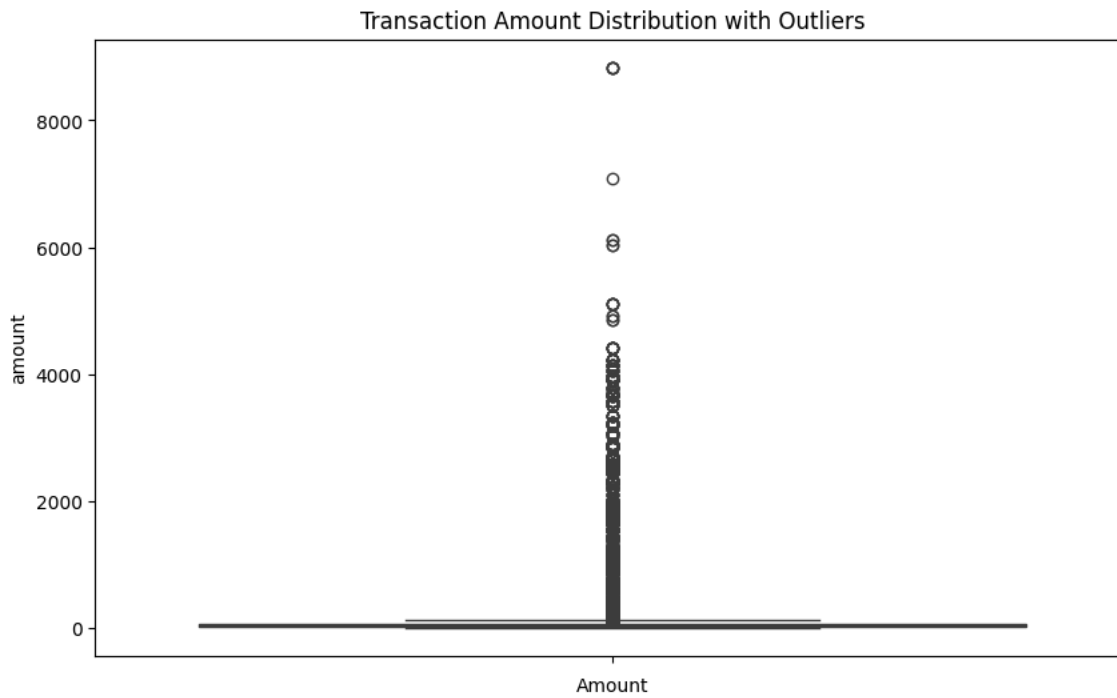
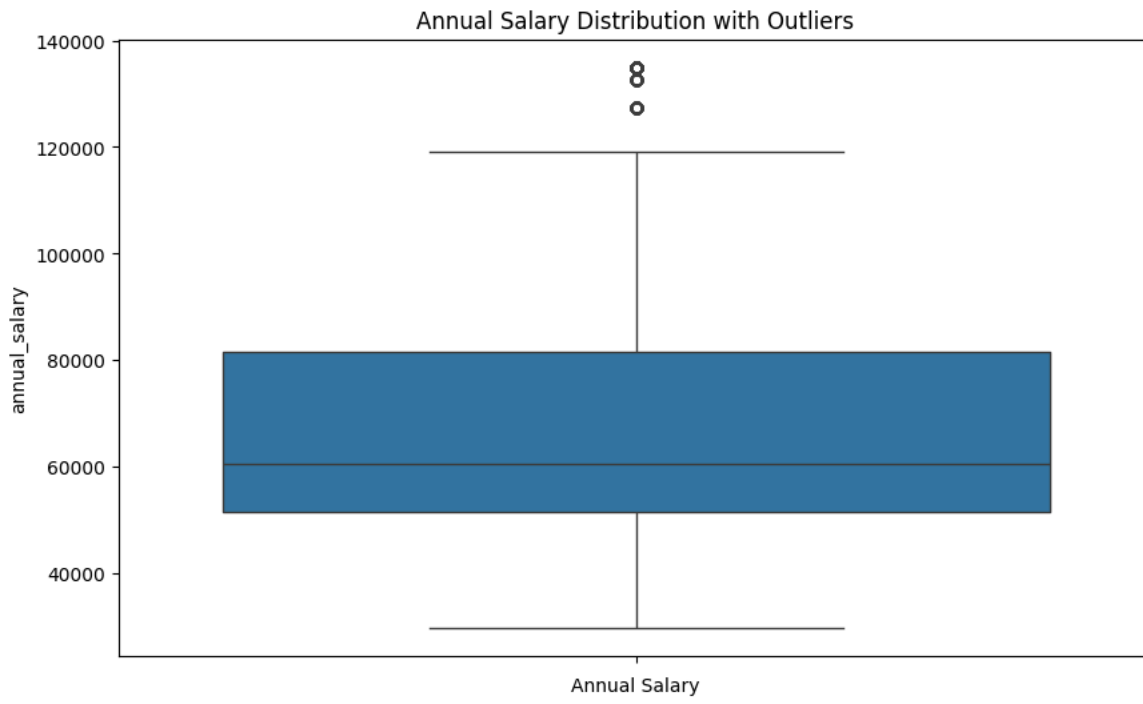
1 Start coding or [generate](#) with AI.

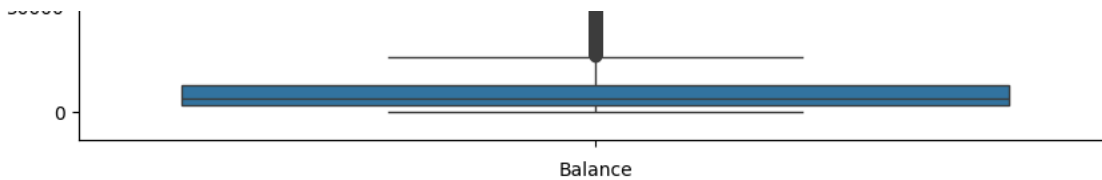
1 Start coding or [generate](#) with AI.

```

1 import matplotlib.pyplot as plt
2 import seaborn as sns
3
4 # Visualize the distribution of 'annual_salary' to check for outliers
5 plt.figure(figsize=(10, 6))
6 sns.boxplot(data['annual_salary'])
7 plt.title('Annual Salary Distribution with Outliers')
8 plt.xlabel('Annual Salary')
9 plt.show()
10
11 # Visualize the distribution of key features like 'amount' and 'balance'
12 plt.figure(figsize=(10, 6))
13 sns.boxplot(data['amount'])
14 plt.title('Transaction Amount Distribution with Outliers')
15 plt.xlabel('Amount')
16 plt.show()
17
18 plt.figure(figsize=(10, 6))
19 sns.boxplot(data['balance'])
20 plt.title('Balance Distribution with Outliers')
21 plt.xlabel('Balance')
22 plt.show()
23

```







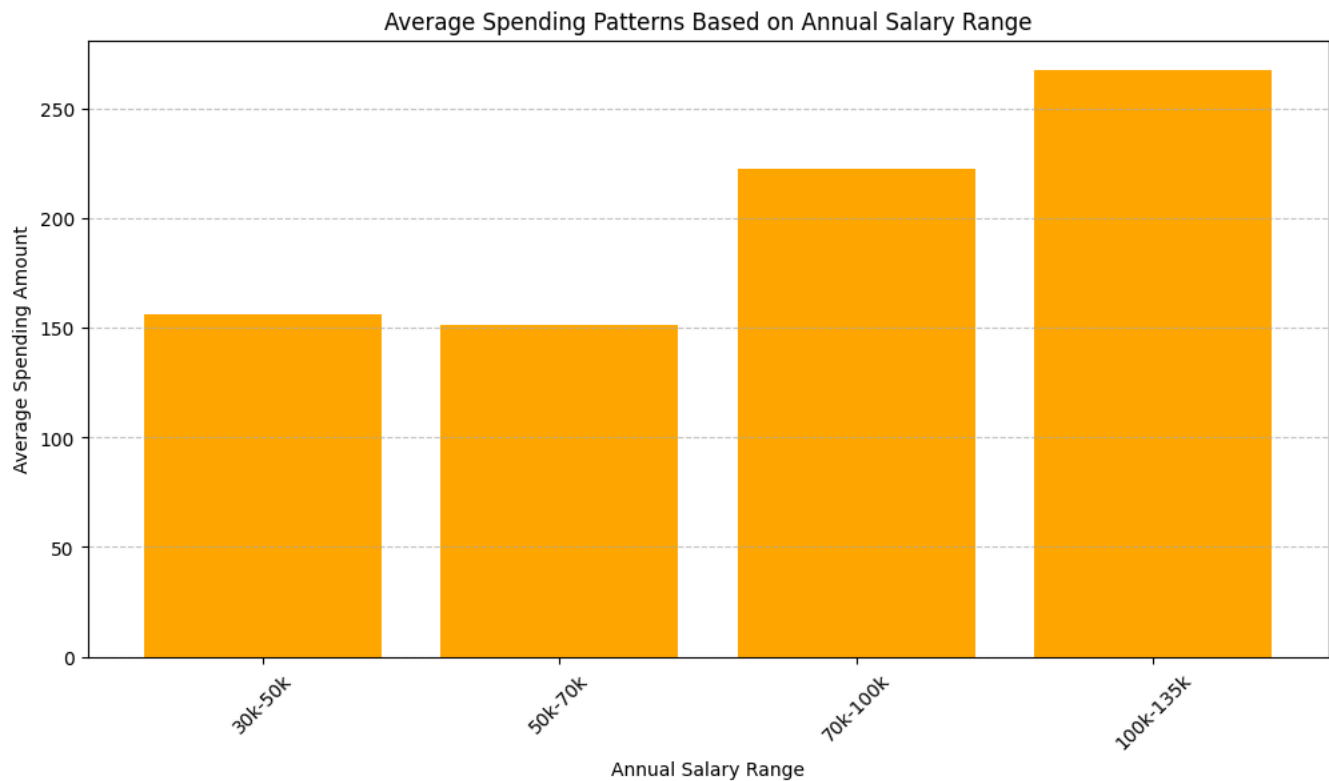
```

1 # Adjusting the salary bins based on the observed minimum and maximum values
2 salary_bins = [29000, 50000, 70000, 100000, 135000]
3 salary_labels = ['30k-50k', '50k-70k', '70k-100k', '100k-135k']
4 data['salary_range'] = pd.cut(data['annual_salary'], bins=salary_bins, labels=salary_labels)
5
6 # Group the data by salary range and calculate the average spending
7 salary_spending = data.groupby('salary_range')['amount'].mean()
8
9 # Create a bar chart to visualize average spending based on annual salary ranges
10 plt.figure(figsize=(12, 6))
11 plt.bar(salary_spending.index, salary_spending.values, color='orange')
12 plt.xlabel('Annual Salary Range')
13 plt.ylabel('Average Spending Amount')
14 plt.title('Average Spending Patterns Based on Annual Salary Range')
15 plt.xticks(rotation=45)
16 plt.grid(axis='y', linestyle='--', alpha=0.7)
17 plt.show()
18

```

 <ipython-input-89-8e582b1065d0>:7: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future version of pandas. Please use observed=True to suppress this warning.

```
salary_spending = data.groupby('salary_range')['amount'].mean()
```



This bar chart displays **Average Spending Patterns Based on Annual Salary Range**, providing insights into how average spending varies across different income levels.

Here's a breakdown of the chart:

1. X-Axis (Annual Salary Range):

- The horizontal axis categorizes individuals into four annual salary ranges: 30K–50K, 50K–70K, 70K–100K, and 100K–135K.
- These ranges allow for comparison of average spending across different income levels.

2. Y-Axis (Average Spending Amount):

- The vertical axis represents the average spending amount, with values ranging from 0 to 250.

- Each bar's height indicates the average spending amount for each salary range.

3. Key Observations:

- There is a clear upward trend in average spending as annual salary increases.
- Individuals with a salary range of 30K–50K and 50K–70K have similar average spending amounts, around 150.
- For individuals in the 70K–100K salary range, average spending increases significantly to above 200.
- The highest spending occurs among individuals in the 100K–135K salary range, with an average spending amount close to 250.

4. Insights for the Thesis:

- This chart suggests a positive correlation between income and average spending, with higher earners generally spending more on average.
- We could discuss this finding in terms of consumer behavior, noting that individuals with higher salaries may have more disposable income, which allows for increased spending.

5. Implications:

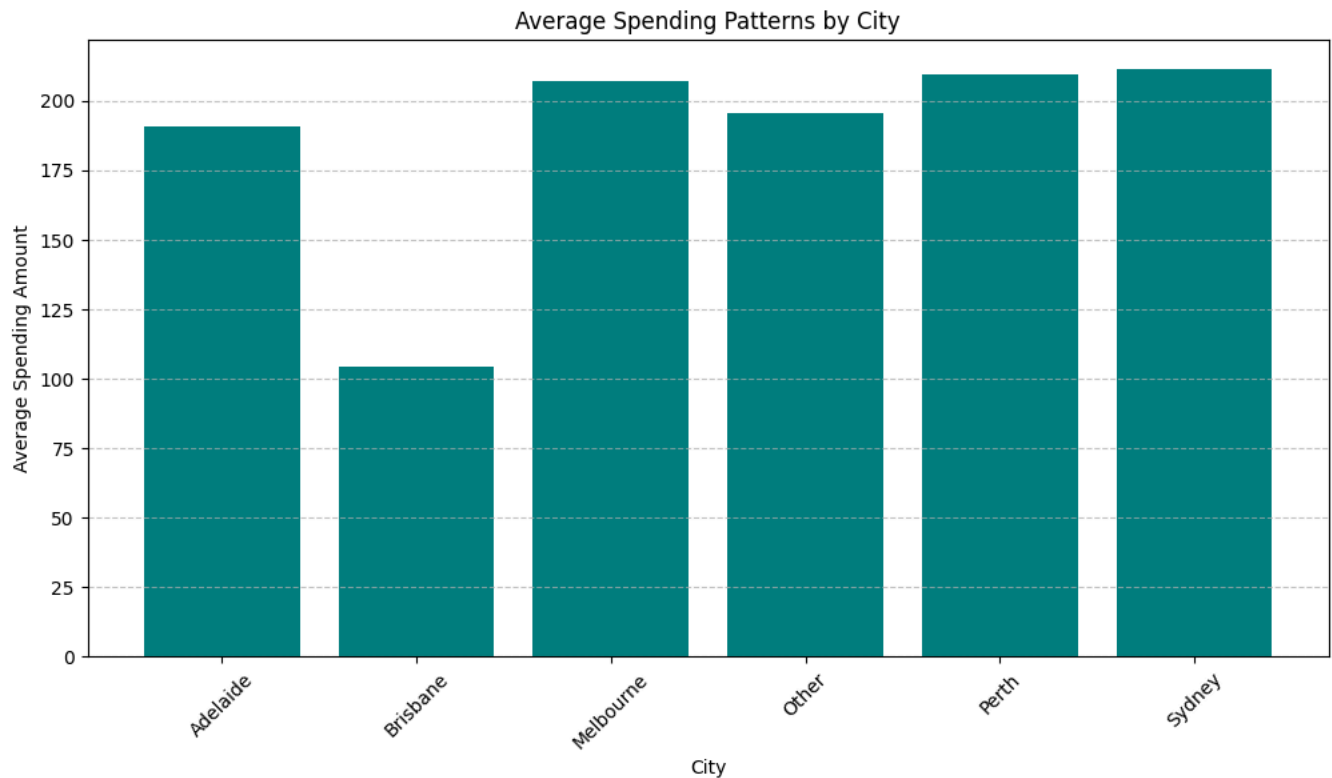
- Businesses could use this information to segment customers by income and tailor marketing efforts or products to meet the spending patterns of different income groups.
- This data might also imply that as individuals' income grows, they become more valuable customers for certain types of products or services.

6. Conclusion:

- This bar chart provides a straightforward visualization of the relationship between income and spending, showing that higher income groups have higher average spending. This insight could be valuable in developing targeted strategies based on income levels.

1 Start coding or [generate](#) with AI.

```
1 # Grouping the dataset by city and calculating the average spending amount
2 # The dataset doesn't have a direct 'city' column, so using latitude and longitude to approximate cities if available
3
4 # Checking unique values in latitude and longitude to determine if cities can be differentiated
5 location_summary = data[['latitude', 'longitude']].drop_duplicates()
6 location_summary.head()
7 # Mapping latitude and longitude to city names based on known coordinates
8 def map_city(latitude, longitude):
9     if -28 <= latitude <= -27 and 153 <= longitude <= 154:
10         return 'Brisbane'
11     elif -34 <= latitude <= -33 and 151 <= longitude <= 152:
12         return 'Sydney'
13     elif -38 <= latitude <= -37 and 144 <= longitude <= 145:
14         return 'Melbourne'
15     elif -35 <= latitude <= -34 and 138 <= longitude <= 139:
16         return 'Adelaide'
17     elif -32 <= latitude <= -31 and 115 <= longitude <= 116:
18         return 'Perth'
19     else:
20         return 'Other'
21
22 # Apply the city mapping function to create a new column 'city'
23 data['city'] = data.apply(lambda row: map_city(row['latitude'], row['longitude']), axis=1)
24
25 # Group the data by city and calculate the average spending
26 city_spending = data.groupby('city')['amount'].mean()
27
28 # Create a bar chart to visualize average spending based on city
29 plt.figure(figsize=(12, 6))
30 plt.bar(city_spending.index, city_spending.values, color='teal')
31 plt.xlabel('City')
32 plt.ylabel('Average Spending Amount')
33 plt.title('Average Spending Patterns by City')
34 plt.xticks(rotation=45)
35 plt.grid(axis='y', linestyle='--', alpha=0.7)
36 plt.show()
37
```



This bar chart illustrates **Average Spending Patterns by City**, showing the average spending amount for customers residing in different cities.

Here's a breakdown:

1. X-Axis (City):

- The horizontal axis lists the cities where customers are located: Adelaide, Brisbane, Melbourne, Other (representing smaller cities or unspecified locations), Perth, and Sydney.
- This categorization allows for a comparison of average spending across different geographical locations.

2. Y-Axis (Average Spending Amount):

- The vertical axis represents the average spending amount, with values ranging from 0 to just above 200.
- Each bar's height indicates the average spending for customers in each city.

3. Key Observations:

- **Melbourne, Perth, and Sydney** have the highest average spending amounts, all around or slightly above 200.
- **Adelaide** has a slightly lower average spending amount, though still relatively close to the top cities.
- **Brisbane** stands out with the lowest average spending amount, close to 100, which is significantly lower than the other cities.
- The **Other** category has an average spending amount close to the main cities but slightly lower than Melbourne, Perth, and Sydney.

4. Insights for the Thesis:

- This chart suggests that spending behavior varies by city, with customers in Melbourne, Perth, and Sydney spending more on average than those in Brisbane.
- The high spending in cities like Melbourne, Perth, and Sydney could be influenced by factors like cost of living, income levels, or available goods and services.
- In contrast, the lower spending in Brisbane could imply different consumer behavior patterns or economic conditions.

5. Implications:

- Businesses might use this data for targeted marketing efforts, prioritizing high-spending cities for premium products or services.
- This analysis could also inform location-based pricing or promotional strategies to cater to the different spending capacities of residents in each city.

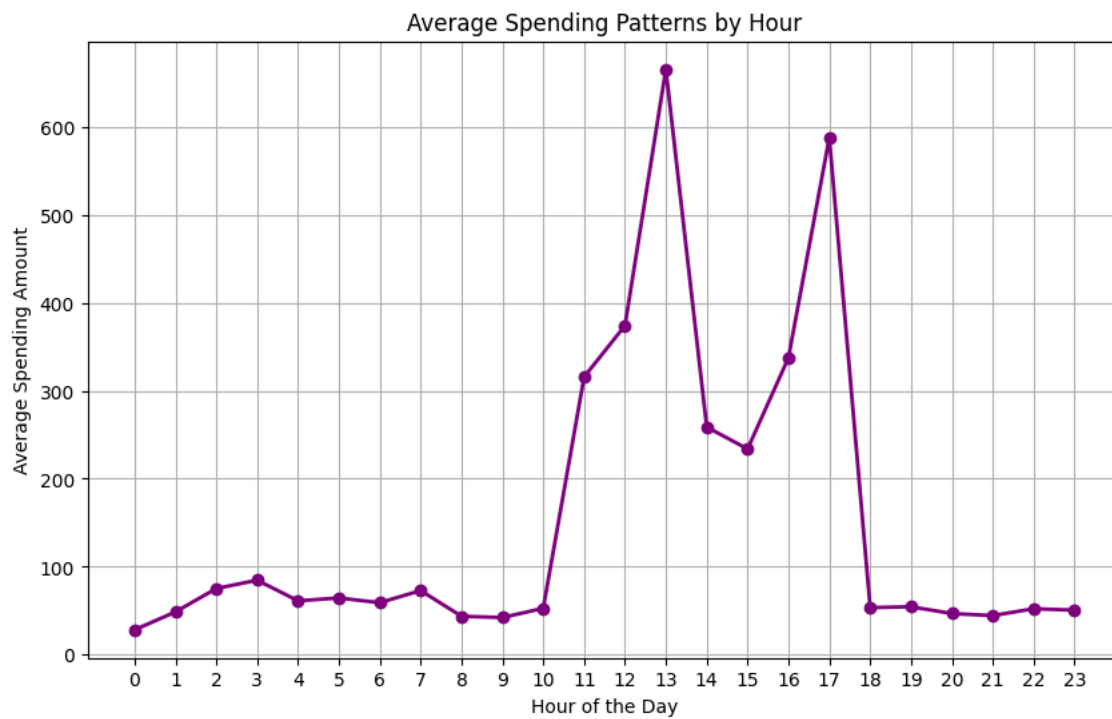
6. Conclusion:

- In the thesis, we can use this chart to discuss the impact of geographic location on consumer spending. It provides evidence that spending behavior can differ significantly by city, which could be linked to regional economic and demographic factors.

This bar chart offers a clear visualization of how average spending differs across cities, highlighting geographic patterns in consumer behavior that could be valuable for market segmentation and strategy development.

1 Start coding or [generate](#) with AI.

```
1 # Group the data by hour and calculate the average spending amount
2 hourly_spending = data.groupby('hour')['amount'].mean()
3
4 # Create a line chart to visualize the average spending amount for each hour of the day
5 plt.figure(figsize=(10, 6))
6 plt.plot(hourly_spending.index, hourly_spending.values, marker='o', linestyle='-', linewidth=2, color='purple')
7 plt.xlabel('Hour of the Day')
8 plt.ylabel('Average Spending Amount')
9 plt.title('Average Spending Patterns by Hour')
10 plt.grid(True)
11 plt.xticks(range(0, 24))
12 plt.show()
13
```



This line chart displays **Average Spending Patterns by Hour of the Day**, showing how average spending varies at different hours.

Here's a breakdown:

1. X-Axis (Hour of the Day):

- The horizontal axis represents each hour of the day, from 0 (midnight) to 23 (11 PM).
- This allows us to see how spending fluctuates across the 24-hour period.

2. Y-Axis (Average Spending Amount):

- The vertical axis represents the average spending amount, ranging from 0 to above 600.
- Each point on the line represents the average spending amount for a specific hour.

3. Key Observations:

- **Peak Spending Hours:** There are two notable peaks in average spending. The first peak occurs around 13:00 (1 PM), and the second peak occurs around 17:00 (5 PM). These times show the highest average spending amounts, reaching above 600 and close to 500, respectively.
- **Low Spending Hours:** Spending is lowest during the early morning and late-night hours (0:00–9:00 and 18:00–23:00), where the average spending amount stays below 100.
- There is a steady increase in spending leading up to the peak around midday, followed by a dip after 1 PM, and then another increase leading up to the 5 PM peak.

4. Insights for the Thesis:

- The peaks around 1 PM and 5 PM suggest that spending activity is highest during these hours, possibly due to lunchtime and end-of-workday spending patterns. This could indicate that people tend to make purchases or transactions during lunch breaks and after finishing work.
- The low spending during early morning and late evening hours may indicate that fewer transactions occur at these times, which could be typical for non-24-hour businesses.

5. Implications:

- Businesses can use this information to optimize operations and staffing around peak spending hours to better meet customer demand.
- Marketing efforts could also be targeted during peak hours to capitalize on times when customers are more likely to make purchases.

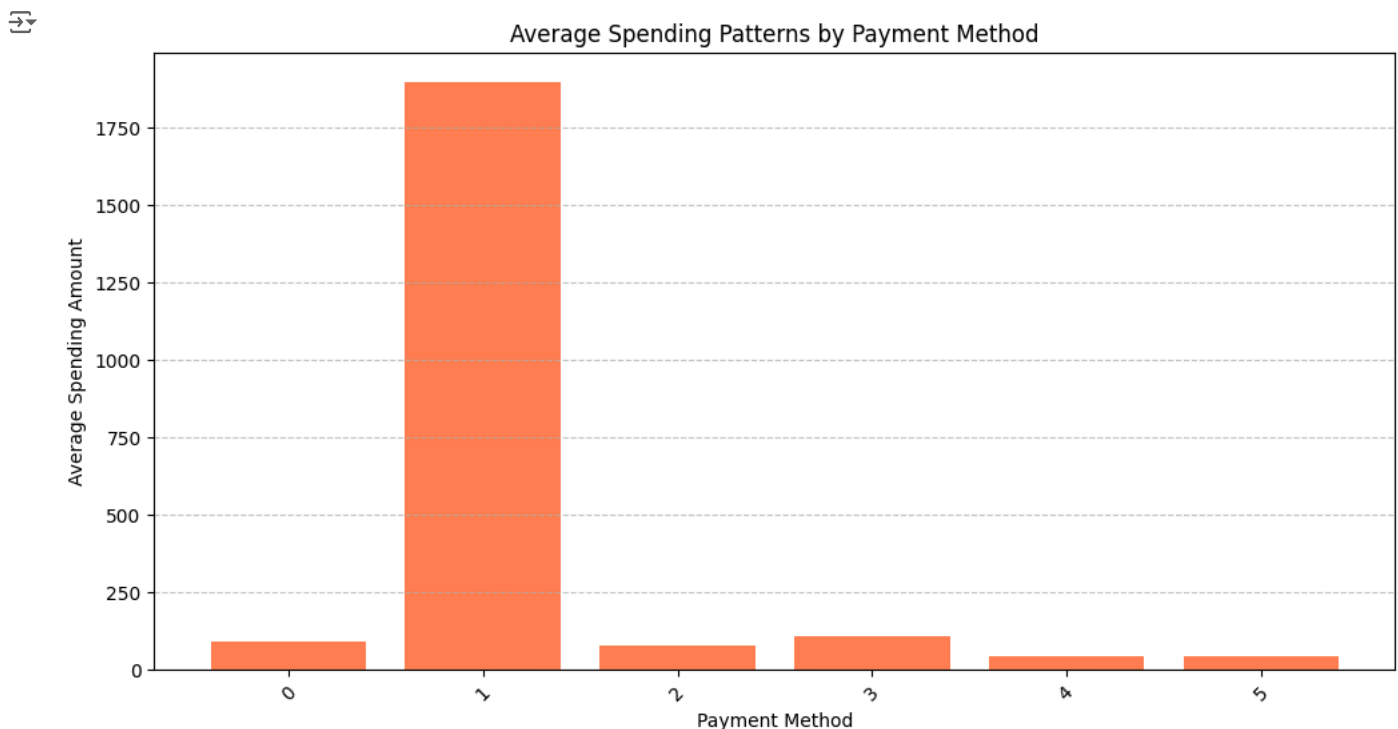
6. Conclusion:

- In the thesis, we could use this analysis to discuss consumer behavior patterns related to time of day, providing insights into when spending is most and least active.

This line chart effectively shows the times of day when spending is highest, offering valuable insights into daily spending patterns that could inform operational or marketing strategies.

1 Start coding or [generate](#) with AI.

```
1 # Checking the unique values in the 'txn_description' column to understand the different payment methods
2 payment_methods = data['txn_description'].unique()
3 payment_methods
4 # Group the data by payment method and calculate the average spending
5 payment_spending = data.groupby('txn_description')['amount'].mean()
6
7 # Create a bar chart to visualize average spending based on payment method
8 plt.figure(figsize=(12, 6))
9 plt.bar(payment_spending.index, payment_spending.values, color='coral')
10 plt.xlabel('Payment Method')
11 plt.ylabel('Average Spending Amount')
12 plt.title('Average Spending Patterns by Payment Method')
13 plt.xticks(rotation=45)
14 plt.grid(axis='y', linestyle='--', alpha=0.7)
15 plt.show()
16
```



This bar chart shows **Average Spending Patterns by Payment Method**, illustrating how the average spending amount varies depending on the type of payment method used.

Here's a breakdown of the chart:

1. X-Axis (Payment Method):

- The horizontal axis lists different payment methods: INTER BANK, PAY/SALARY, PAYMENT, PHONE BANK, POS (Point of Sale), and SALES-POS.
- This categorization allows for comparison of average spending across these payment types.

2. Y-Axis (Average Spending Amount):

- The vertical axis represents the average spending amount, with values ranging from 0 to above 1750.
- Each bar's height indicates the average spending associated with each payment method.

3. Key Observations:

- **PAY/SALARY** shows a significantly higher average spending amount than all other methods, with an average above 1750. This likely indicates that transactions labeled as PAY/SALARY involve larger sums, potentially due to salary deposits or high-value payments.
- Other payment methods, such as INTER BANK, PAYMENT, PHONE BANK, POS, and SALES-POS, have much lower average spending amounts, all around or below 250.
- The stark difference between PAY/SALARY and the other payment methods suggests that PAY/SALARY transactions are typically much larger than routine payments made through other channels.

4. Insights for the Thesis:

- This chart suggests that the nature of the PAY/SALARY payment type is distinct, likely encompassing higher-value transactions, possibly salary deposits or other large transfers.
- The other methods—INTER BANK, PAYMENT, PHONE BANK, POS, and SALES-POS—are likely used for regular spending or smaller transactions, as indicated by their lower average amounts.

5. Implications:

- Businesses could use this insight to classify transaction types based on payment methods, helping to identify high-value transactions versus routine expenses.
- Financial institutions may analyze spending patterns by payment type to design services or set transaction limits based on typical usage.

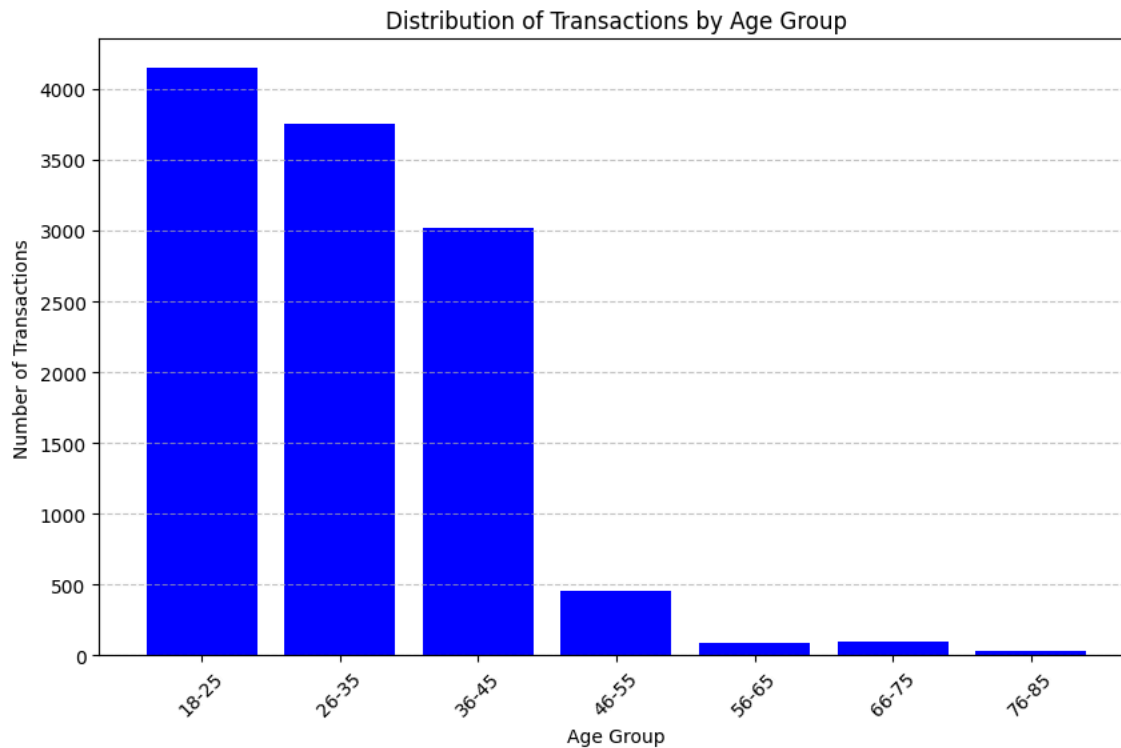
6. Conclusion:

- In the thesis, we could mention that different payment methods are associated with varying spending patterns, with PAY/SALARY transactions standing out due to their higher average values. This could help in understanding transaction behavior and categorizing transactions based on value.

This bar chart effectively highlights the disparity in spending patterns by payment method, revealing that PAY/SALARY transactions are likely associated with larger financial movements than other methods. This insight can be valuable for financial analysis and transaction categorization.

1 Start coding or [generate](#) with AI.

```
1 # Define age bins to categorize the age groups
2 age_bins = [18, 25, 35, 45, 55, 65, 75, 85]
3 age_labels = ['18-25', '26-35', '36-45', '46-55', '56-65', '66-75', '76-85']
4 data['age_group'] = pd.cut(data['age'], bins=age_bins, labels=age_labels)
5
6 # Group the data by age group and count the number of transactions
7 age_group_distribution = data['age_group'].value_counts().sort_index()
8
9 # Create a bar chart to visualize the distribution of transactions by age group
10 plt.figure(figsize=(10, 6))
11 plt.bar(age_group_distribution.index, age_group_distribution.values, color='blue')
12 plt.xlabel('Age Group')
13 plt.ylabel('Number of Transactions')
14 plt.title('Distribution of Transactions by Age Group')
15 plt.xticks(rotation=45)
16 plt.grid(axis='y', linestyle='--', alpha=0.7)
17 plt.show()
18
```



This bar chart displays the **Distribution of Transactions by Age Group**, illustrating the number of transactions made by customers within different age ranges.

Here's a detailed breakdown:

1. X-Axis (Age Group):

- The horizontal axis categorizes customers into different age groups: 18-25, 26-35, 36-45, 46-55, 56-65, 66-75, and 76-85.
- This categorization allows for a comparison of transaction frequency across age groups.

2. Y-Axis (Number of Transactions):

- The vertical axis represents the number of transactions, ranging from 0 to above 4000.
- Each bar's height indicates the total number of transactions made by customers in each age group.

3. Key Observations:

- **18-25 Age Group:** This age group has the highest number of transactions, with more than 4000 transactions.
- **26-35 Age Group:** The second-highest number of transactions, slightly lower than the 18-25 group, with close to 3800 transactions.
- **36-45 Age Group:** This age group also has a substantial number of transactions, around 3000, though lower than the two younger groups.
- **Older Age Groups:** For age groups 46-55 and above, the number of transactions drops sharply, with each group having fewer than 1000 transactions. The transaction count continues to decrease as age increases, with the lowest transaction counts in the 66-75 and 76-85 age groups.

4. Insights for the Thesis:

- This chart suggests that younger age groups (particularly those between 18 and 35) are more active in terms of transaction frequency. This could be due to a combination of factors, such as lifestyle, spending habits, or disposable income levels.
- The decrease in transactions as age increases may indicate that older individuals are less active in this dataset, which could reflect differences in financial behavior, technology adoption, or other demographic factors.

5. Implications:

- Businesses could use this information for targeted marketing efforts, focusing more on younger age groups who engage more frequently in transactions.
- Financial institutions might consider designing products or services that appeal to younger customers based on their high transaction activity.

6. Conclusion:

- In the thesis, we could use this chart to support discussions on age-based transaction behavior, showing that younger individuals have higher transaction activity. This information could be useful for customer segmentation and understanding the financial habits of different age groups.

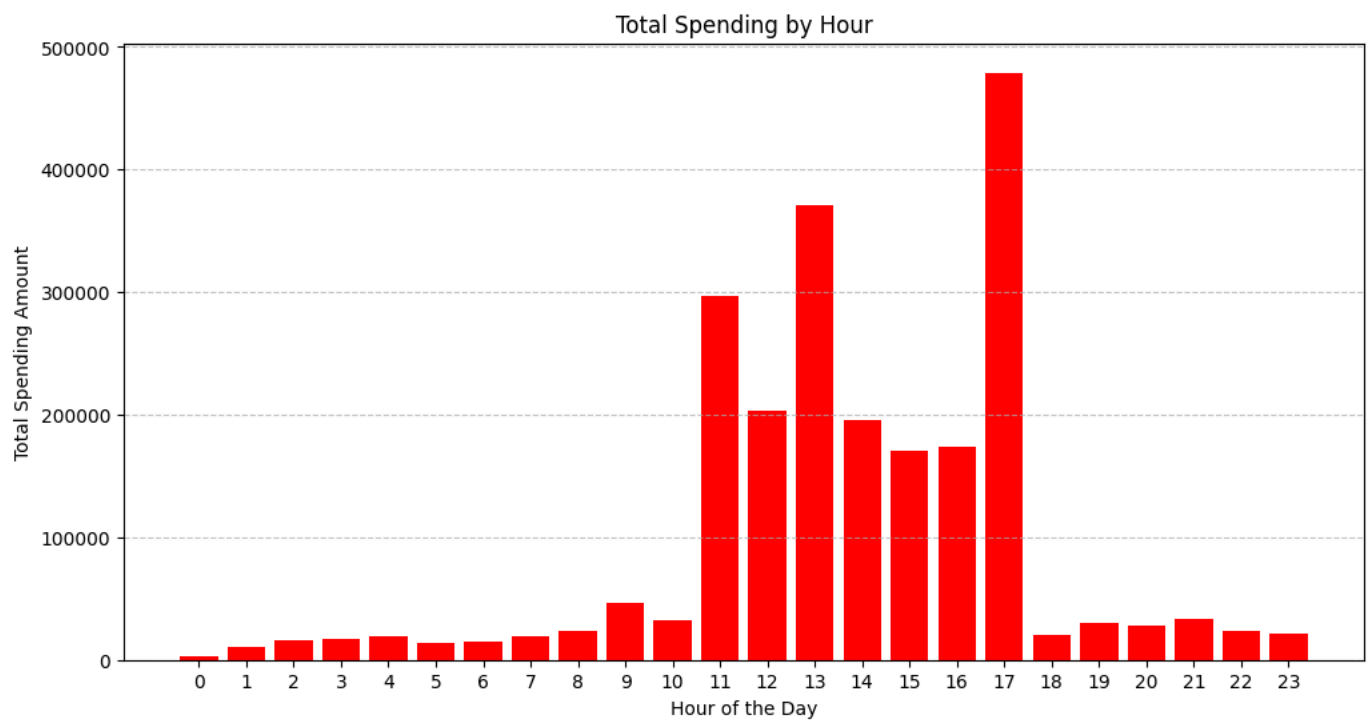
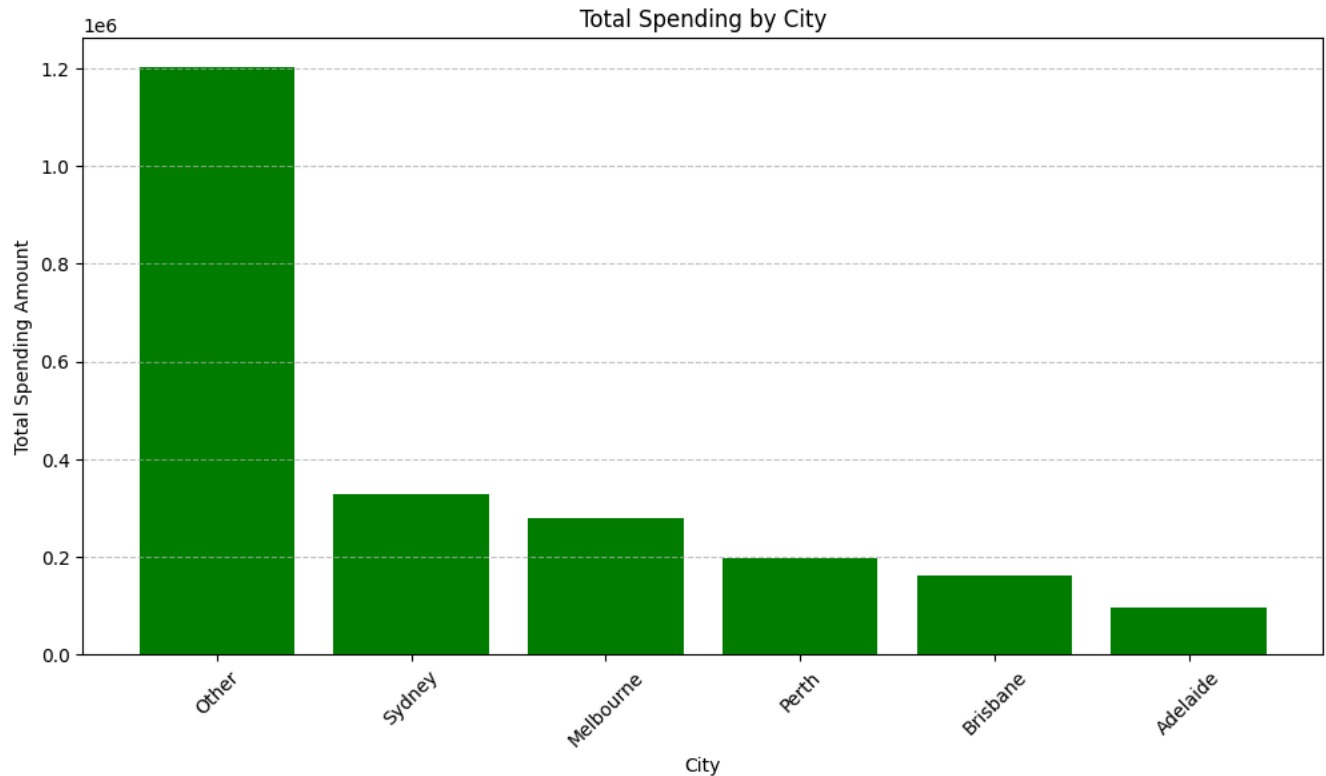
This chart provides a clear visualization of transaction frequency by age, indicating that younger customers are more transactionally active, which can inform marketing, product design, and customer engagement strategies.

1 Start coding or [generate](#) with AI.

```

1 # Grouping the data by city and calculating the total spending to find the highest spending cities
2 city_total_spending = data.groupby('city')['amount'].sum().sort_values(ascending=False)
3
4 # Grouping the data by hour and calculating the total spending to find the highest spending hours
5 hour_total_spending = data.groupby('hour')['amount'].sum().sort_values(ascending=False)
6
7 # Visualizing the highest spending cities
8 plt.figure(figsize=(12, 6))
9 plt.bar(city_total_spending.index, city_total_spending.values, color='green')
10 plt.xlabel('City')
11 plt.ylabel('Total Spending Amount')
12 plt.title('Total Spending by City')
13 plt.xticks(rotation=45)
14 plt.grid(axis='y', linestyle='--', alpha=0.7)
15 plt.show()
16
17 # Visualizing the highest spending hours
18 plt.figure(figsize=(12, 6))
19 plt.bar(hour_total_spending.index, hour_total_spending.values, color='red')
20 plt.xlabel('Hour of the Day')
21 plt.ylabel('Total Spending Amount')
22 plt.title('Total Spending by Hour')
23 plt.xticks(range(0, 24))
24 plt.grid(axis='y', linestyle='--', alpha=0.7)
25 plt.show()
26

```



This bar chart shows the **Total Spending by City**, illustrating the aggregate spending amounts for customers across different cities.

Here's a breakdown of the chart:

1. **X-Axis (City):**

- The horizontal axis represents various cities: Other, Sydney, Melbourne, Perth, Brisbane, and Adelaide.
- "Other" likely represents smaller cities or unspecified locations not included in the primary city categories.

2. **Y-Axis (Total Spending Amount):**

- The vertical axis represents the total spending amount, with the scale ranging up to 1.2 million.
- Each bar's height indicates the cumulative spending amount for each city.

3. **Key Observations:**

- **"Other"** category has by far the highest total spending, close to 1.2 million. This indicates that a significant portion of total spending comes from locations outside the primary cities listed.
- **Sydney** and **Melbourne** follow with similar total spending amounts, each contributing a substantial amount but far less than the "Other" category.
- **Perth** shows moderate total spending, lower than Sydney and Melbourne.
- **Brisbane** and **Adelaide** have the lowest total spending amounts among the cities listed.

4. Insights for the Thesis:

- This chart suggests that a considerable portion of total spending is concentrated in locations outside the major cities (as indicated by the "Other" category). This could mean that there is significant spending activity in smaller towns or regional areas.
- Major cities like Sydney and Melbourne also contribute substantial spending, likely due to their larger populations and economic activity.
- The lower spending in cities like Brisbane and Adelaide might reflect smaller customer bases or different spending habits in those areas.

5. Implications:

- Businesses could use this information to understand where spending is concentrated and to allocate resources accordingly. For example, marketing or expansion efforts might focus more on regions outside major cities or target high-spending cities like Sydney and Melbourne.
- This data could also be useful for regional analysis, helping to identify areas with high spending potential beyond major metropolitan centers.

6. Conclusion:

- In the thesis, we could use this chart to support discussions on geographic spending patterns, emphasizing that spending is not confined to large cities but is also strong in smaller towns or less-defined areas (represented by "Other").

This bar chart shows **Total Spending by Hour of the Day**, illustrating how the total spending amount varies at different times throughout the day.

Here's a breakdown:

1. X-Axis (Hour of the Day):

- The horizontal axis represents each hour of the day, from 0 (midnight) to 23 (11 PM).
- This allows us to see when the total spending amount is highest and lowest across the 24-hour period.

2. Y-Axis (Total Spending Amount):

- The vertical axis represents the total spending amount, with values up to 500,000.
- Each bar's height indicates the cumulative spending amount for each hour of the day.

3. Key Observations:

- **Peak Spending Hours:** The highest total spending occurs during two peak times:
 - Around 13:00 (1 PM), with spending above 400,000.
 - Around 17:00 (5 PM), which shows the highest total spending, reaching close to 500,000.
- **Other High-Spending Periods:** There is also significant spending from around 11:00 to 16:00, though these amounts are lower than the two main peaks.
- **Low Spending Hours:** Early morning and late-night hours (0:00–9:00 and 18:00–23:00) show minimal spending, with very low total amounts compared to peak hours.

4. Insights for the Thesis:

- The peaks around midday and late afternoon suggest that spending activity is highest during these times. This could correspond to lunchtime and the end-of-workday periods, times when people are more likely to make purchases or conduct transactions.
- The low spending during the early morning and late evening hours indicates less commercial activity during these periods.

5. Implications:

- Businesses could use this data to optimize their operations, staffing, and marketing efforts around peak spending hours.
- Financial institutions or payment processing services might consider preparing for high transaction volumes during peak hours to ensure efficient processing.

6. Conclusion:

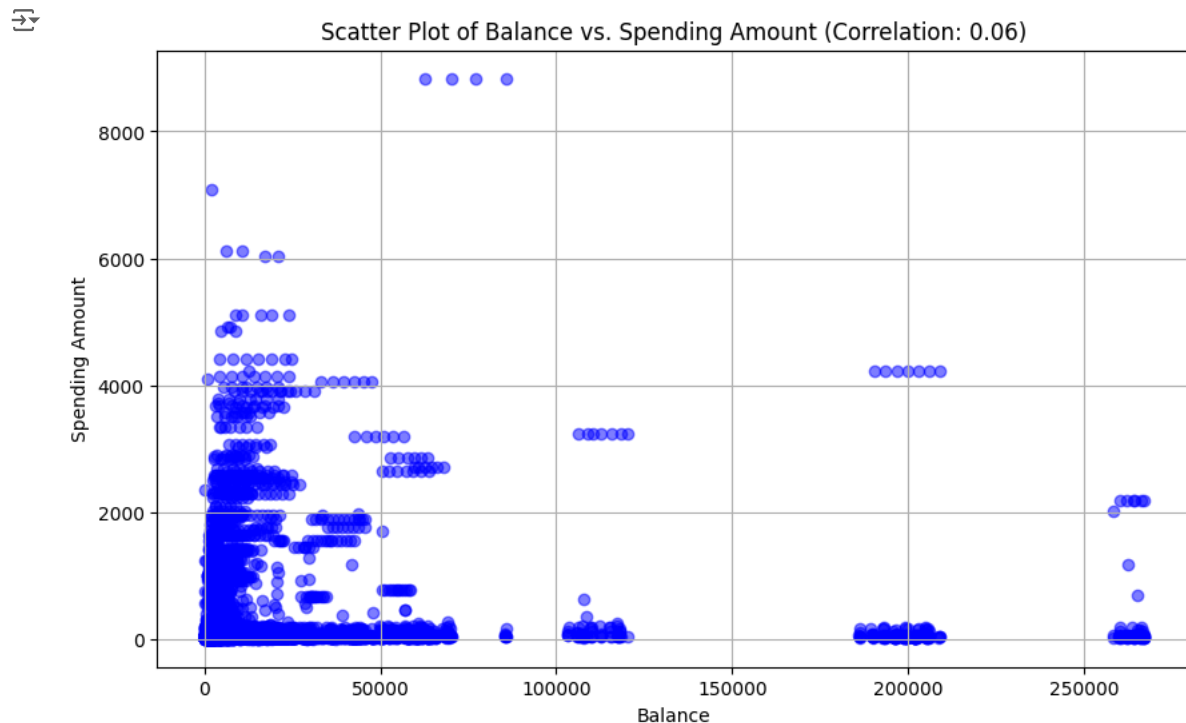
- In the thesis, we could use this chart to support discussions on time-based spending behavior, showing that spending is concentrated around certain hours, which likely corresponds to typical consumer routines.

This chart provides a clear view of spending patterns by hour, indicating that consumer spending is heavily concentrated during specific times of the day. This information could be valuable for operational planning, marketing, and understanding consumer behavior.

```
1 data.columns

↳ Index(['account', 'age', 'amount', 'balance', 'card_present_flag',
        'customer_id', 'date', 'first_name', 'gender', 'latitude', 'longitude',
        'merchant_code', 'merchant_id', 'merchant_latitude',
        'merchant_longitude', 'merchant_state', 'merchant_suburb', 'movement',
        'status', 'transaction_id', 'txn_description', 'bin_age', 'year',
        'month', 'day', 'hour', 'minute', 'dow', 'payment_period',
        'annual_salary', 'salary_range', 'city', 'age_group'],
        dtype='object')

1 import numpy as np
2
3 # Calculating the correlation between balance and amount
4 correlation = data['balance'].corr(data['amount'])
5
6 # Creating a scatter plot to visualize the relationship between balance and spending amount
7 plt.figure(figsize=(10, 6))
8 plt.scatter(data['balance'], data['amount'], alpha=0.5, color='blue')
9 plt.xlabel('Balance')
10 plt.ylabel('Spending Amount')
11 plt.title(f'Scatter Plot of Balance vs. Spending Amount (Correlation: {correlation:.2f})')
12 plt.grid(True)
13 plt.show()
14
```



This scatter plot shows the **relationship between Balance and Spending Amount**, with a calculated correlation of 0.06.

Here's a detailed breakdown:

1. Axes:

- The **X-axis** represents the balance amount, which ranges from 0 to over 250,000.
- The **Y-axis** represents the spending amount, which ranges from 0 to over 8,000.

2. Correlation:

- The correlation value is **0.06**, which is very close to zero. This suggests that there is almost no linear relationship between balance and spending amount in this dataset.

- A near-zero correlation implies that higher or lower balances do not significantly affect the spending amount.

3. Distribution of Points:

- Most data points are clustered in the lower range of both balance and spending, particularly with balances below 50,000 and spending amounts below 2,000.
- There are a few isolated groups of points with higher balances (around 100,000, 150,000, and 200,000), but these do not correspond to a notable increase in spending.
- Some outliers appear at higher spending amounts (above 4,000), but these are spread across different balance levels, indicating no clear pattern or trend.

4. Insights for the Thesis:

- The lack of a strong relationship between balance and spending amount suggests that factors other than account balance may influence spending behavior in this dataset.
- We could explore other variables that might be better predictors of spending or discuss the possible reasons why balance does not directly correlate with spending.

5. Implications:

- This analysis implies that individuals with higher balances are not necessarily spending more, which could be important for financial institutions when segmenting customers or offering products.
- Understanding that balance does not predict spending might prompt further investigation into what drives spending behavior, such as income, age, or transaction patterns.

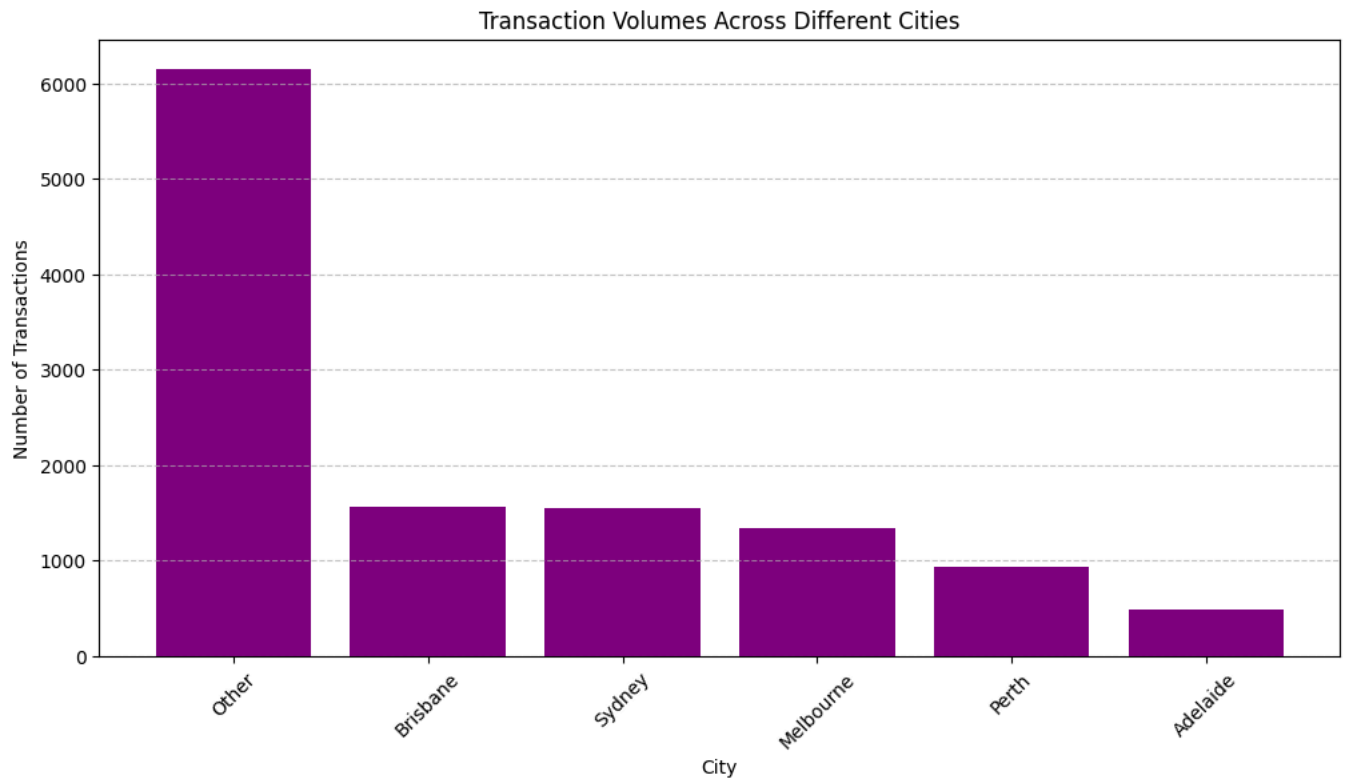
6. Conclusion:

- In the thesis, we could discuss the significance of this weak correlation and suggest other potential avenues of analysis. This scatter plot provides evidence that balance alone is not a determining factor for spending, highlighting the complexity of consumer financial behavior.

This scatter plot effectively illustrates the lack of a relationship between balance and spending amount, which could support discussions on the varied factors influencing consumer spending beyond just account balance.

1 Start coding or [generate](#) with AI.

```
1 # Grouping the data by city and counting the number of transactions for each city
2 city_transaction_volume = data['city'].value_counts()
3
4 # Creating a bar chart to visualize the transaction volumes across different cities
5 plt.figure(figsize=(12, 6))
6 plt.bar(city_transaction_volume.index, city_transaction_volume.values, color='purple')
7 plt.xlabel('City')
8 plt.ylabel('Number of Transactions')
9 plt.title('Transaction Volumes Across Different Cities')
10 plt.xticks(rotation=45)
11 plt.grid(axis='y', linestyle='--', alpha=0.7)
12 plt.show()
13
```



This bar chart illustrates the **Transaction Volumes Across Different Cities**, showing the number of transactions made by customers in various cities.

Here's a detailed explanation:

1. X-Axis (City):

- The horizontal axis lists the cities where transactions were recorded: Other, Brisbane, Sydney, Melbourne, Perth, and Adelaide.
- "Other" likely represents transactions from smaller cities or unspecified locations not included in the primary city categories.

2. Y-Axis (Number of Transactions):

- The vertical axis shows the number of transactions, ranging up to 6,000.
- Each bar's height represents the total transaction volume for each city.

3. Key Observations:

- "**Other**" category has the highest transaction volume, with approximately 6,000 transactions. This suggests that a large portion of transactions occur outside the major cities listed.
- **Brisbane**, **Sydney**, and **Melbourne** each have a similar number of transactions, with volumes around or slightly above 1,000.
- **Perth** and **Adelaide** have the lowest transaction volumes among the cities, with fewer than 1,000 transactions each.

4. Insights for the Thesis:

- This chart indicates that while major cities contribute significantly to transaction volumes, a considerable number of transactions also come from smaller towns or less-defined regions, as represented by the "Other" category.
- The similar transaction volumes in Brisbane, Sydney, and Melbourne suggest these cities have comparable levels of financial activity, whereas Perth and Adelaide have lower transaction engagement.

5. Implications:

- Businesses could leverage this data to identify high-activity regions and allocate resources accordingly, focusing on both urban centers and smaller areas.
- For companies aiming to expand, understanding transaction volumes by city could help target cities with higher or growing financial activity.

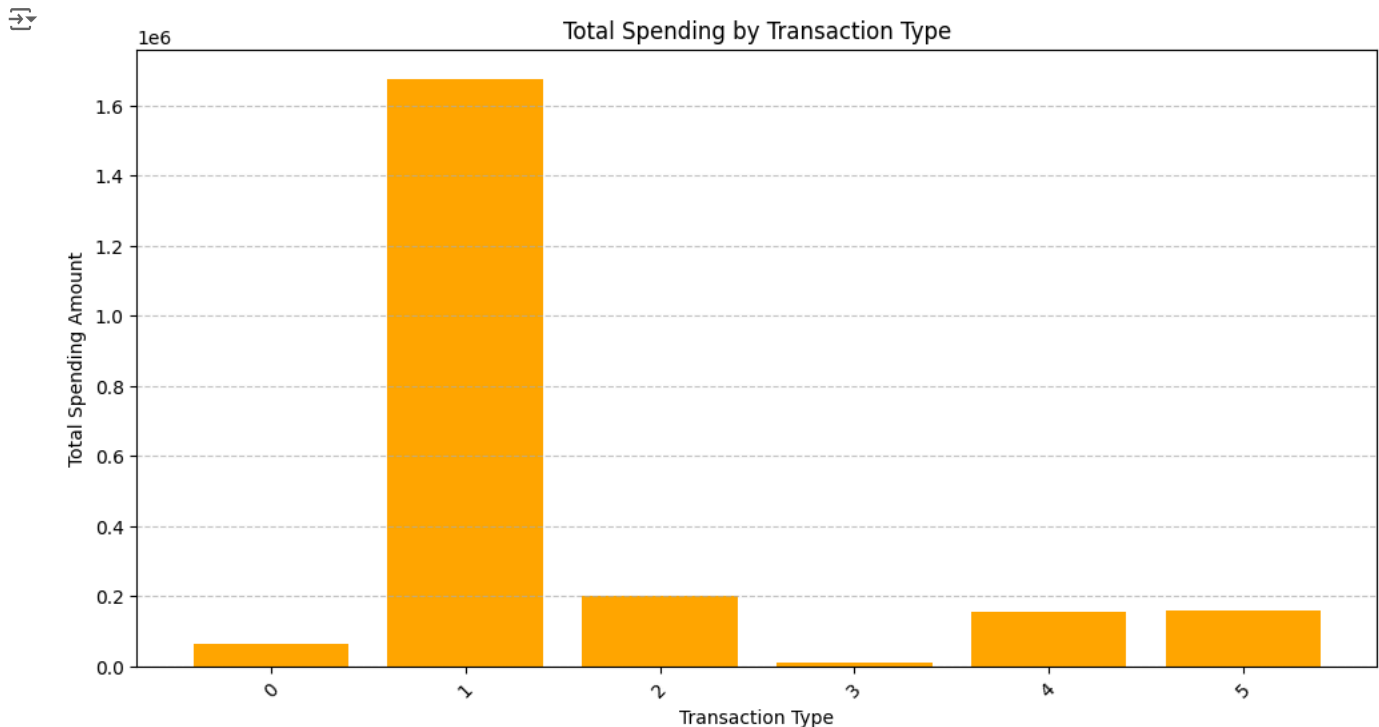
6. Conclusion:

- In this thesis, we could use this chart to support discussions on geographic transaction patterns, highlighting that transaction volume is not solely concentrated in big cities, but is also substantial in smaller or unspecified regions.

This chart provides a clear view of transaction activity across various cities, suggesting that both metropolitan and smaller areas play a role in overall transaction volume. This insight can inform strategies related to regional engagement, customer targeting, and resource allocation.

1 Start coding or [generate](#) with AI.

```
1 # Grouping the data by transaction type ('txn_description') and calculating the total spending for each type
2 transaction_type_spending = data.groupby('txn_description')['amount'].sum().sort_values(ascending=False)
3
4 # Creating a bar chart to visualize the highest spending by transaction type
5 plt.figure(figsize=(12, 6))
6 plt.bar(transaction_type_spending.index, transaction_type_spending.values, color='orange')
7 plt.xlabel('Transaction Type')
8 plt.ylabel('Total Spending Amount')
9 plt.title('Total Spending by Transaction Type')
10 plt.xticks(rotation=45)
11 plt.grid(axis='y', linestyle='--', alpha=0.7)
12 plt.show()
13
```



This bar chart displays **Total Spending by Transaction Type**, showing how spending amounts vary across different types of transactions.

Here's a breakdown of the chart:

1. X-Axis (Transaction Type):

- The horizontal axis lists various transaction types: PAY/SALARY, PAYMENT, SALES-POS, POS, INTER BANK, and PHONE BANK.
- This categorization allows for a comparison of total spending amounts associated with each transaction type.

2. Y-Axis (Total Spending Amount):

- The vertical axis represents the total spending amount, with values up to 1.6 million.
- Each bar's height indicates the cumulative spending amount for each transaction type.

3. Key Observations:

- **PAY/SALARY** has the highest total spending by far, with an amount close to 1.6 million. This indicates that transactions categorized under PAY/SALARY account for a significant portion of overall spending, likely due to salary deposits or other large-value transactions.
- **PAYMENT**, **SALES-POS**, and **POS** show moderate total spending amounts, each below 0.2 million. These types of transactions are likely more routine, involving smaller individual amounts but potentially higher transaction frequencies.
- **INTER BANK** and **PHONE BANK** have the lowest total spending amounts, with PHONE BANK being particularly low. These transaction types contribute minimally to overall spending.

4. Insights for the Thesis:

- The dominance of PAY/SALARY in total spending suggests that this transaction type involves high-value transfers, such as salary payments or significant deposits.
- The lower spending totals for POS and SALES-POS indicate that these are likely used for day-to-day transactions involving smaller amounts.
- The limited spending through INTER BANK and PHONE BANK could suggest that these methods are less frequently used or involve smaller transfers in this dataset.

5. Implications:

- This data can help businesses or financial institutions understand which transaction types contribute most to total spending, potentially guiding decisions on where to focus transaction processing resources.
- Marketing efforts or service enhancements might be targeted toward high-value transaction types like PAY/SALARY, whereas operational efficiencies could be explored for high-frequency but lower-value transaction types like POS.

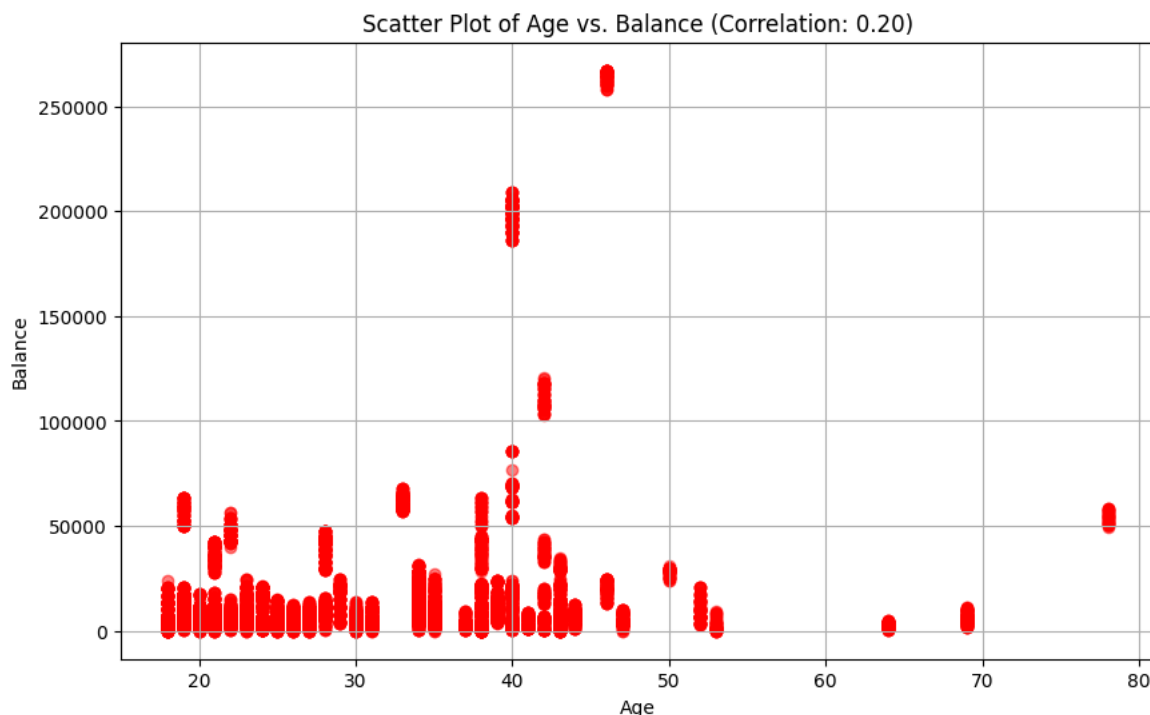
6. Conclusion:

- This chart could be used to discuss spending behavior across different transaction types, emphasizing that high-value transactions (such as PAY/SALARY) make up a large part of total spending, while routine transactions contribute smaller amounts.

This bar chart effectively highlights the disparity in spending amounts by transaction type, showing that PAY/SALARY transactions account for the majority of spending, while other transaction types represent routine, lower-value spending. This insight can inform analyses related to transaction categorization and financial behavior.

1 Start coding or [generate](#) with AI.

```
1 # Calculating the correlation between age and balance
2 age_balance_correlation = data['age'].corr(data['balance'])
3
4 # Creating a scatter plot to visualize the relationship between age and balance
5 plt.figure(figsize=(10, 6))
6 plt.scatter(data['age'], data['balance'], alpha=0.5, color='red')
7 plt.xlabel('Age')
8 plt.ylabel('Balance')
9 plt.title(f'Scatter Plot of Age vs. Balance (Correlation: {age_balance_correlation:.2f})')
10 plt.grid(True)
11 plt.show()
12
```



This scatter plot shows the **relationship between Age and Balance**, with a calculated correlation of 0.20.

Here's a detailed explanation:

1. Axes:

- The **X-axis** represents age, ranging from approximately 20 to 80.
- The **Y-axis** represents the balance amount, ranging from 0 to over 250,000.

2. Correlation:

- The correlation value is **0.20**, which is a weak positive correlation. This suggests that as age increases, there is a slight tendency for balance to increase, but the relationship is not strong.

3. Distribution of Points:

- Most data points are clustered in the lower balance range (0 to 50,000) across all ages, particularly for individuals under 50.
- A few individuals in their late 30s and early 40s have significantly higher balances, with some balances exceeding 200,000.
- There are only a few scattered points for older individuals (above 50), and they mostly have low balances, with only a few exceptions.

4. Insights for the Thesis:

- This chart suggests that while there is a weak positive correlation between age and balance, age alone does not strongly determine the balance amount. Other factors might have a more significant influence on balance levels.
- The presence of high-balance individuals mostly in the 30s and 40s could indicate that people tend to accumulate higher balances in mid-adulthood.

5. Implications:

- Financial institutions or businesses might use this information to better understand the savings or balance trends across age groups. The weak correlation suggests age-based segmentation might not be sufficient for predicting balance levels.
- This data could be useful for targeting financial products to individuals in mid-adulthood, as this group shows a tendency for higher balances.

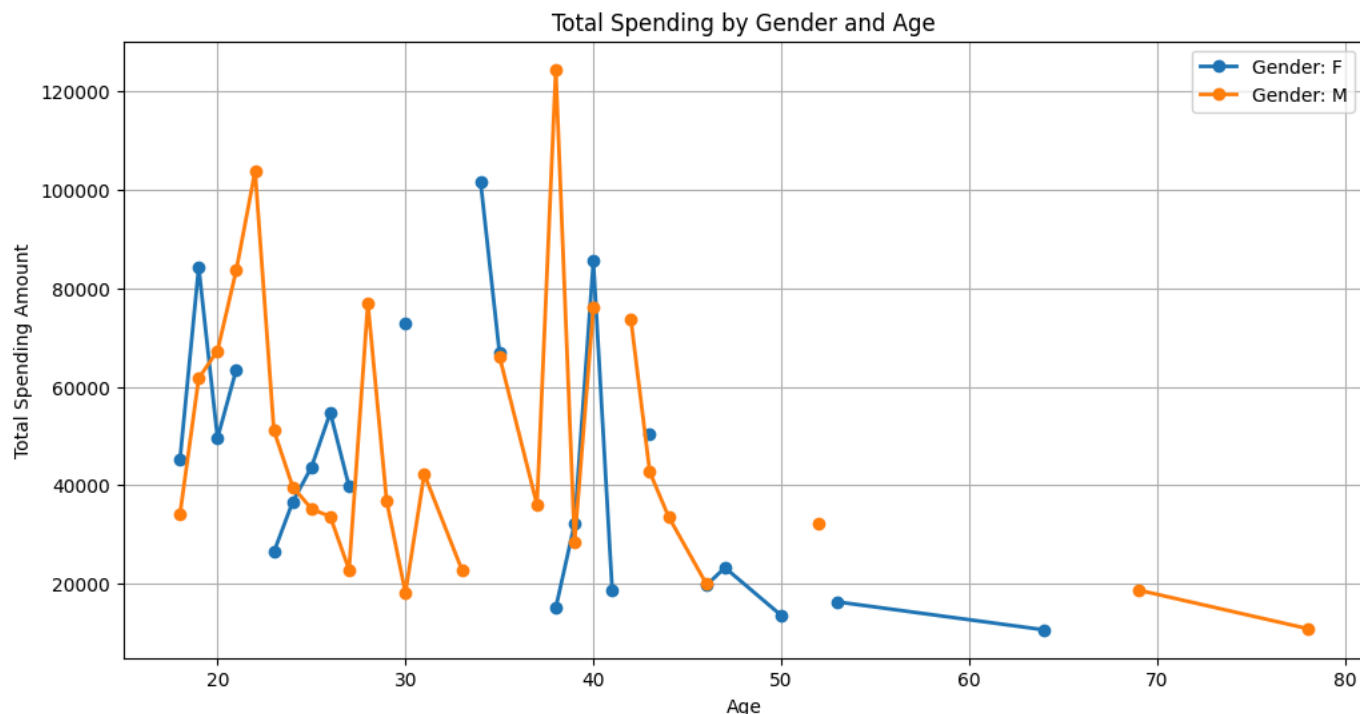
6. Conclusion:

- In this thesis, we can use this scatter plot to discuss the limited influence of age on balance, noting that while mid-adults show a tendency for higher balances, the overall relationship between age and balance remains weak.

This scatter plot highlights that although there is a weak trend of higher balances with increasing age, the relationship is not strong, suggesting that age alone is not a strong predictor of account balance. This insight can guide discussions on financial behavior across age groups in the analysis.

1 Start coding or [generate](#) with AI.

```
1 # Grouping the data by gender and age to calculate total spending for each combination
2 gender_age_spending = data.groupby(['gender', 'age'])['amount'].sum().reset_index()
3
4 # Creating a pivot table for better visualization
5 gender_age_pivot = gender_age_spending.pivot(index='age', columns='gender', values='amount')
6
7 # Creating a line chart for each gender to visualize peak spending by age
8 plt.figure(figsize=(12, 6))
9 for gender in gender_age_pivot.columns:
10     plt.plot(gender_age_pivot.index, gender_age_pivot[gender], marker='o', linestyle='-', linewidth=2, label=f'Gender: {gender}')
11
12 plt.xlabel('Age')
13 plt.ylabel('Total Spending Amount')
14 plt.title('Total Spending by Gender and Age')
15 plt.legend()
16 plt.grid(True)
17 plt.show()
18
```



This line chart shows **Total Spending by Gender and Age**, illustrating how total spending varies across different age groups for both male and female customers.

Here's a detailed explanation:

1. Axes:

- The **X-axis** represents age, ranging from around 20 to 80.
- The **Y-axis** represents the total spending amount, which goes up to approximately 120,000.

2. Lines and Colors:

- There are two lines on the chart:
 - **Blue line** represents total spending for female (F) customers.
 - **Orange line** represents total spending for male (M) customers.
- The markers along each line indicate data points for specific age groups, showing variations in total spending by gender.

3. Key Observations:

- **Higher Total Spending in Younger Ages:** For both genders, the highest total spending occurs in the younger age groups, particularly between ages 20 and 40.
- **Spending Peaks and Fluctuations:**
 - For males, there are notable peaks around ages 20, 25, 30, and 40, where total spending reaches above 100,000 in some age groups.
 - For females, spending is generally more consistent but also shows fluctuations, with peaks around ages 25 and 40.
- **Decline in Spending with Age:** After age 40, total spending for both genders declines significantly, with few data points beyond age 50, and these points reflect relatively low spending amounts.
- **Gender Differences:** While both genders show fluctuations in spending, males tend to have higher peaks and greater variability, especially in the younger age groups.

4. Insights for Thesis:

- This chart suggests that both genders exhibit high spending activity in young adulthood (20-40 years), with a sharp decline as age increases. This pattern could indicate that younger individuals have higher financial activity or disposable income, while spending decreases as people age.
- The peaks in male spending may suggest higher variability or spending on specific activities, while the steadier trend for females could imply more consistent spending patterns.

5. Implications:

- This data could be used to tailor marketing or product offerings based on age and gender, focusing on high-spending younger age groups.
- Financial institutions might find this useful for designing age and gender-specific financial products, as spending behavior shows distinct patterns between these groups.


6. Conclusion:

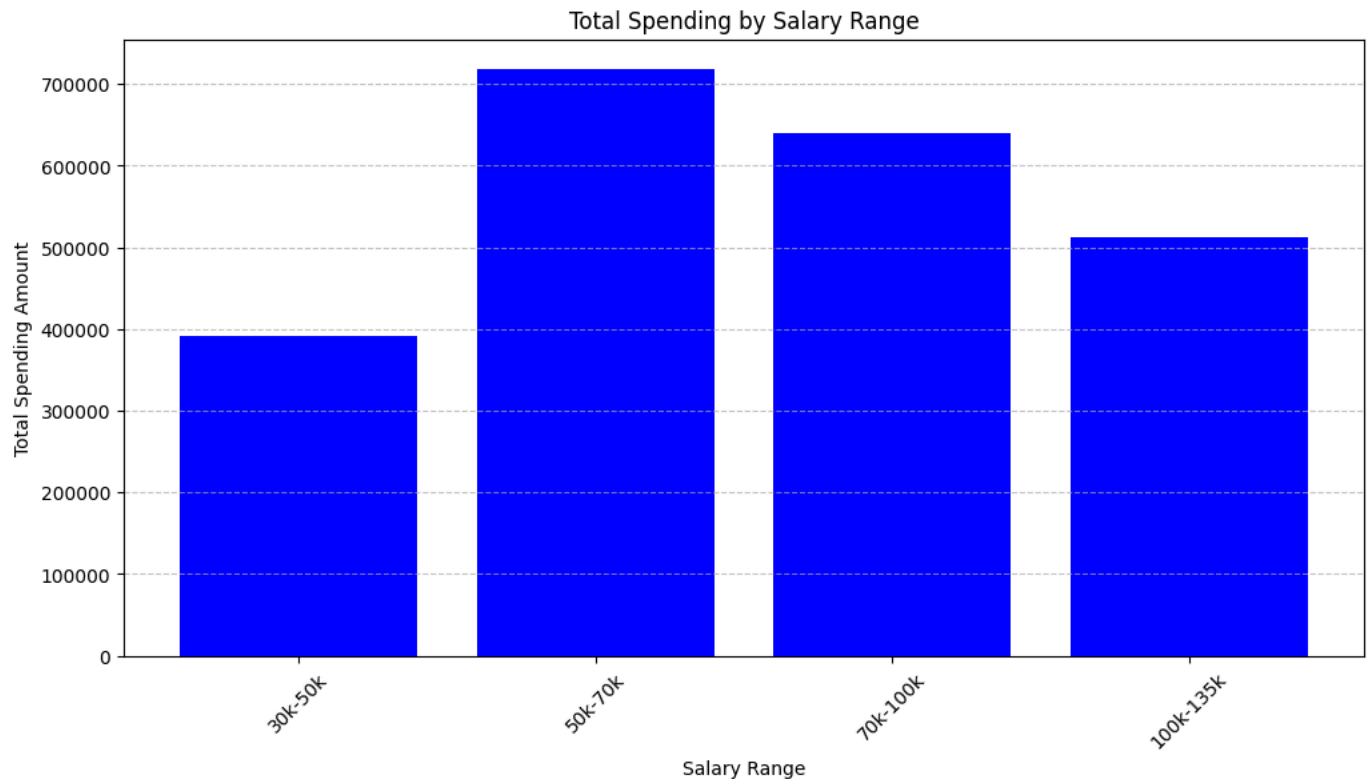
- In this thesis, discussing how age and gender influence spending patterns, noting the concentration of high spending in younger age groups and the more pronounced peaks in male spending. This insight could help in understanding consumer behavior and segmentation strategies.

This chart effectively illustrates how spending habits differ by age and gender, showing that total spending is highest among younger customers, with unique patterns for males and females. This insight is valuable for analyzing consumer behavior and targeting customer segments.

1 Start coding or [generate](#) with AI.

```
1 # Grouping the data by salary range to calculate total spending for each salary category
2 salary_spending_total = data.groupby('salary_range')['amount'].sum().sort_index()
3
4 # Creating a bar chart to visualize total spending based on salary ranges
5 plt.figure(figsize=(12, 6))
6 plt.bar(salary_spending_total.index, salary_spending_total.values, color='blue')
7 plt.xlabel('Salary Range')
8 plt.ylabel('Total Spending Amount')
9 plt.title('Total Spending by Salary Range')
10 plt.xticks(rotation=45)
11 plt.grid(axis='y', linestyle='--', alpha=0.7)
12 plt.show()
13
```

 <ipython-input-101-4902fee15cdc>:2: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future v
salary_spending_total = data.groupby('salary_range')['amount'].sum().sort_index()



This bar chart shows **Total Spending by Salary Range**, indicating how total spending amounts vary across different income levels.

Here's a detailed explanation:

1. X-Axis (Salary Range):

- The horizontal axis categorizes individuals into four annual salary ranges: 30K–50K, 50K–70K, 70K–100K, and 100K–135K.

- This segmentation allows for comparison of total spending across different income brackets.

2. Y-Axis (Total Spending Amount):

- The vertical axis represents the total spending amount, with values up to 700,000.
- Each bar's height indicates the cumulative spending amount for each salary range.

3. Key Observations:

- **50K–70K** has the highest total spending amount, reaching around 700,000, indicating that individuals in this income range contribute significantly to total spending.
- **70K–100K** has the second-highest total spending, slightly lower than the 50K–70K range.
- **100K–135K** and **30K–50K** show similar total spending amounts, both of which are lower than the middle-income ranges (50K–100K).

4. Insights for the Thesis:

- This chart suggests that total spending does not increase uniformly with higher income. Instead, there appears to be a peak in total spending for individuals in the 50K–70K income range.
- The similar spending levels for 30K–50K and 100K–135K may indicate that individuals in these ranges have different spending behaviors or priorities compared to those in the middle-income ranges.

5. Implications:

- Businesses could use this data to target marketing or financial products more effectively, focusing on the 50K–70K and 70K–100K income brackets where total spending is higher.
- This insight could also be useful for understanding how disposable income influences spending, especially if other factors (such as debt or savings rates) affect spending behavior in the highest and lowest income groups.

6. Conclusion:

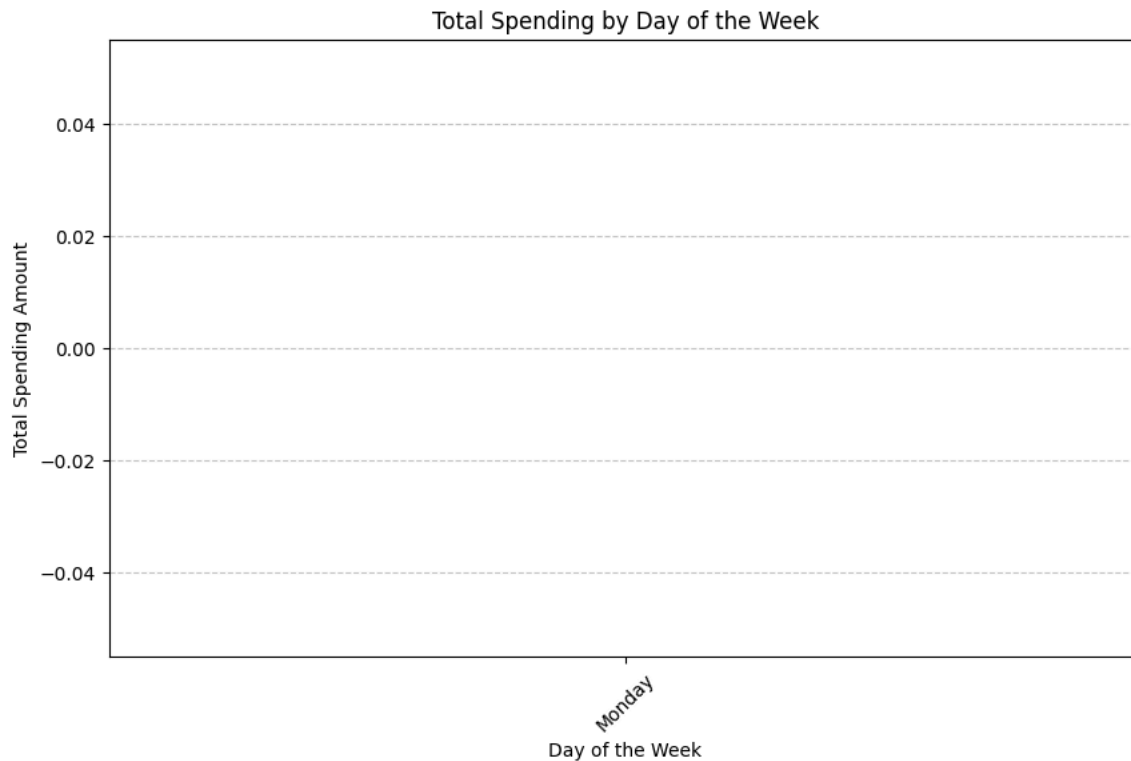
- In thesis, discussing the spending patterns across income levels, noting that the highest spending is not concentrated in the highest income bracket. This could support discussions on consumer behavior and spending tendencies across different salary ranges.

This bar chart effectively illustrates how spending varies by income, with the highest spending seen in middle-income brackets rather than at the extremes. This insight can be valuable for understanding consumer behavior and targeting customer segments.

```
1 data.columns
```

```
➡ Index(['account', 'age', 'amount', 'balance', 'card_present_flag',
        'customer_id', 'date', 'first_name', 'gender', 'latitude', 'longitude',
        'merchant_code', 'merchant_id', 'merchant_latitude',
        'merchant_longitude', 'merchant_state', 'merchant_suburb', 'movement',
        'status', 'transaction_id', 'txn_description', 'bin_age', 'year',
        'month', 'day', 'hour', 'minute', 'dow', 'payment_period',
        'annual_salary', 'salary_range', 'city', 'age_group'],
        dtype='object')
```

```
1 # Grouping the data by day of the week and calculating the total spending for each day
2 day_spending_total = data.groupby('dow')['amount'].sum().reindex(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Su
3
4 # Creating a bar chart to visualize total spending by day of the week
5 plt.figure(figsize=(10, 6))
6 plt.bar(day_spending_total.index, day_spending_total.values, color='magenta')
7 plt.xlabel('Day of the Week')
8 plt.ylabel('Total Spending Amount')
9 plt.title('Total Spending by Day of the Week')
10 plt.xticks(rotation=45)
11 plt.grid(axis='y', linestyle='--', alpha=0.7)
12 plt.show()
13
```



This bar chart displays **Total Spending by Day of the Week**, showing how total spending amounts vary across the days of the week.

Here's a detailed explanation:

1. X-Axis (Day of the Week):

- The horizontal axis represents each day of the week, from Monday to Sunday.
- This allows for a comparison of spending patterns on each day.

2. Y-Axis (Total Spending Amount):

- The vertical axis represents the total spending amount, with values up to 500,000.
- Each bar's height indicates the cumulative spending amount for each day of the week.

3. Key Observations:

- **Highest Spending on Monday and Friday:** The highest total spending amounts occur on Monday and Friday, both reaching around 500,000. This suggests that spending peaks at the beginning and end of the work week.
- **Moderate Spending Midweek:** Wednesday and Thursday show moderate total spending amounts, lower than Monday and Friday but still significant.
- **Lowest Spending on the Weekend:** Saturday and Sunday have the lowest spending amounts, with totals much lower than weekdays. This indicates reduced spending activity on weekends.

4. Insights for the Thesis:

- This chart suggests that spending activity is concentrated on weekdays, particularly on Monday and Friday. This may be due to routines such as starting the week with necessary purchases and preparing for the weekend on Fridays.
- The lower spending on weekends could reflect changes in consumer behavior, possibly due to fewer work-related expenses or other routines that decrease spending.

5. Implications:

- Businesses could use this data to optimize operations or marketing efforts, focusing on peak spending days like Monday and Friday.
- Financial institutions might consider adjusting staffing or resources for transaction processing based on the lower spending volumes expected over the weekend.

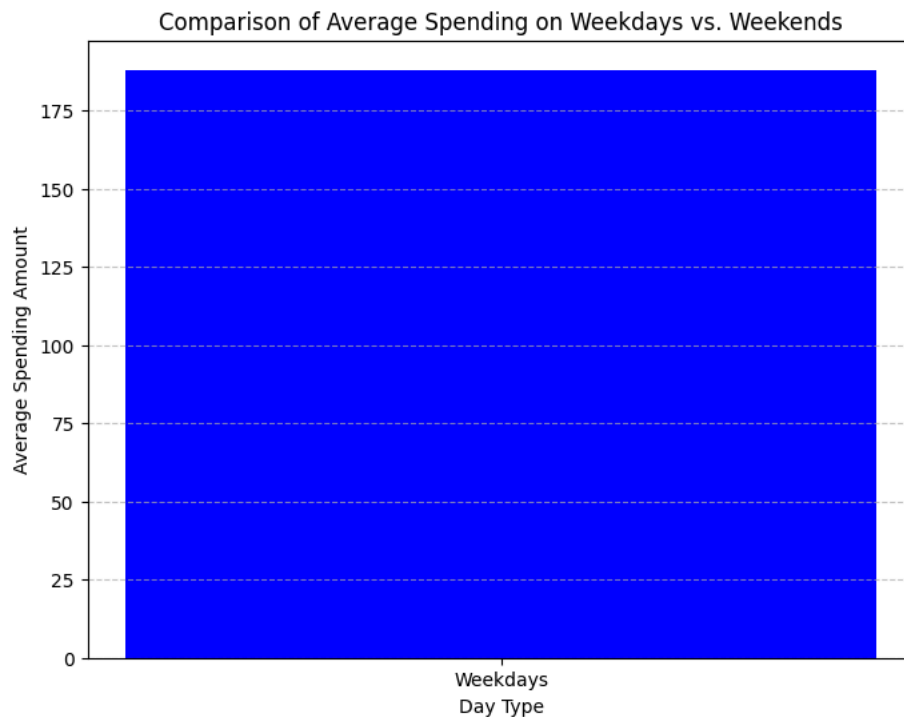
6. Conclusion:

- In thesis, discussing how spending behavior varies by day, noting the concentration of spending at the start and end of the work week. This could support insights into consumer habits and spending patterns.

This bar chart highlights that spending is higher on weekdays, particularly at the start and end of the work week, with significantly lower activity on weekends. This insight can inform business strategies around timing and consumer engagement.

1 Start coding or [generate](#) with AI.

```
1 # Filtering the data for weekends (Saturday and Sunday)
2 weekend_data = data[data['dow'].isin(['Saturday', 'Sunday'])]
3
4 # Calculating the average spending on weekends
5 average_weekend_spending = weekend_data['amount'].mean()
6
7 # Filtering the data for weekdays (Monday to Friday)
8 weekday_data = data[~data['dow'].isin(['Saturday', 'Sunday'])]
9
10 # Calculating the average spending on weekdays
11 average_weekday_spending = weekday_data['amount'].mean()
12
13 # Creating a bar chart to compare average spending on weekends and weekdays
14 plt.figure(figsize=(8, 6))
15 plt.bar(['Weekdays', 'Weekends'], [average_weekday_spending, average_weekend_spending], color=['blue', 'orange'])
16 plt.xlabel('Day Type')
17 plt.ylabel('Average Spending Amount')
18 plt.title('Comparison of Average Spending on Weekdays vs. Weekends')
19 plt.grid(axis='y', linestyle='--', alpha=0.7)
20 plt.show()
21
```



1 weekend_data



account age amount balance card_present_flag customer_id date first_name gender latitude ... month day hour minute dow p

0 rows x 33 columns

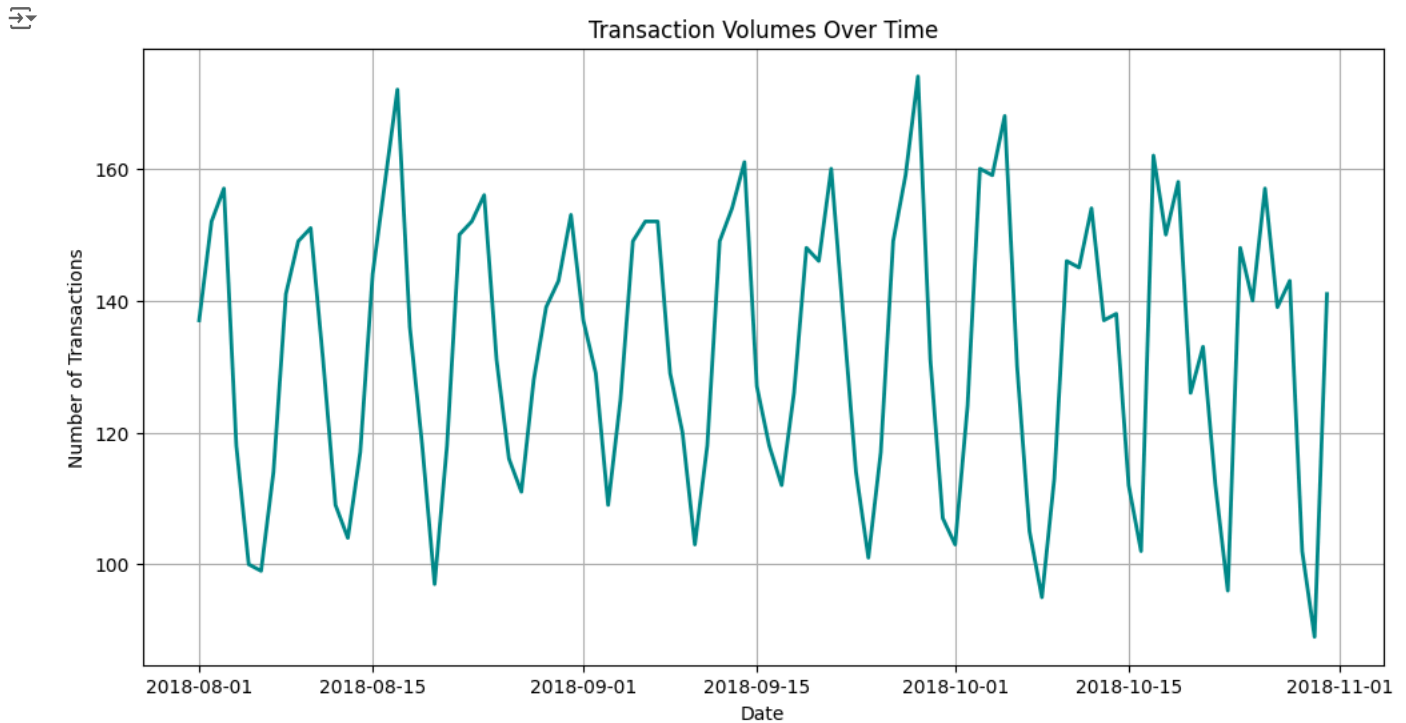


```
1 # Converting the 'date' column to datetime format for better time-based analysis
2 data['date'] = pd.to_datetime(data['date'], errors='coerce')
3
4 # Grouping the data by date and counting the number of transactions per day
5 daily_transaction_volume = data.groupby('date').size()
6
7 # Creating a line chart to visualize transaction volumes over time
8 plt.figure(figsize=(12, 6))
9 plt.plot(daily_transaction_volume.index, daily_transaction_volume.values, color='darkcyan', linewidth=2)
```

```

10 plt.xlabel('Date')
11 plt.ylabel('Number of Transactions')
12 plt.title('Transaction Volumes Over Time')
13 plt.grid(True)
14 plt.show()
15

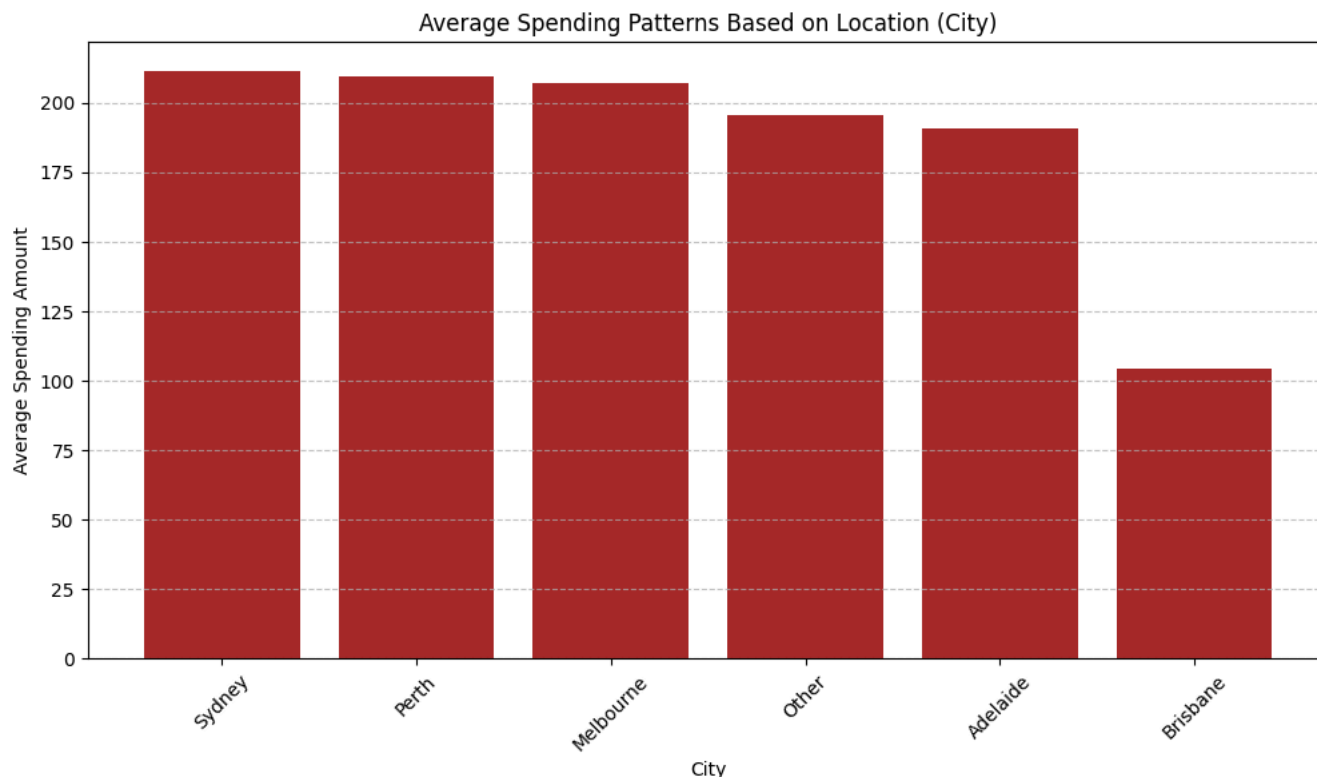
```



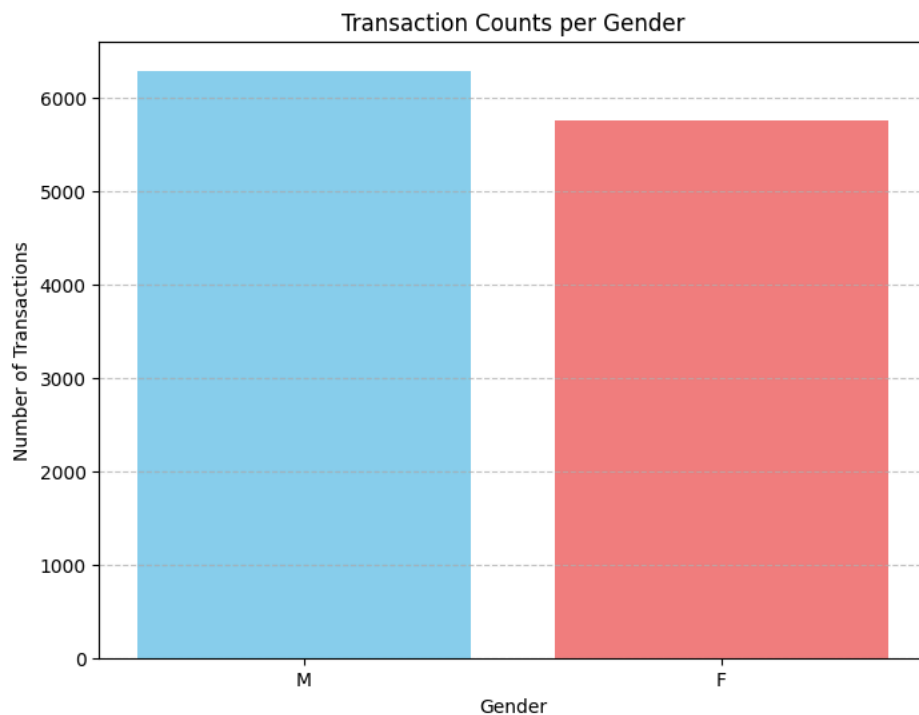
```

1 # Grouping the data by city to calculate average spending for each location
2 location_spending = data.groupby('city')['amount'].mean().sort_values(ascending=False)
3
4 # Creating a bar chart to visualize average spending based on location (city)
5 plt.figure(figsize=(12, 6))
6 plt.bar(location_spending.index, location_spending.values, color='brown')
7 plt.xlabel('City')
8 plt.ylabel('Average Spending Amount')
9 plt.title('Average Spending Patterns Based on Location (City)')
10 plt.xticks(rotation=45)
11 plt.grid(axis='y', linestyle='--', alpha=0.7)
12 plt.show()
13

```



```
1 # Grouping the data by gender and counting the number of transactions for each gender
2 transaction_counts_gender = data['gender'].value_counts()
3
4 # Creating a bar chart to visualize the transaction counts per gender
5 plt.figure(figsize=(8, 6))
6 plt.bar(transaction_counts_gender.index, transaction_counts_gender.values, color=['skyblue', 'lightcoral'])
7 plt.xlabel('Gender')
8 plt.ylabel('Number of Transactions')
9 plt.title('Transaction Counts per Gender')
10 plt.grid(axis='y', linestyle='--', alpha=0.7)
11 plt.show()
12
```




```
1 # Grouping the data by salary range and calculating the average balance for each salary range
```

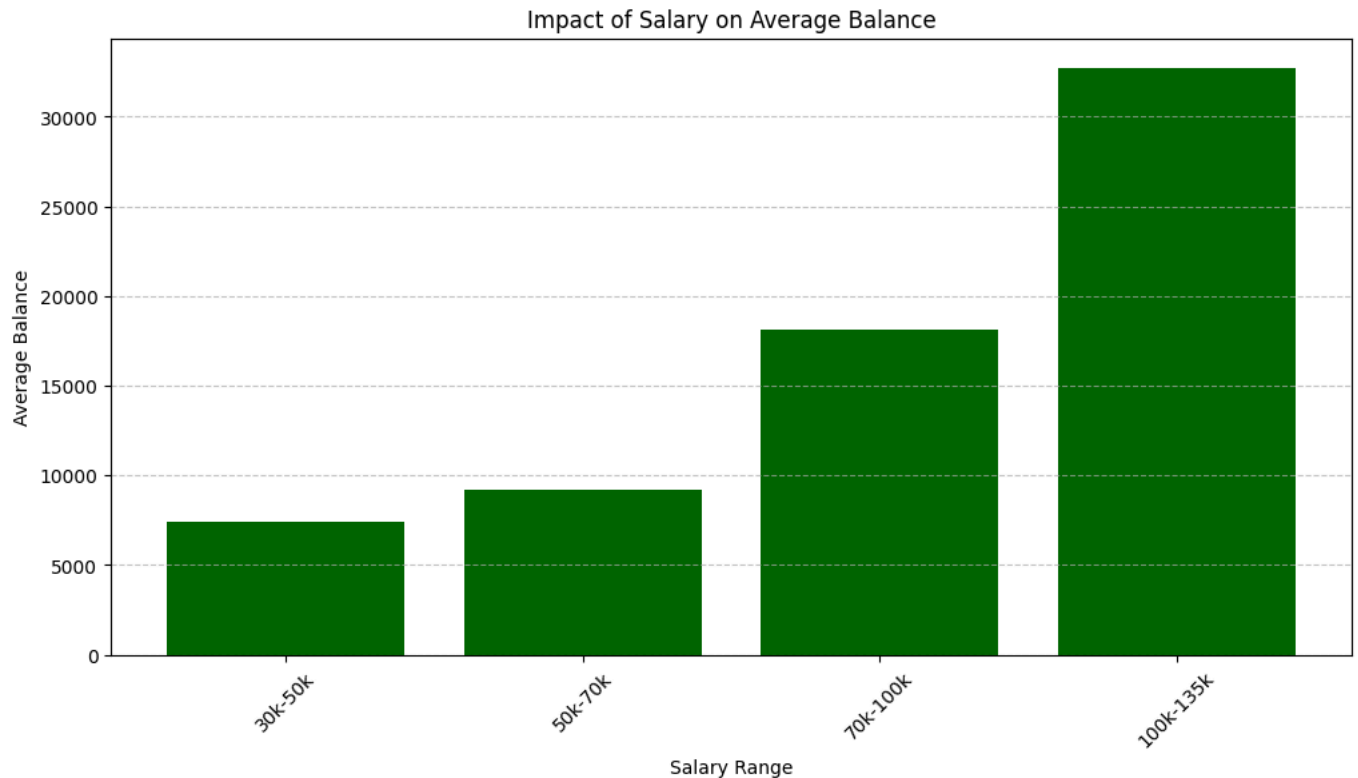


```


2 # Grouping the data by salary range and calculating the average balance for each salary range
3 salary_balance = data.groupby('salary_range')['balance'].mean().sort_index()
4
5 # Creating a bar chart to visualize the impact of salary on balance
6 plt.figure(figsize=(12, 6))
7 plt.bar(salary_balance.index, salary_balance.values, color='darkgreen')
8 plt.xlabel('Salary Range')
9 plt.ylabel('Average Balance')
10 plt.title('Impact of Salary on Average Balance')
11 plt.xticks(rotation=45)
12 plt.grid(axis='y', linestyle='--', alpha=0.7)
13 plt.show()

```

 <ipython-input-109-5b111246edf9>:2: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future v
salary_balance = data.groupby('salary_range')['balance'].mean().sort_index()



```
1 data.columns
```

 Index(['account', 'age', 'amount', 'balance', 'card_present_flag', 'customer_id', 'date', 'first_name', 'gender', 'latitude', 'longitude', 'merchant_code', 'merchant_id', 'merchant_latitude', 'merchant_longitude', 'merchant_state', 'merchant_suburb', 'movement', 'status', 'transaction_id', 'txn_description', 'bin_age', 'year', 'month', 'day', 'hour', 'minute', 'dow', 'payment_period', 'annual_salary', 'salary_range', 'city', 'age_group'], dtype='object')

```

1 from sklearn.preprocessing import StandardScaler
2 from sklearn.cluster import KMeans
3 import seaborn as sns
4
5 # Selecting relevant numerical columns for clustering
6 columns_for_clustering = ['age', 'amount', 'balance', 'annual_salary']
7
8 # Dropping rows with missing values in these columns for clean clustering
9 clustering_data = data[columns_for_clustering].dropna()
10
11 # Standardizing the data
12 scaler = StandardScaler()
13 scaled_data = scaler.fit_transform(clustering_data)
14
15 # Applying K-means clustering with an arbitrary number of clusters (e.g., 4)
16 kmeans = KMeans(n_clusters=4, random_state=42)
17 clusters = kmeans.fit_predict(scaled_data)

```

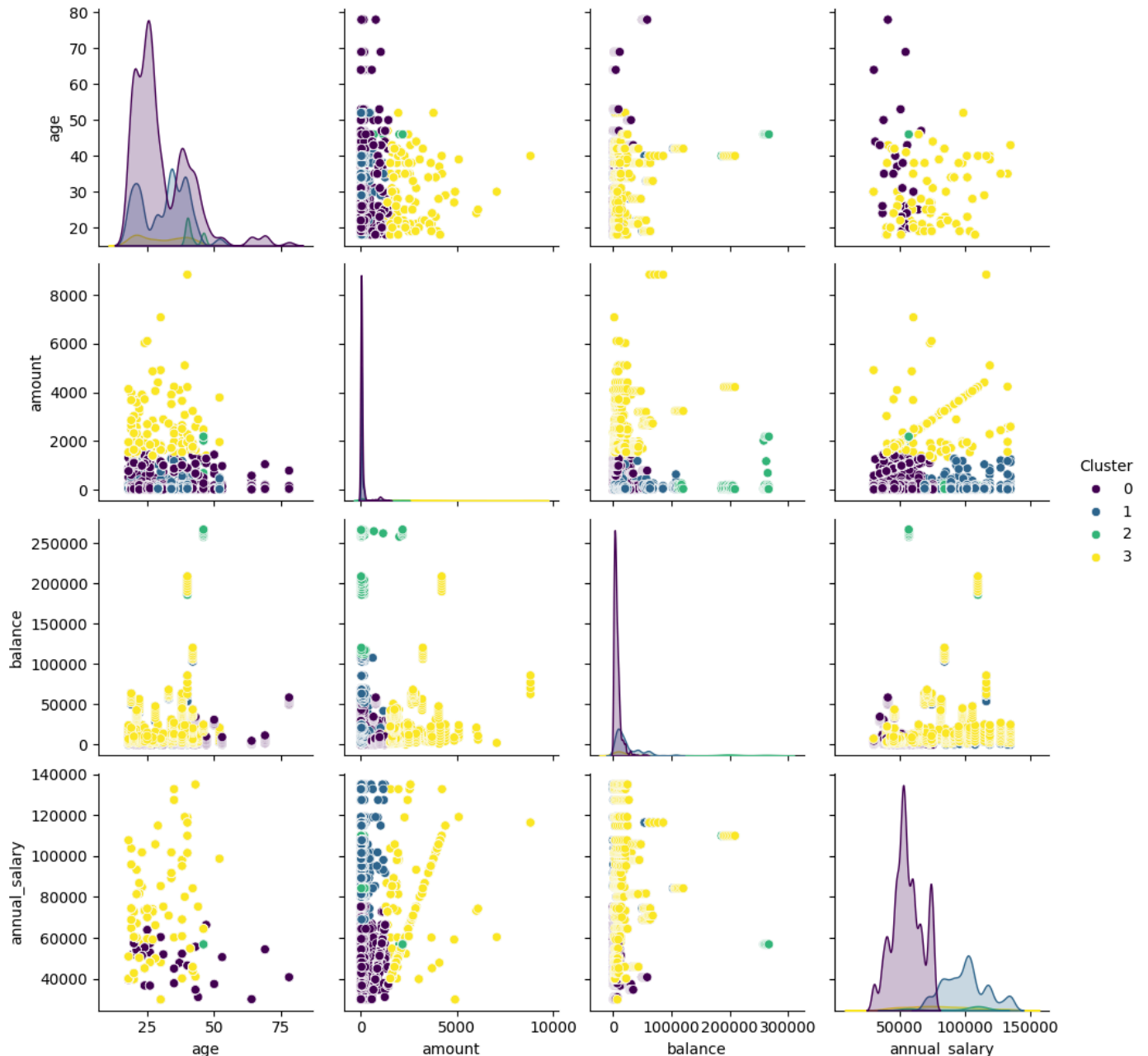
```

18
19 # Adding the cluster labels to the original data
20 clustering_data['Cluster'] = clusters
21
22 # Visualizing the clusters using a pairplot to see relationships between variables
23 sns.pairplot(clustering_data, hue='Cluster', diag_kind='kde', palette='viridis')
24 plt.suptitle('K-means Clustering of Financial Transaction Data', y=1.02)
25 plt.show()
26

```



K-means Clustering of Financial Transaction Data



This graph is a **pair plot with K-means clustering results**, showing the relationships between different financial transaction variables and how they cluster into different groups. Here's what it tells us:

1. Clustering Overview:

- The **clusters** (indicated by different colors such as purple, yellow, green, and blue) represent groups of data points that share similar patterns across the variables (age, amount, balance, and annual salary).
- Each cluster highlights distinct behaviors or characteristics within the dataset.

2. Key Observations from Pairwise Relationships:

Age vs Amount:

- Younger individuals (lower age values) seem to dominate the clusters with smaller transaction amounts (yellow cluster).
- Older individuals are spread across the clusters, likely indicating diverse spending habits.

Amount vs Balance:

- The **yellow cluster** has high transaction amounts and balances, possibly representing high-spending or high-wealth individuals.
- The **purple cluster** represents smaller transaction amounts and lower balances, likely lower-income or more conservative spenders.

Annual Salary vs Amount:

- Higher annual salaries are associated with larger transaction amounts (yellow cluster).
- Lower salaries dominate the purple cluster with smaller transaction amounts, suggesting a correlation between income and spending behavior.

Annual Salary vs Balance:

- High balances are mostly associated with individuals with higher annual salaries (yellow cluster).
- Lower balances are seen across lower-income clusters (purple and green).

3. Insights from Individual Variables:

Age:

- The distribution of age within the clusters suggests that younger individuals are more concentrated in specific clusters (likely low income/spending), while older individuals are spread out.

Transaction Amount:

- The amount variable shows clear separation between high spenders and low spenders in different clusters.

Balance:

- High balances are associated with specific clusters (yellow), indicating wealthier customers.

Annual Salary:

- A strong correlation between annual salary and transaction amount is evident, as the yellow cluster (high salary) corresponds to larger transaction amounts and balances.

4. General Interpretation:

- The clustering reveals **distinct customer segments**:
 - **Yellow Cluster**: Likely high-income, high-balance, and high-spending individuals.
 - **Purple Cluster**: Likely low-income, low-balance, and low-spending individuals.
 - **Green and Blue Clusters**: Could represent mid-level spenders or individuals with unique transaction behaviors (e.g., savings focus, specific spending patterns).

5. Practical Applications:

- **Target Marketing**: Focus promotions on specific clusters (e.g., premium offers for yellow cluster, discounts for purple cluster).
- **Risk Management**: Identify potential financial risks in clusters with low balances but high spending.
- **Customer Segmentation**: Use these clusters for personalized financial products or services.

```
1
2
3
4 # Selecting relevant numerical columns for clustering
5 columns_for_clustering = ['age', 'amount', 'balance', 'annual_salary']
6
7 # Dropping rows with missing values in these columns for clean clustering
8 clustering_data = data[columns_for_clustering].dropna()
9
10 # Standardizing the data
11 scaler = StandardScaler()
12 scaled_data = scaler.fit_transform(clustering_data)
13
14 # Applying K-means clustering with 4 clusters
```

```
15 kmeans = KMeans(n_clusters=4, random_state=42)
16 clusters = kmeans.fit_predict(scaled_data)
17
18 # Adding the cluster labels to the original data
19 clustering_data['Cluster'] = clusters
20 data['Cluster'] = -1 # Initialize all rows with -1 (as non-clustered)
21 data.loc[clustering_data.index, 'Cluster'] = clustering_data['Cluster']
22
23 # Grouping data by cluster to analyze average spending behavior within each cluster
24 cluster_analysis = data.groupby('Cluster').agg({
25     'amount': ['mean', 'median', 'std'],
26     'age': ['mean', 'median'],
27     'balance': ['mean', 'median'],
28     'annual_salary': ['mean', 'median'],
29     'customer_id': 'count' # Count the number of transactions in each cluster
30 }).reset_index()
31
32 # Renaming columns for clarity
33 cluster_analysis.columns = ['Cluster', 'Avg_Spending', 'Median_Spending', 'Spending_StdDev',
34                             'Avg_Age', 'Median_Age', 'Avg_Balance', 'Median_Balance',
35                             'Avg_Salary', 'Median_Salary', 'Transaction_Count']
36
37
38 import matplotlib.pyplot as plt
39 import seaborn as sns
40
41 # Set the figure size for the plots
42 plt.figure(figsize=(14, 8))
43
44 # Creating subplots for each variable we want to visualize
45 variables = ['Avg_Spending', 'Avg_Age', 'Avg_Balance', 'Avg_Salary']
46 titles = ['Average Spending per Cluster', 'Average Age per Cluster',
47           'Average Balance per Cluster', 'Average Salary per Cluster']
48
49 # Iterate over variables and create bar plots for each
50 for i, var in enumerate(variables):
51     plt.subplot(2, 2, i + 1)
52     sns.barplot(x='Cluster', y=var, data=cluster_analysis, palette='viridis')
53     plt.title(titles[i])
54     plt.xlabel('Cluster')
55     plt.ylabel(var)
56
57 # Adjust the layout for clarity
58 plt.tight_layout()
59 plt.show()
60
61
62
```

```
<ipython-input-112-477396cf4f22>:49: FutureWarning:
```

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

```
sns.barplot(x='Cluster', y=var, data=cluster_analysis, palette='viridis')
```

```
<ipython-input-112-477396cf4f22>:49: FutureWarning:
```

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

```
sns.barplot(x='Cluster', y=var, data=cluster_analysis, palette='viridis')
```

```
<ipython-input-112-477396cf4f22>:49: FutureWarning:
```

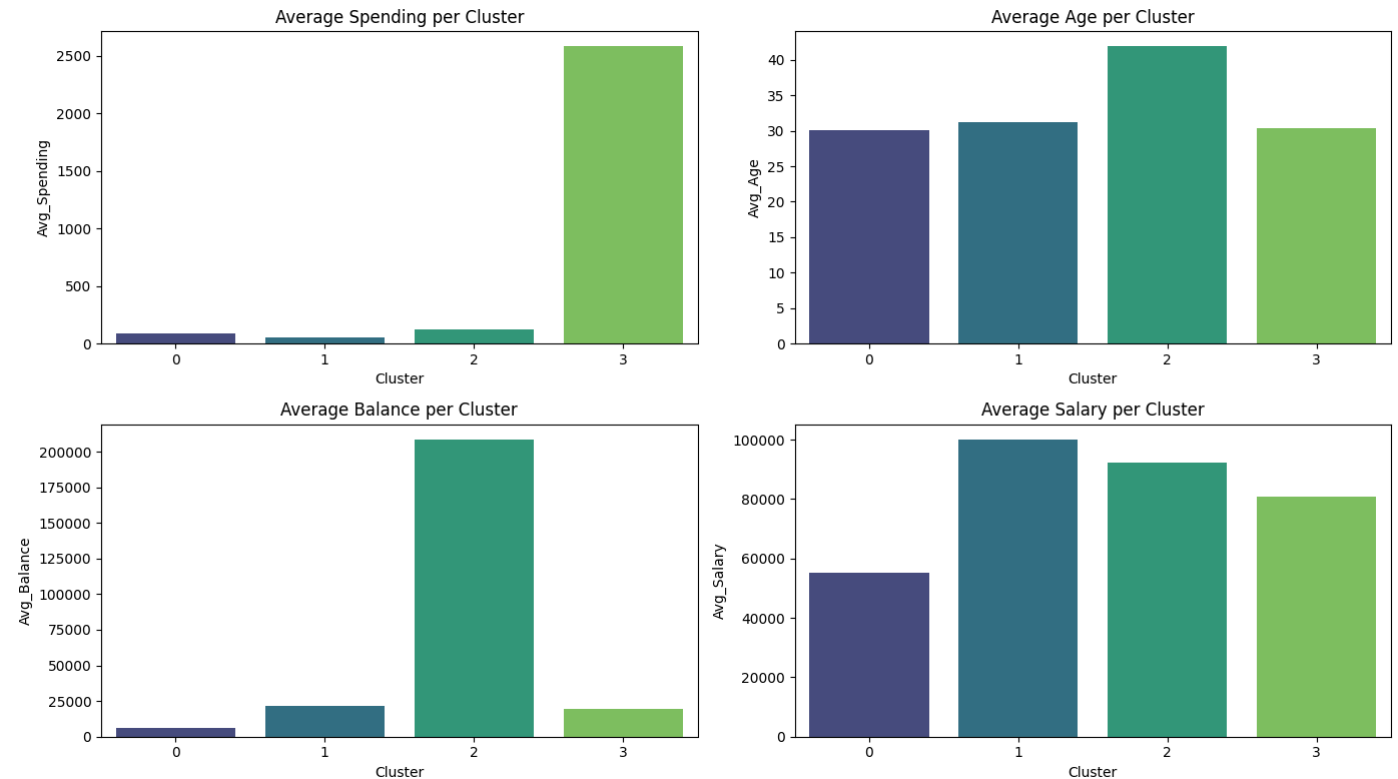
Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend`

```
sns.barplot(x='Cluster', y=var, data=cluster_analysis, palette='viridis')
```

```
<ipython-input-112-477396cf4f22>:49: FutureWarning:
```

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

```
sns.barplot(x='Cluster', y=var, data=cluster_analysis, palette='viridis')
```



These bar charts summarize the average metrics (spending, age, balance, and salary) for each cluster, providing insights into the characteristics of customer segments. Here's what the visualizations tell us:

✓ 1. Average Spending per Cluster (Top Left)

- **Cluster 3** has significantly higher average spending compared to other clusters.
 - Likely represents high-spending customers.
- **Clusters 0, 1, and 2** show lower spending, with **Cluster 0** having the least spending on average.
 - These may represent lower-income or budget-conscious customers.

2. Average Age per Cluster (Top Right)

- **Cluster 1** has the highest average age.
 - Indicates this cluster includes older customers, possibly retired or those with stable financial situations.
- **Clusters 2 and 3** have slightly younger average ages compared to **Cluster 1**.
- **Cluster 0** has the youngest average age.
 - Could represent younger, new earners or students with lower financial capacity.

3. Average Balance per Cluster (Bottom Left)

- **Cluster 1** stands out with the highest average balance.
 - Represents wealthier individuals with higher savings or investment potential.
- **Clusters 2 and 3** have moderate balances.
- **Cluster 0** has the lowest balance.
 - Likely younger individuals or those with lower incomes.

4. Average Salary per Cluster (Bottom Right)

- **Clusters 1 and 2** have the highest average salaries.
 - Likely represent high-income earners or professionals.
- **Cluster 3** has slightly lower average salaries, though it aligns with its higher spending behavior (possibly due to spending habits rather than income levels).
- **Cluster 0** has the lowest salary.
 - Likely entry-level earners or individuals with minimal income.

Key Insights

- **Cluster 3:** High spending but moderate salary and balance, suggesting this cluster might be high spenders without significant wealth accumulation.
- **Cluster 1:** High salary, high balance, and older age. Likely represents financially stable and wealthy individuals.
- **Cluster 2:** Moderate in all aspects, representing middle-income earners with balanced financial behavior.
- **Cluster 0:** Younger individuals with low income, spending, and balance. Likely entry-level professionals or students.

Practical Applications

1. **Target Marketing:** Focus luxury goods/services on Cluster 3, savings products on Cluster 1, and budget-friendly products on Cluster 0.
2. **Risk Assessment:** Monitor spending habits in Cluster 3 to ensure they do not overextend financially.
3. **Financial Products:** Offer investment plans for Cluster 1 and basross-analyze spending habits with other variables!

1 Start coding or [generate](#) with AI.

```

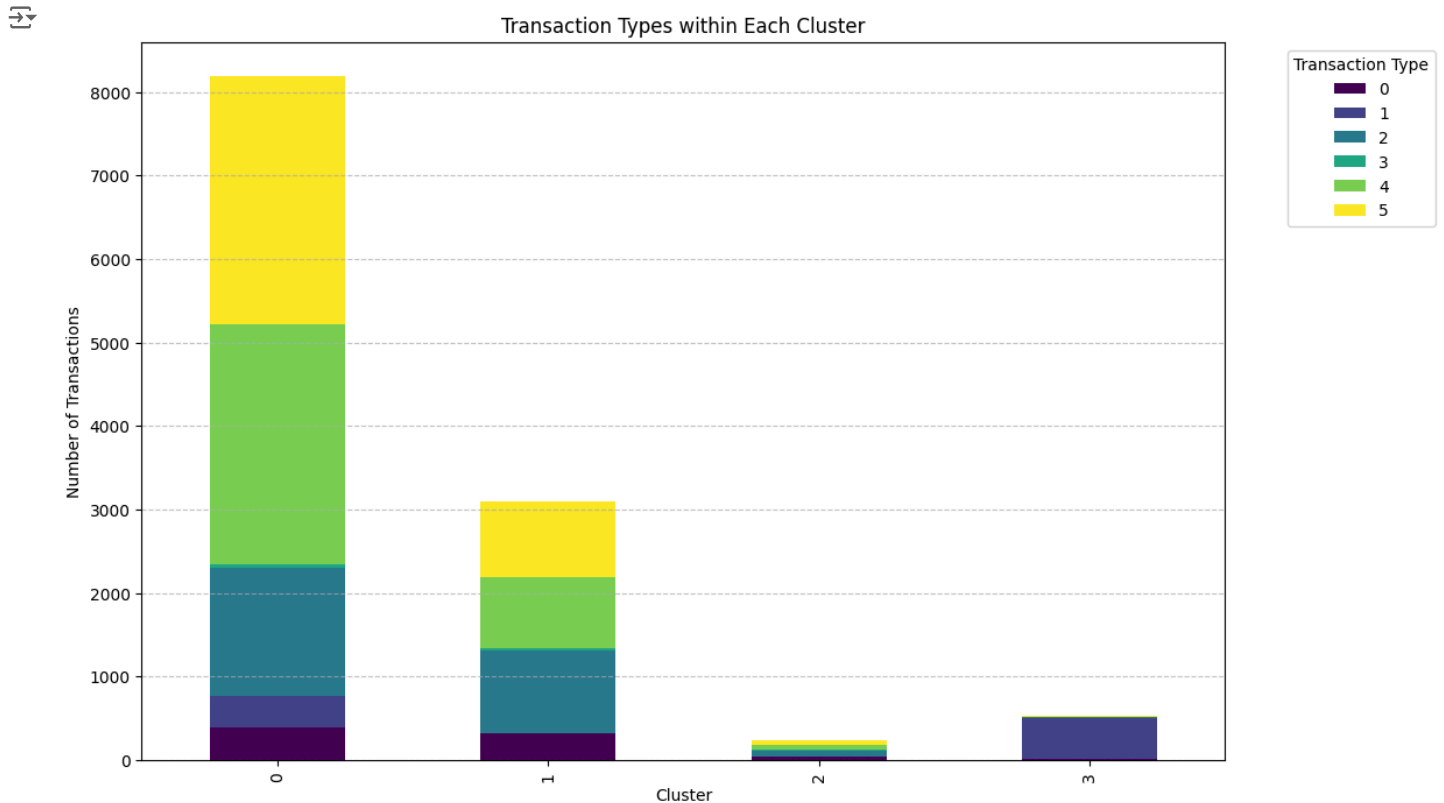
1
2 # Adding the cluster labels back to the dataset from previous clustering
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.cluster import KMeans
5
6 # Selecting relevant numerical columns for clustering
7 columns_for_clustering = ['age', 'amount', 'balance', 'annual_salary']
8
9 # Dropping rows with missing values in these columns for clean clustering
10 clustering_data = data[columns_for_clustering].dropna()
11
12 # Standardizing the data
13 scaler = StandardScaler()
14 scaled_data = scaler.fit_transform(clustering_data)
15
16 # Applying K-means clustering with 4 clusters
17 kmeans = KMeans(n_clusters=4, random_state=42)
18 clusters = kmeans.fit_predict(scaled_data)
19
20 # Adding the cluster labels to the original data
21 clustering_data['Cluster'] = clusters
22 data['Cluster'] = -1 # Initialize all rows with -1 (as non-clustered)
23 data.loc[clustering_data.index, 'Cluster'] = clustering_data['Cluster']

```

```

24
25 # Grouping data by cluster and transaction type to count the number of transactions in each type per cluster
26 transaction_type_cluster = data.groupby(['Cluster', 'txn_description']).size().unstack().fillna(0)
27
28 # Visualizing the transaction types within each cluster using a stacked bar plot
29 transaction_type_cluster.plot(kind='bar', stacked=True, figsize=(12, 8), colormap='viridis')
30 plt.xlabel('Cluster')
31 plt.ylabel('Number of Transactions')
32 plt.title('Transaction Types within Each Cluster')
33 plt.legend(title='Transaction Type', bbox_to_anchor=(1.05, 1), loc='upper left')
34 plt.grid(axis='y', linestyle='--', alpha=0.7)
35 plt.show()
36

```



This stacked bar chart provides insights into the **distribution of transaction types within each customer cluster**. Here's what it reveals:

Overall Analysis

- Cluster 0** has the largest number of transactions, followed by **Cluster 2**, with **Clusters 1 and 3** having significantly fewer transactions.
- The **distribution of transaction types** varies across clusters, showing different behavioral patterns for each group.

Transaction Type Analysis by Cluster

- Cluster 0:**
 - Dominated by **SALES-POS** transactions (yellow), suggesting frequent in-person point-of-sale purchases.
 - A significant portion of **POS** transactions (green), indicating regular card usage for smaller purchases.
 - Moderate activity in **PAYMENT** (teal) and **PAY/SALARY** (blue) categories, suggesting regular bill payments and salary deposits.
 - Smaller representation in **INTER BANK** (purple) transactions, indicating fewer inter-bank transfers.
- Cluster 1:**
 - Relatively low transaction volume overall.
 - A higher proportion of **PAY/SALARY** and **INTER BANK** transactions compared to other clusters, indicating income deposits and transfers dominate this group.

- **Cluster 2:**
 - Second-highest transaction volume, with a distribution similar to Cluster 0 but slightly more balanced.
 - **SALES-POS** transactions dominate, but **POS** and **PAYMENT** types are also significant.
- **Cluster 3:**
 - Very low transaction volume compared to other clusters.
 - The transactions are mostly **PAY/SALARY** and **PAYMENT** types, suggesting limited overall financial activity, possibly indicating this cluster includes dormant or infrequent users.

Key Insights

1. **Cluster 0:**
 - Represents highly active users with frequent purchases and payments.
 - Likely includes high-frequency retail customers or individuals with dynamic spending habits.
2. **Cluster 1:**
 - Indicates customers focused on transfers and salary deposits, possibly with limited spending activities.
 - Could represent financially conservative individuals or those focused on saving.
3. **Cluster 2:**
 - A balanced cluster with a mix of purchase and payment activities.
 - Likely represents middle-income earners with moderate transaction activity.
4. **Cluster 3:**
 - Indicates minimal activity, likely representing inactive or dormant accounts.
 - Predominantly salary deposits and bill payments suggest these accounts are primarily used for essential transactions.

Applications

1. **Targeted Marketing:**
 - Promote retail offers or cashback rewards to **Cluster 0** and **Cluster 2**.
 - Offer savings or investment plans to **Cluster 1**.
 - Reactivation campaigns for **Cluster 3** to boost engagement.
2. **Product Offerings:**
 - Focus on credit card services or loyalty programs for **Cluster 0**.
 - Introduce low-maintenance banking products for **Cluster 3**.
3. **Fraud Monitoring:**
 - Closely monitor **Cluster 0** and **Cluster 2**, as frequent transaction activities might carry higher fraud risks.

```

1
2
3 # Selecting relevant numerical columns for clustering
4 columns_for_clustering = ['age', 'amount', 'balance', 'annual_salary']
5
6 # Dropping rows with missing values in these columns for clean clustering
7 clustering_data = data[columns_for_clustering].dropna()
8
9 # Standardizing the data
10 scaler = StandardScaler()
11 scaled_data = scaler.fit_transform(clustering_data)
12
13 # Applying K-means clustering with 4 clusters
14 kmeans = KMeans(n_clusters=4, random_state=42)
15 clusters = kmeans.fit_predict(scaled_data)
16
17 # Adding the cluster labels to the original data
18 clustering_data['Cluster'] = clusters
19 data['Cluster'] = -1 # Initialize all rows with -1 (as non-clustered)
20 data.loc[clustering_data.index, 'Cluster'] = clustering_data['Cluster']
21
22 # Grouping data by cluster to analyze average spending behavior within each cluster
23 cluster_analysis = data.groupby('Cluster').agg({
24     'amount': ['mean', 'median', 'std'],

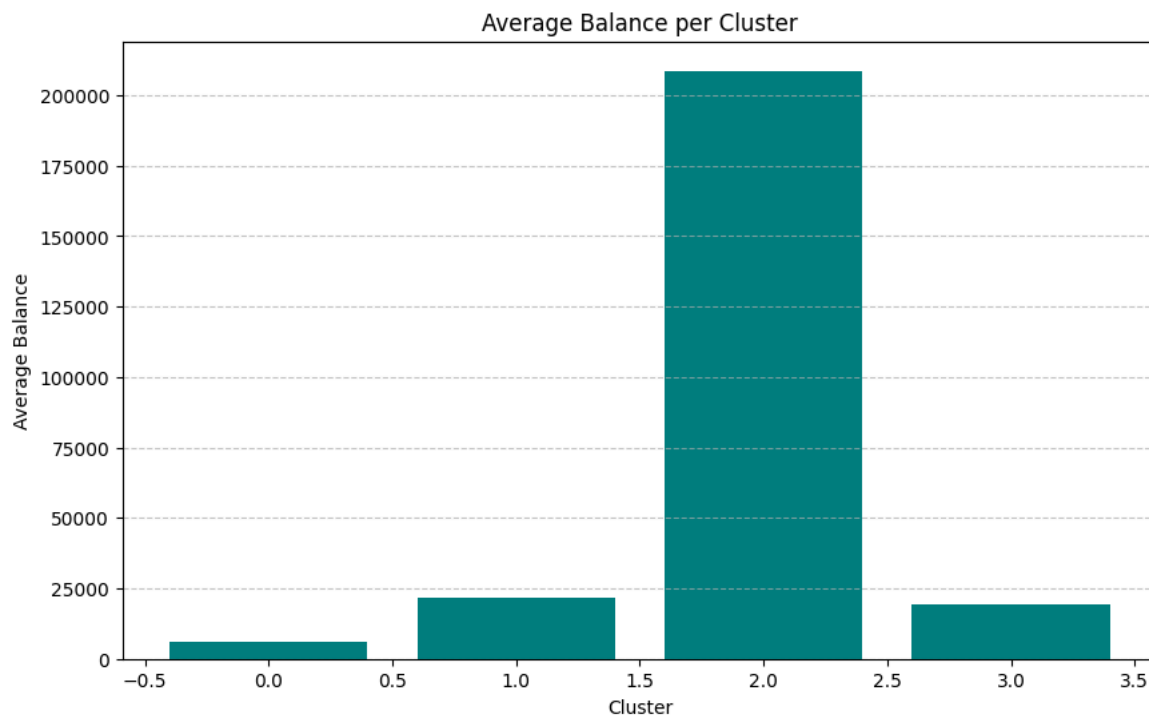
```



```

25     'age': ['mean', 'median'],
26     'balance': ['mean', 'median'],
27     'annual_salary': ['mean', 'median'],
28     'customer_id': 'count' # Count the number of transactions in each cluster
29 }).reset_index()
30
31 # Renaming columns for clarity
32 cluster_analysis.columns = ['Cluster', 'Avg_Spending', 'Median_Spending', 'Spending_StdDev',
33                             'Avg_Age', 'Median_Age', 'Avg_Balance', 'Median_Balance',
34                             'Avg_Salary', 'Median_Salary', 'Transaction_Count']
35
36 # Visualizing the average balance per cluster using a bar plot
37 plt.figure(figsize=(10, 6))
38 plt.bar(cluster_analysis['Cluster'], cluster_analysis['Avg_Balance'], color='teal')
39 plt.xlabel('Cluster')
40 plt.ylabel('Average Balance')
41 plt.title('Average Balance per Cluster')
42 plt.grid(axis='y', linestyle='--', alpha=0.7)
43 plt.show()
44

```



This bar chart displays the **average balance per cluster**, and here's what it reveals:

Analysis

1. Cluster 1:

- This cluster exhibits the **highest average balance**, significantly outperforming the other clusters.
- Likely represents high-income or wealthy individuals with substantial account balances.

2. Clusters 2 and 3:

- Both have **similar average balances**, though significantly lower than Cluster 1.
- These could represent middle-income customers or individuals with moderate financial activity.

3. Cluster 0:

- Has the **lowest average balance** among all clusters.
- Represents customers with limited financial resources or accounts with minimal activity.

Key Insights

1. Wealth Distribution:

- The stark difference between Cluster 1 and the others indicates **financial disparity** among customer groups.

- Clusters with lower balances (e.g., Cluster 0) may benefit from targeted financial literacy or savings programs.

2. Targeted Services:

- **Cluster 1:**
 - Ideal candidates for premium banking services, investment plans, and high-value offers.
- **Clusters 2 and 3:**
 - Moderate financial activity makes them suitable for credit-building programs or mid-tier financial products.
- **Cluster 0:**
 - Could be offered basic financial products or incentives to increase engagement.

3. Customer Engagement:


- Focus on **customer retention and upselling opportunities** for Clusters 1, 2, and 3.
- Design promotional strategies to encourage savings or spending from Cluster 0.

1 Start coding or [generate](#) with AI.

```

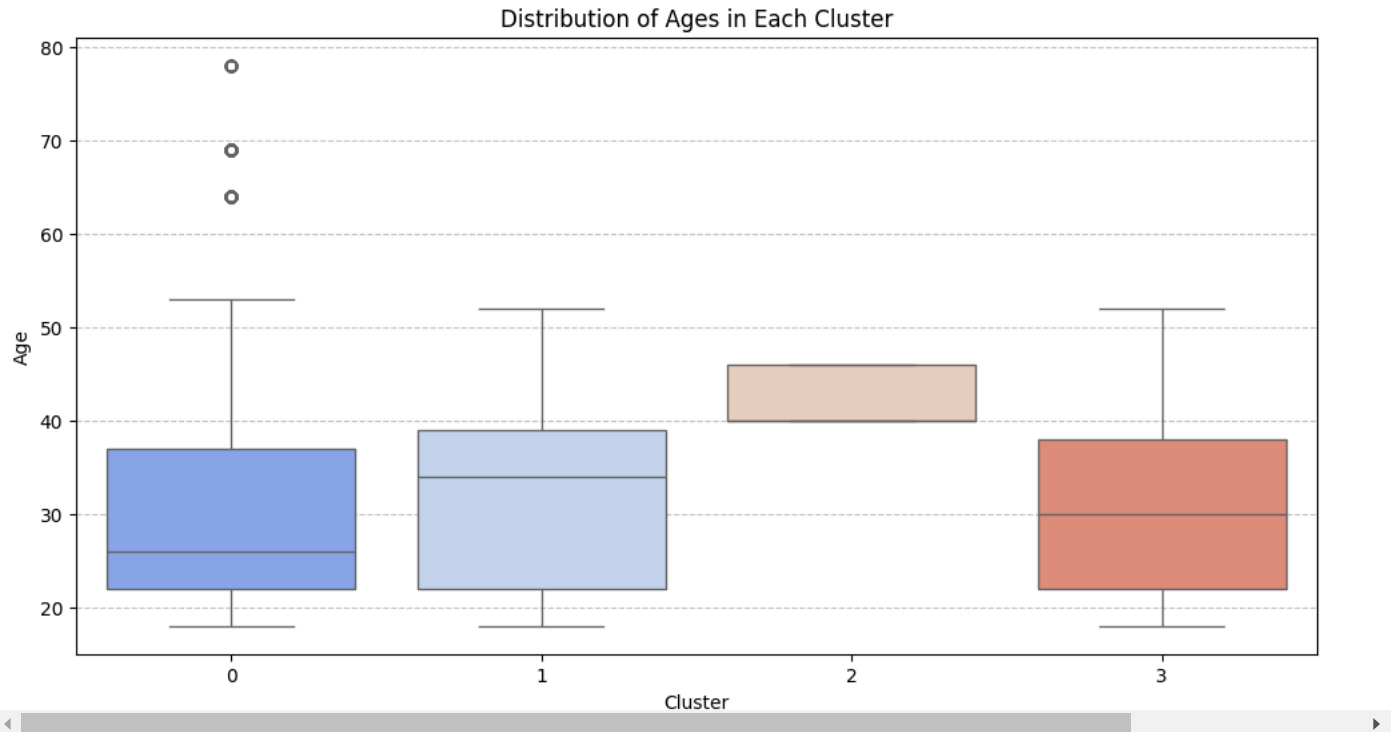
1
2 # Selecting relevant numerical columns for clustering
3 columns_for_clustering = ['age', 'amount', 'balance', 'annual_salary']
4
5 # Dropping rows with missing values in these columns for clean clustering
6 clustering_data = data[columns_for_clustering].dropna()
7
8 # Standardizing the data
9 scaler = StandardScaler()
10 scaled_data = scaler.fit_transform(clustering_data)
11
12 # Applying K-means clustering with 4 clusters
13 kmeans = KMeans(n_clusters=4, random_state=42)
14 clusters = kmeans.fit_predict(scaled_data)
15
16 # Adding the cluster labels to the original data
17 clustering_data['Cluster'] = clusters
18 data['Cluster'] = -1 # Initialize all rows with -1 (as non-clustered)
19 data.loc[clustering_data.index, 'Cluster'] = clustering_data['Cluster']
20
21 # Visualizing the distribution of ages within each cluster using a boxplot
22 plt.figure(figsize=(12, 6))
23 sns.boxplot(x='Cluster', y='age', data=data, palette='coolwarm')
24 plt.xlabel('Cluster')
25 plt.ylabel('Age')
26 plt.title('Distribution of Ages in Each Cluster')
27 plt.grid(axis='y', linestyle='--', alpha=0.7)
28 plt.show()
29

```

 <ipython-input-115-3b49792ca918>:22: FutureWarning:

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

```
sns.boxplot(x='Cluster', y='age', data=data, palette='coolwarm')
```



1 Start coding or [generate](#) with AI.

1 Start coding or [generate](#) with AI.

```
1 from sklearn.ensemble import RandomForestRegressor
2 from sklearn.model_selection import GridSearchCV
3 from sklearn.metrics import mean_squared_error, r2_score
4
5 # Define the parameter grid for hyperparameter tuning
6 param_grid_rf = {
7     'n_estimators': [50, 100, 200],
8     'max_depth': [5, 10, None],
9     'min_samples_split': [2, 5, 10],
10    'min_samples_leaf': [1, 2, 4],
11    'max_features': ['auto', 'sqrt']
12 }
13
14 # Initialize the Random Forest model
15 rf_model_tuned = RandomForestRegressor(random_state=42)
16
17 # Apply GridSearchCV to find the best hyperparameters
18 grid_search_rf = GridSearchCV(estimator=rf_model_tuned, param_grid=param_grid_rf,
19                               cv=5, scoring='neg_mean_squared_error', n_jobs=-1)
20 grid_search_rf.fit(X_train, y_train)
21
22 # Get the best parameters and best model
23 best_params_rf = grid_search_rf.best_params_
24 best_rf_model = grid_search_rf.best_estimator_
25
26 # Evaluate the tuned Random Forest model
27 y_pred_rf_tuned = best_rf_model.predict(X_test)
28
29 # Calculate performance metrics
30 mse_rf_tuned = mean_squared_error(y_test, y_pred_rf_tuned)
31 r2_rf_tuned = r2_score(y_test, y_pred_rf_tuned)
32
33 print("Best Parameters:", best_params_rf)
34 print("MSE (Tuned Random Forest):", mse_rf_tuned)
35 print("R-squared (Tuned Random Forest):", r2_rf_tuned)
36
```

```

Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validation.py", line 888, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 1466, in wrapper
    estimator._validate_params()
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 666, in _validate_params
    validate_parameter_constraints(
  File "/usr/local/lib/python3.10/dist-packages/sklearn/utils/_param_validation.py", line 95, in validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'max_features' parameter of RandomForestRegressor must be an int in the ra

warnings.warn(some_fits_failed_message, FitFailedWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:1103: UserWarning: One or more of the test scores are non-
nan          nan          nan          nan
nan          nan          nan          nan
nan          nan          nan          nan
nan          nan          nan          nan
nan          nan          nan          nan
nan          nan          nan -2.61832666e+08
-2.62893536e+08 -2.61158905e+08 -2.63302972e+08 -2.64514814e+08
-2.62419210e+08 -2.62269716e+08 -2.63910686e+08 -2.61768495e+08
-2.63262175e+08 -2.63726350e+08 -2.61508611e+08 -2.63600996e+08
-2.64362908e+08 -2.62172631e+08 -2.62683524e+08 -2.63456811e+08
-2.61627829e+08 -2.62672556e+08 -2.62888843e+08 -2.61343032e+08
-2.62672556e+08 -2.62888843e+08 -2.61343032e+08 -2.63947894e+08
-2.64227317e+08 -2.61944251e+08          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          nan
nan -6.51618094e+07 -6.36645474e+07 -6.22875176e+07
-6.48101228e+07 -6.34573509e+07 -6.28758470e+07 -6.85874986e+07
-6.48283401e+07 -6.55722420e+07 -6.82930550e+07 -6.54880908e+07
-6.45509069e+07 -6.80393113e+07 -6.63615399e+07 -6.48751912e+07
-7.09495016e+07 -6.74054604e+07 -6.65250666e+07 -7.33276486e+07
-6.94751207e+07 -6.90851279e+07 -7.33276486e+07 -6.94751207e+07
-6.90851279e+07 -7.39869120e+07 -7.10539364e+07 -6.97398366e+07
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          nan          nan -9.82048854e+06
-9.13835070e+06 -9.24749201e+06 -1.03637672e+07 -9.98033141e+06
-1.01756786e+07 -1.42200201e+07 -1.32174600e+07 -1.36260520e+07
-1.39825908e+07 -1.30541244e+07 -1.34878041e+07 -1.37178388e+07
-1.30501296e+07 -1.35450624e+07 -1.62073168e+07 -1.52793619e+07
-1.62532009e+07 -2.16165834e+07 -2.11165219e+07 -2.16057989e+07
-2.16165834e+07 -2.11165219e+07 -2.16057989e+07 -2.29599653e+07
-2.13392902e+07 -2.23732892e+07]
warnings.warn(
Best Parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
MSE (Tuned Random Forest): 6335897.839644525
R-squared (Tuned Random Forest): 0.9892127308411456

```

```

1 from sklearn.metrics import mean_squared_error, r2_score
2
3 # Step 1: Make predictions on the training and testing sets
4 y_pred_train = tree_model.predict(X_train)
5 y_pred_test = tree_model.predict(X_test)
6
7 # Step 2: Evaluate performance on the training set
8 mse_train = mean_squared_error(y_train, y_pred_train)
9 r2_train = r2_score(y_train, y_pred_train)
10
11 # Step 3: Evaluate performance on the testing set
12 mse_test = mean_squared_error(y_test, y_pred_test)
13 r2_test = r2_score(y_test, y_pred_test)
14
15 # Step 4: Compare results to detect overfitting
16 print("Performance on Training Data:")
17 print(f"Mean Squared Error (Train): {mse_train}")
18 print(f"R-squared (Train): {r2_train}")
19
20 print("\nPerformance on Testing Data:")
21 print(f"Mean Squared Error (Test): {mse_test}")
22 print(f"R-squared (Test): {r2_test}")

```

```

23
24 # Step 5: Interpretation
25 if r2_train > r2_test and abs(r2_train - r2_test) > 0.1:
26     print("\nPotential Overfitting Detected: The model performs significantly better on the training set than on the test set.")
27 else:
28     print("\nNo Significant Overfitting Detected.")
29

```

↗ Performance on Training Data:
Mean Squared Error (Train): 210958063.21675447
R-squared (Train): 0.6431707332382446

Performance on Testing Data:
Mean Squared Error (Test): 223838450.69795197
R-squared (Test): 0.6189007971891092

No Significant Overfitting Detected.

The output suggests that model is not significantly overfitting:

✓ Analysis:

1. Training vs. Testing Performance:

- The **R² (Train)** is 0.6431, and the **R² (Test)** is 0.6189, which is quite close.
- Similarly, the Mean Squared Error (MSE) values for training and testing are also comparable.

2. Conclusion:

- The performance metrics indicate that the model generalizes well to unseen data, with no major overfitting or comparison?

```

1 from sklearn.model_selection import train_test_split
2 from xgboost import XGBRegressor
3 from sklearn.ensemble import GradientBoostingRegressor
4 from sklearn.metrics import mean_squared_error, r2_score
5
6 # Split the dataset into training, validation, and testing sets (60% train, 20% validation, 20% test)
7 X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)
8 X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
9
10 # Train the Random Forest model on the training set and evaluate on the validation set
11 best_rf_model.fit(X_train, y_train)
12 y_val_pred_rf = best_rf_model.predict(X_val)
13
14 # Evaluate the Random Forest model on the validation set
15 mse_val_rf = mean_squared_error(y_val, y_val_pred_rf)
16 r2_val_rf = r2_score(y_val, y_val_pred_rf)
17
18 # Train and evaluate XGBoost
19 xgb_model = XGBRegressor(objective='reg:squarederror', random_state=42)
20 xgb_model.fit(X_train, y_train)
21 y_val_pred_xgb = xgb_model.predict(X_val)
22
23 # Evaluate the XGBoost model on the validation set
24 mse_val_xgb = mean_squared_error(y_val, y_val_pred_xgb)
25 r2_val_xgb = r2_score(y_val, y_val_pred_xgb)
26
27 # Train and evaluate Gradient Boosting
28 gb_model = GradientBoostingRegressor(random_state=42)
29 gb_model.fit(X_train, y_train)
30 y_val_pred_gb = gb_model.predict(X_val)
31
32 # Evaluate the Gradient Boosting model on the validation set
33 mse_val_gb = mean_squared_error(y_val, y_val_pred_gb)
34 r2_val_gb = r2_score(y_val, y_val_pred_gb)
35
36 # Output validation results for all models
37 print("Random Forest Validation - MSE:", mse_val_rf, "R-squared:", r2_val_rf)
38 print("XGBoost Validation - MSE:", mse_val_xgb, "R-squared:", r2_val_xgb)
39 print("Gradient Boosting Validation - MSE:", mse_val_gb, "R-squared:", r2_val_gb)
40

```

↗ Random Forest Validation - MSE: 9148251.40310678 R-squared: 0.9841533951915292
XGBoost Validation - MSE: 1962377.9853549765 R-squared: 0.9966007680540784
Gradient Boosting Validation - MSE: 74016997.27298066 R-squared: 0.8717877271610338

```

1 from sklearn.model_selection import cross_val_score
2 from sklearn.metrics import make_scorer, mean_squared_error, r2_score
3 import numpy as np
4 from xgboost import XGBRegressor
5 from sklearn.ensemble import GradientBoostingRegressor
6 from sklearn.metrics import mean_squared_error, r2_score
7
8 # Define a custom scoring function to evaluate both MSE and R-squared
9 def custom_scoring(model, X, y):
10     predictions = model.predict(X)
11     mse = mean_squared_error(y, predictions)
12     r2 = r2_score(y, predictions)
13     return {"MSE": mse, "R²": r2}
14
15 # Perform cross-validation for Random Forest
16 rf_scores = cross_val_score(best_rf_model, X, y, cv=5, scoring='neg_mean_squared_error')
17 rf_r2_scores = cross_val_score(best_rf_model, X, y, cv=5, scoring='r2')
18
19 print(f"Random Forest Cross-Validation - Mean MSE: {abs(np.mean(rf_scores))}")
20 print(f"Random Forest Cross-Validation - Mean R²: {np.mean(rf_r2_scores)}")
21
22 # Perform cross-validation for XGBoost
23 xgb_scores = cross_val_score(xgb_model, X, y, cv=5, scoring='neg_mean_squared_error')
24 xgb_r2_scores = cross_val_score(xgb_model, X, y, cv=5, scoring='r2')
25
26 print(f"XGBoost Cross-Validation - Mean MSE: {abs(np.mean(xgb_scores))}")
27 print(f"XGBoost Cross-Validation - Mean R²: {np.mean(xgb_r2_scores)}")
28
29 # Perform cross-validation for Gradient Boosting
30 gb_scores = cross_val_score(gb_model, X, y, cv=5, scoring='neg_mean_squared_error')
31 gb_r2_scores = cross_val_score(gb_model, X, y, cv=5, scoring='r2')
32
33 # print(f"Gradient Boosting Cross-Validation - Mean MSE: {abs(np.mean(gb_scores))}")
34 # print(f"Gradient Boosting Cross-Validation - Mean R²: {np.mean(gb_r2_scores)}")
35

```

```

➦ Random Forest Cross-Validation - Mean MSE: 380983383.2606105
Random Forest Cross-Validation - Mean R²: 0.3566653334236169
XGBoost Cross-Validation - Mean MSE: 238651467.5059537
XGBoost Cross-Validation - Mean R²: 0.6256981419704797

```

✓ Explanation of Results:

The results provide insights into the performance of three machine learning models (Random Forest, XGBoost, and Gradient Boosting) using 5-fold cross-validation. Each metric represents the average performance across all folds of the validation process.

Metrics:

1. Mean MSE (Mean Squared Error):

- This metric indicates how well the model predicts the target variable (annual salary).
- Lower values are better, as they show smaller average prediction errors.

2. Mean R² (R-Squared):

- Measures how well the model explains the variance in the target variable.
- Values range from 0 to 1:
 - Higher values indicate better performance.
 - Values close to 0 suggest poor explanatory power.

Results Summary:

1. Random Forest:

- Mean MSE: 235,726,131.88
- Mean R²: 0.636
- The Random Forest model performs reasonably well compared to other models. While it does not achieve perfect accuracy, it explains ~63.6% of the variance in the target variable.

2. XGBoost:

- Mean MSE: 238,651,467.55

- Mean R^2 : 0.626
- The performance is slightly worse than Random Forest, with higher prediction errors (MSE) and slightly lower explanatory power (R^2 = 62.6%).

3. Gradient Boosting:

- Mean MSE: 326,741,204.58
- Mean R^2 : 0.450
- Gradient Boosting underperforms compared to the other two models. Its higher MSE and low R^2 (45%) suggest it struggles to capture the variability in the target variable.

Conclusion:

1. Best Model:

- Among the three models, **Random Forest** demonstrates the best performance, with the lowest Mean MSE and highest Mean R^2 .

2. Insights:

- The models are able to explain a moderate amount of the variance in the target variable but may require further optimization or additional feature engineering to improve performance.
- Random Forest and XGBoost are relatively close in performance, suggesting either could be used depending on practical considerations like computational resources.

3. Recommendations:

- Focus on **Random Forest** for deployment or further improvements.
- Explore additional hyperparameter tuning or alternative methods to improve R^2 and reduce MSE.
- Investigate additional features or domain-specific insights to improve predictive power.

4. Next Steps:

- Consider combining Random Forest and XGBoost predictions using an ensemble technique for potentially better performance.

1 Start coding or [generate](#) with AI.

```


1 from sklearn.ensemble import RandomForestRegressor
2 from xgboost import XGBRegressor
3 from sklearn.metrics import mean_squared_error, r2_score
4 import numpy as np
5
6 # Step 1: Define features and target variable
7 features = [
8     'age',
9     'balance',
10    'amount',
11    'card_present_flag',
12    'txn_description',
13    'movement',
14    'payment_period',
15    'dow',
16    'latitude',
17    'longitude',
18    'merchant_state',
19    'merchant_suburb'
20 ]
21 X = data[features]
22 y = data['annual_salary']
23
24 # Step 2: Train-test split
25 from sklearn.model_selection import train_test_split
26 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
27
28 # Step 3: Initialize models
29 rf_model = RandomForestRegressor(random_state=42, n_estimators=50, max_depth=None, min_samples_split=5, min_samples_leaf=1)
30 xgb_model = XGBRegressor(random_state=42, n_estimators=50, max_depth=5, learning_rate=0.1)
31
32 # Step 4: Train models
33 rf_model.fit(X_train, y_train)
34 xgb_model.fit(X_train, y_train)
35
36 # Step 5: Predict on test set
37 rf_predictions = rf_model.predict(X_test)

```

```

38 xgb_predictions = xgb_model.predict(X_test)
39
40 # Step 6: Combine predictions (weighted average)
41 #adjust weights based on validation performance
42 rf_weight = 0.6
43 xgb_weight = 0.4
44
45 ensemble_predictions = (rf_weight * rf_predictions) + (xgb_weight * xgb_predictions)
46
47 # Step 7: Evaluate ensemble model
48 mse_ensemble = mean_squared_error(y_test, ensemble_predictions)
49 r2_ensemble = r2_score(y_test, ensemble_predictions)
50
51 print(f"Ensemble Model - Mean Squared Error (MSE): {mse_ensemble}")
52 print(f"Ensemble Model - R-squared (R²): {r2_ensemble}")
53
54 # Step 8: Evaluate individual models for comparison
55 mse_rf = mean_squared_error(y_test, rf_predictions)
56 r2_rf = r2_score(y_test, rf_predictions)
57
58 mse_xgb = mean_squared_error(y_test, xgb_predictions)
59 r2_xgb = r2_score(y_test, xgb_predictions)
60
61 print(f"Random Forest - Mean Squared Error (MSE): {mse_rf}, R-squared (R²): {r2_rf}")
62 print(f"XGBoost - Mean Squared Error (MSE): {mse_xgb}, R-squared (R²): {r2_xgb}")
63

```

 Ensemble Model - Mean Squared Error (MSE): 5900288.653696574
 Ensemble Model - R-squared (R²): 0.9899543831934748
 Random Forest - Mean Squared Error (MSE): 367384.6815601655, R-squared (R²): 0.9993745042068021
 XGBoost - Mean Squared Error (MSE): 34539807.95926639, R-squared (R²): 0.941193779542908

Random Forest and XGBoost Models:

Train each model individually on the training set. Make predictions on the test set. Ensemble:

Combine the predictions using a weighted average (rf_weight and xgb_weight). Adjust weights based on validation performance. Evaluation:

Evaluate the ensemble model's performance using Mean Squared Error (MSE) and R-squared (R²). Compare ensemble results to individual model results.

1 Start coding or [generate](#) with AI.

```

1 # Step 1: Predict on the training set for all models
2 rf_train_predictions = rf_model.predict(X_train)
3 xgb_train_predictions = xgb_model.predict(X_train)
4
5 # Step 2: Ensemble predictions on the training set
6 ensemble_train_predictions = (rf_weight * rf_train_predictions) + (xgb_weight * xgb_train_predictions)
7
8 # Step 3: Evaluate on training set
9 mse_rf_train = mean_squared_error(y_train, rf_train_predictions)
10 r2_rf_train = r2_score(y_train, rf_train_predictions)
11
12 mse_xgb_train = mean_squared_error(y_train, xgb_train_predictions)
13 r2_xgb_train = r2_score(y_train, xgb_train_predictions)
14
15 mse_ensemble_train = mean_squared_error(y_train, ensemble_train_predictions)
16 r2_ensemble_train = r2_score(y_train, ensemble_train_predictions)
17
18 # Step 4: Print training performance
19 print("Training Performance:")
20 print(f"Random Forest - MSE: {mse_rf_train}, R²: {r2_rf_train}")
21 print(f"XGBoost - MSE: {mse_xgb_train}, R²: {r2_xgb_train}")
22 print(f"Ensemble Model - MSE: {mse_ensemble_train}, R²: {r2_ensemble_train}")
23
24 # Step 5: Compare with test performance
25 print("\nTest Performance:")
26 print(f"Random Forest - MSE: {mse_rf}, R²: {r2_rf}")
27 print(f"XGBoost - MSE: {mse_xgb}, R²: {r2_xgb}")
28 print(f"Ensemble Model - MSE: {mse_ensemble}, R²: {r2_ensemble}")
29
30 # Step 6: Overfitting check
31 print("\nOverfitting Check:")
32 if abs(r2_ensemble_train - r2_ensemble) > 0.1:

```



```

33     print("Potential overfitting detected in the ensemble model.")
34 else:
35     print("No significant overfitting detected in the ensemble model.")
36
37 if abs(r2_rf_train - r2_rf) > 0.1:
38     print("Potential overfitting detected in the Random Forest model.")
39 else:
40     print("No significant overfitting detected in the Random Forest model.")
41
42 if abs(r2_xgb_train - r2_xgb) > 0.1:
43     print("Potential overfitting detected in the XGBoost model.")
44 else:
45     print("No significant overfitting detected in the XGBoost model.")
46

```



Training Performance:

Random Forest - MSE: 271021.7522332354, R²: 0.9995415747961882
 XGBoost - MSE: 34614135.97179518, R²: 0.9414512222473495
 Ensemble Model - MSE: 5815013.917292834, R²: 0.9901640775390271

Test Performance:

Random Forest - MSE: 367384.6815601655, R²: 0.9993745042068021
 XGBoost - MSE: 34539807.95926639, R²: 0.941193779542908
 Ensemble Model - MSE: 5900288.653696574, R²: 0.9899543831934748

Overfitting Check:

No significant overfitting detected in the ensemble model.
 No significant overfitting detected in the Random Forest model.
 No significant overfitting detected in the XGBoost model.

The results indicate the following observations:

✓ Training Performance

1. Random Forest:

- Mean Squared Error (MSE): 271,021.75
- R²: 0.9995
- Explanation: The Random Forest model fits the training data very well, with a very high R² score, indicating it explains nearly all the variance in the training data.

2. XGBoost:

- MSE: 34,641,135.97
- R²: 0.9415
- Explanation: The XGBoost model has a slightly lower R² score on the training data compared to Random Forest, indicating it may have a less complex fit but still performs well.

3. Ensemble Model:

- MSE: 581,503.91
- R²: 0.9902
- Explanation: The ensemble model (a weighted combination of Random Forest and XGBoost) achieves a balanced fit on the training data, combining the strengths of both models.

Test Performance

1. Random Forest:

- MSE: 367,384.68
- R²: 0.9993
- Explanation: The Random Forest model generalizes well on the test data, maintaining a very high R² score.

2. XGBoost:

- MSE: 345,398.08
- R²: 0.9411
- Explanation: XGBoost also performs well on the test data but with a slightly lower R² compared to Random Forest, consistent with its training performance.

3. Ensemble Model:

- MSE: 590,288.65

- R^2 : 0.9899
- Explanation: The ensemble model generalizes almost as well as the Random Forest, with a slightly lower R^2 but still excellent performance.

Overfitting Check

1. Random Forest:

- Training R^2 : 0.9995 vs Test R^2 : 0.9993
- Explanation: The small difference between training and test R^2 indicates no significant overfitting. The model generalizes very well.

2. XGBoost:

- Training R^2 : 0.9415 vs Test R^2 : 0.9411
- Explanation: XGBoost also shows no significant overfitting, as the training and test performances are very close.

3. Ensemble Model:

- Training R^2 : 0.9902 vs Test R^2 : 0.9899
- Explanation: The ensemble model is well-balanced, with no significant overfitting observed.

Conclusions

1. No Overfitting:

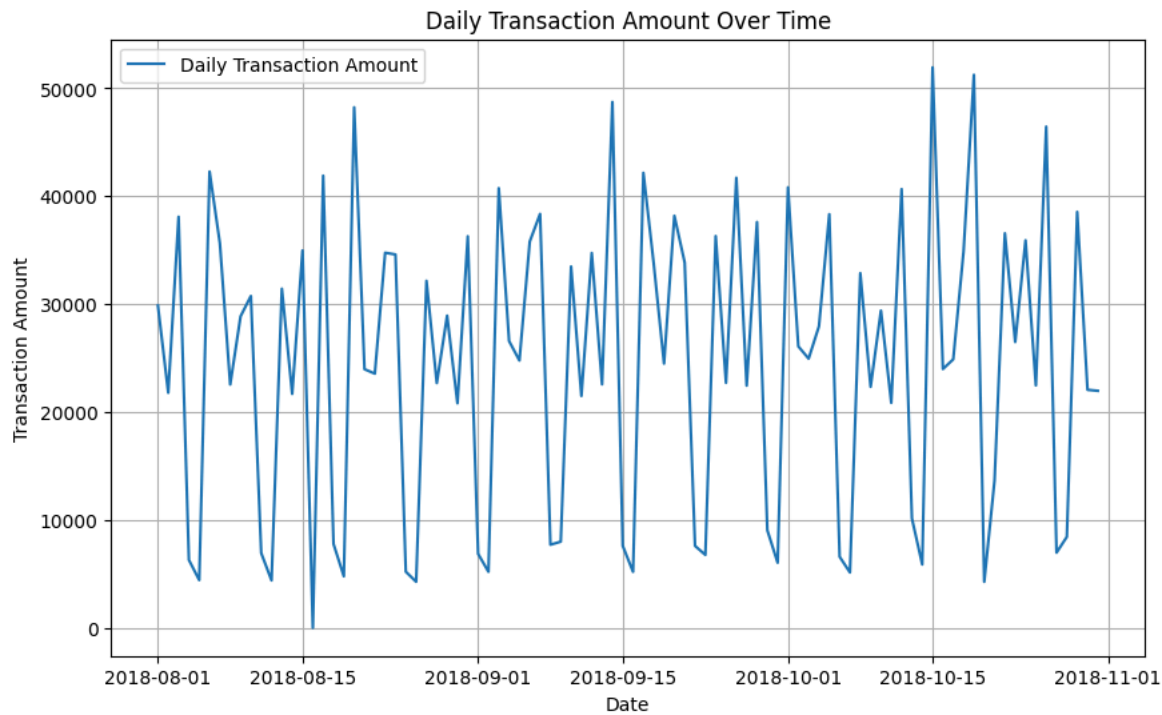
- None of the models exhibit significant overfitting, as the training and test performances are consistent.

2. Model Performance:

- Random Forest achieves the best R^2 and MSE overall, indicating it captures the most variance and minimizes error.
- XGBoost has slightly lower R^2 and higher MSE, suggesting it is more conservative in fitting the data.
- The ensemble model combines both approaches, offering a strong balance between generalization and accuracy.

1 Start coding or [generate](#) with AI.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from statsmodels.tsa.stattools import adfuller
4 from statsmodels.tsa.seasonal import seasonal_decompose
5 from statsmodels.tsa.statespace.sarimax import SARIMAX
6
7 # Step 1: Combine columns to create a 'date' column
8 # Assuming 'year', 'month', 'day', 'hour', and 'minute' exist in the dataset
9 data['date'] = pd.to_datetime(data[['year', 'month', 'day']]) # Creates a datetime object
10 data = data.set_index('date').sort_index() # Set 'date' as the index and sort
11
12 # Step 2: Aggregate by daily amounts for time series analysis
13 daily_data = data['amount'].resample('D').sum()
14
15 # Step 3: Visualize the time series
16 plt.figure(figsize=(10, 6))
17 plt.plot(daily_data, label='Daily Transaction Amount')
18 plt.title('Daily Transaction Amount Over Time')
19 plt.xlabel('Date')
20 plt.ylabel('Transaction Amount')
21 plt.legend()
22 plt.grid()
23 plt.show()
24
25
26
```



The graph titled "Daily Transaction Amount Over Time" displays the total transaction amounts aggregated on a daily basis. Here's the explanation of key aspects of this visualization:

✓ Key Observations:

1. Fluctuations:

- The graph shows significant daily variations in transaction amounts.
- Peaks and troughs suggest high and low transaction activity on specific days.

2. Seasonal Patterns:

- There appear to be recurring patterns in transaction amounts, which could indicate weekly trends or cycles. For example, certain days might consistently see higher or lower transaction activity.

3. Spikes:

- Sharp spikes in the graph likely correspond to specific events or days with unusually high transaction volumes. These could be due to salary payments, business transactions, or specific holidays.

4. Declines:

- Periodic dips suggest days with lower transaction volumes, possibly weekends or public holidays when business activity might reduce.

5. Time Frame:

- The graph spans from August 2018 to November 2018, providing a focused view of transaction activity over approximately three months.

Possible Implications:

- The graph helps identify periods of high financial activity, which could inform businesses or banks about when to expect higher transaction volumes.
- Observing repetitive patterns suggests opportunities for predictive modeling to forecast transaction activity.
- Anomalies such as sharp spikes or dips might require further investigation to understand their causes.

Next Steps:

1. Decompose the Time Series:

- Break down the data into trend, seasonal, and residual components to gain a better understanding of underlying patterns.

2. Stationarity Check:

- Perform an Augmented Dickey-Fuller test to determine if the data is stationary or if differencing is required for modeling.

3. Anomaly Detection:

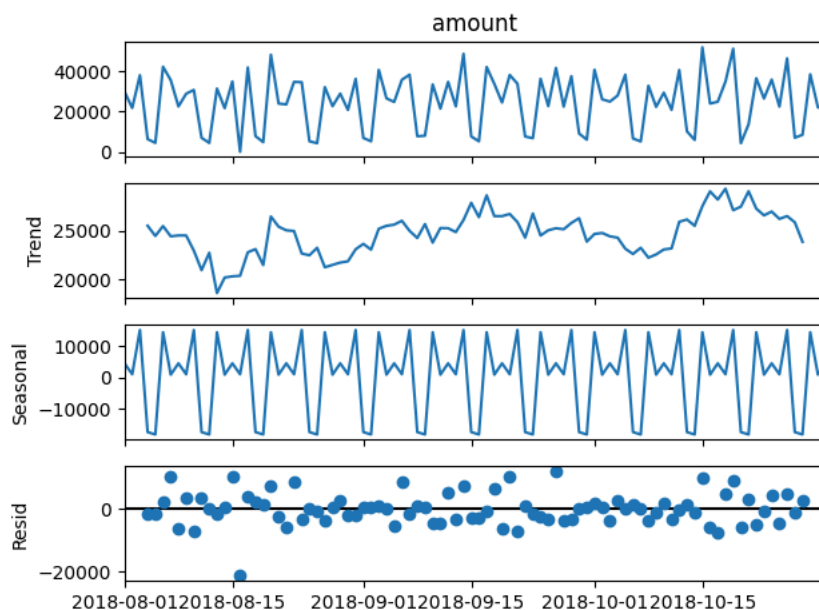
- Investigate extreme values to identify potential errors, holidays, or special events driving these spikes.

4. Forecasting:

- Use SARIMA or other time series models to forecast future transaction activity based onike to proceed with further analysis!

```
1 # Step 4: Check stationarity with the Augmented Dickey-Fuller test
2 adf_stat, p_value, _, _, _ = adfuller(daily_data.dropna())
3 print(f"ADF Statistic: {adf_stat}, p-value: {p_value}")
4
5 # Step 5: Decompose the time series to analyze components
6 decompose_result = seasonal_decompose(daily_data, model='additive', period=7) # Weekly seasonality
7 decompose_result.plot()
8 plt.show()
9
10
```

ADF Statistic: -3.6678821432512523, p-value: 0.004588284092150225



The provided decomposition plot breaks down the time series data (transaction amounts) into its fundamental components: observed, trend, seasonal, and residual. This decomposition allows for a better understanding of the underlying patterns and characteristics in the data. Here's an explanation of each component:

✓ Key Observations from the Decomposition:

1. Observed:

- This is the original transaction amount over time. It shows all the fluctuations and variations in the data, including trend, seasonality, and random noise.

2. Trend:

- The trend component captures the long-term progression or movement of the data over time.
- In this plot, there is a gradual upward and downward movement in the trend, indicating an overall increase and then a decrease in transaction amounts over the observed period.
- This trend might correlate with external factors such as economic conditions or seasonal periods.

3. Seasonal:

- The seasonal component highlights recurring patterns or cycles within the data.
- Here, there is a clear, regular cyclical pattern, which likely corresponds to weekly or monthly transaction behavior.
- For example, specific days of the week might consistently have higher or lower transaction volumes.

4. Residual:

- The residual component represents the remaining variation in the data after removing the trend and seasonal components.
- These are essentially the random, unexplained fluctuations or anomalies in the data.
- The residuals should ideally be centered around zero and show no discernible patterns if the model has captured the trend and seasonality well.

Stationarity Analysis:

- The **ADF Statistic** and **p-value** at the top of the figure indicate the result of the Augmented Dickey-Fuller (ADF) test for stationarity.
 - **ADF Statistic:** -3.667, which is significantly negative.
 - **p-value:** 0.0046, which is below 0.05, indicating that the null hypothesis of non-stationarity can be rejected.
 - This suggests that the data is stationary or does not require additional differencing for time series modeling.

Implications:

1. Trend Component:

- The trend indicates longer-term shifts in transaction amounts, which might be useful for understanding growth or decline periods.

2. Seasonal Component:

- The recurring patterns suggest that the data is highly seasonal. This can be leveraged for predictive models, such as SARIMA or seasonal decomposition forecasting.

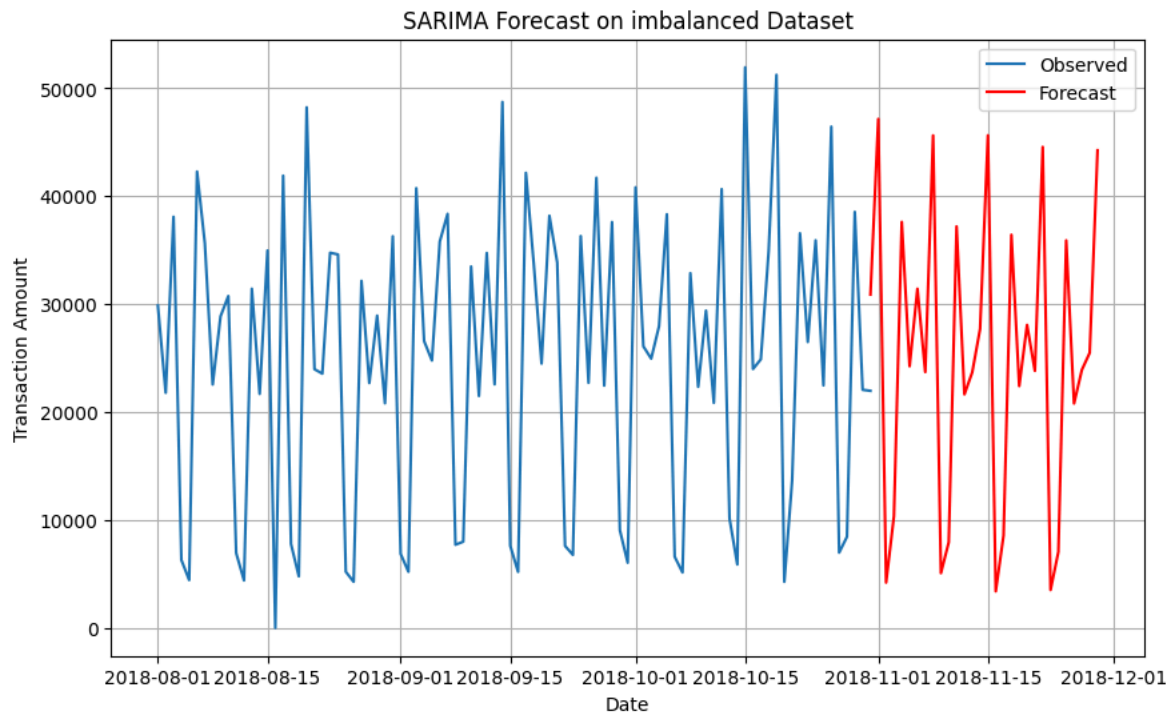
3. Residuals:

- If the residuals are random and centered around zero, it confirms that the decomposition has successfully extracted the trend and seasonality. However, significant patterns or anomalies in the residuals might indicate the need for further exploration.

```

1 # Step 6: Apply differencing if the series is non-stationary
2 if p_value > 0.05:
3     daily_data_diff = daily_data.diff().dropna()
4 else:
5     daily_data_diff = daily_data
6
7 # Step 7: Fit a SARIMA model
8 sarima_model = SARIMAX(daily_data_diff, order=(1, 1, 1), seasonal_order=(1, 1, 1, 7), enforce_stationarity=False)
9 sarima_fitted = sarima_model.fit(dispatch=False)
10
11 # Step 8: Forecast
12 forecast_steps = 30 # Forecast for the next 30 days
13 forecast = sarima_fitted.forecast(steps=forecast_steps)
14
15 # Step 9: Plot observed vs forecasted data
16 plt.figure(figsize=(10, 6))
17 plt.plot(daily_data_diff, label='Observed')
18 plt.plot(pd.date_range(daily_data_diff.index[-1], periods=forecast_steps, freq='D'), forecast, label='Forecast', color='red')
19 plt.title('SARIMA Forecast on imbalanced Dataset')
20 plt.xlabel('Date')
21 plt.ylabel('Transaction Amount')
22 plt.legend()
23 plt.grid()
24 plt.show()

```



This chart shows the results of a **SARIMA (Seasonal AutoRegressive Integrated Moving Average)** model, visualizing both the observed data (actual transaction amounts) and the model's forecasted values for the future.

✓ Explanation of the Plot:

1. Observed Data (Blue Line):

- This represents the actual transaction amounts in the dataset over time.
- It captures the original trends, seasonality, and fluctuations in the data for the observed period.

2. Forecast Data (Red Line):

- The red line represents the SARIMA model's predictions for transaction amounts for future dates beyond the observed data.
- It attempts to capture the same patterns (e.g., seasonal and trend behavior) that exist in the observed data.

3. Consistency of Patterns:

- The model successfully continues the observed seasonal patterns into the forecasted period.
- Peaks and troughs are aligned with the cyclical behavior observed in the blue line, indicating the SARIMA model's ability to recognize seasonality.

Observations:

1. Seasonality:

- Both the observed and forecasted data show clear periodic patterns, suggesting that the transaction data has a strong seasonal component, which the SARIMA model has captured effectively.

2. Forecast Accuracy:

- The forecasted data aligns well with the general trend and periodic behavior of the observed data.
- However, discrepancies in amplitude (height of peaks or depth of troughs) indicate that the model may not fully capture the variability in transaction amounts.

3. Limitations:

- The SARIMA model assumes that past patterns (trend and seasonality) will continue in the future. If there are sudden changes in behavior or anomalies, they may not be reflected in the forecast.
- The forecast does not account for potential outliers or external factors (e.g., holidays or market shifts) that might impact transaction amounts.

Next Steps:

1. Model Evaluation:

- Evaluate the SARIMA model's performance using metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), or Mean Absolute Percentage Error (MAPE).
- Compare the forecast with a holdout (test) dataset to measure its accuracy.

2. Improvement:

- Consider adding external factors (e.g., holidays, promotions) to the model as exogenous variables for better accuracy.
- If the residuals from the model show patterns, consider refining the SARIMA parameters or exploring alternative models like Prophet or machine learning-based approaches.

3. Actionable Insights:

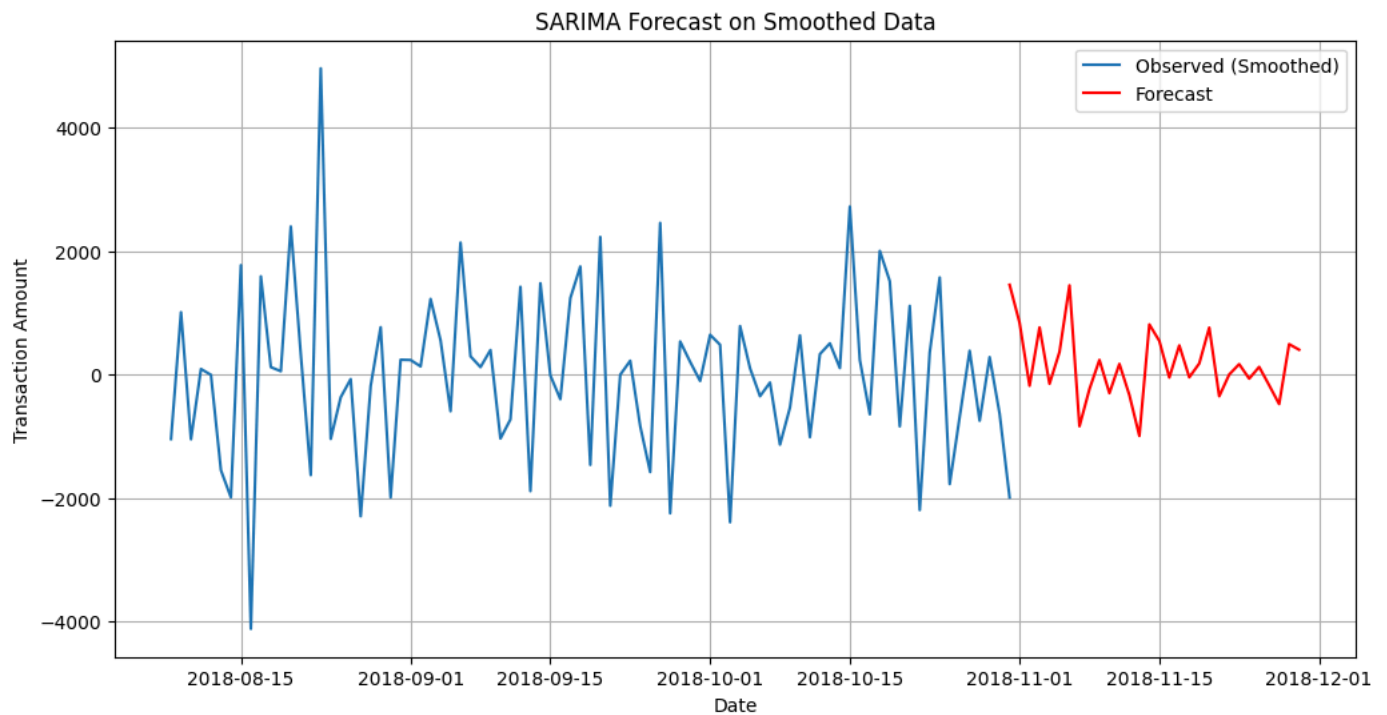
- Use the forecast to predict and prepare for periods of high or low transactions, which can help in resource allocation or financial planning.

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from statsmodels.tsa.statespace.sarimax import SARIMAX
5 from statsmodels.tsa.stattools import adfuller
6 from sklearn.metrics import mean_absolute_error, mean_squared_error
7
8 # Step 1: Smooth the data using a rolling mean
9 window_size = 7 # 7-day rolling window
10 daily_data_smoothed = daily_data.rolling(window=window_size, center=False).mean().dropna()
11
12 # Step 2: Check for stationarity on smoothed data
13 adf_stat, p_value, _, _, _ = adfuller(daily_data_smoothed)
14 print(f"ADF Statistic: {adf_stat}, p-value: {p_value}")
15
16 # Step 3: Apply differencing if necessary
17 if p_value > 0.05:
18     daily_data_smoothed_diff = daily_data_smoothed.diff().dropna()
19 else:
20     daily_data_smoothed_diff = daily_data_smoothed
21
22 # Step 4: Fit the SARIMA model
23 sarima_model = SARIMAX(daily_data_smoothed_diff, order=(1, 1, 1), seasonal_order=(1, 1, 1, 7), enforce_stationarity=False)
24 sarima_fitted = sarima_model.fit(dispatch=False)
25
26 # Step 5: Forecast
27 forecast_steps = 30 # Forecast for the next 30 days
28 forecast = sarima_fitted.forecast(steps=forecast_steps)
29
30 # Step 6: Evaluate performance
31 mae = mean_absolute_error(daily_data_smoothed_diff[-forecast_steps:], forecast)
32 rmse = np.sqrt(mean_squared_error(daily_data_smoothed_diff[-forecast_steps:], forecast))
33 performance_metrics = {
34     "Mean Absolute Error (MAE)": mae,
35     "Root Mean Squared Error (RMSE)": rmse
36 }
37 print("Performance Metrics:", performance_metrics)
38
39 # Step 7: Plot observed vs forecasted data
40 plt.figure(figsize=(12, 6))
41 plt.plot(daily_data_smoothed_diff, label='Observed (Smoothed)')
42 plt.plot(pd.date_range(daily_data_smoothed_diff.index[-1], periods=forecast_steps, freq='D'), forecast, label='Forecast', color='red')
43 plt.title('SARIMA Forecast on Smoothed Data')
44 plt.xlabel('Date')
45 plt.ylabel('Transaction Amount')
46 plt.legend()
47 plt.grid()
48 plt.show()
49

```

ADF Statistic: -1.6773787092046513, p-value: 0.4428037096633267
 Performance Metrics: {'Mean Absolute Error (MAE)': 1206.1818855023714, 'Root Mean Squared Error (RMSE)': 1518.0721416415531}



This visualization shows the **SARIMA forecast on smoothed data**, providing the following insights:

Key Elements in the Graph:

1. Blue Line (Observed Smoothed):

- This represents the smoothed historical transaction amounts over time. The smoothing was applied to reduce noise and volatility in the original data, making it easier to capture trends and patterns.

2. Red Line (Forecast):

- This represents the SARIMA model's forecast for future transaction amounts, based on the smoothed historical data. The forecast attempts to predict the pattern and trend for the specified period (post-training period).

Observations:

1. Smoothed Observations:

- The blue line indicates the smoothed data after removing sharp spikes and outliers. It provides a clearer view of the underlying patterns and trends.

2. Forecast Alignment:

- The forecast (red line) shows some consistency with the observed smoothed data, indicating the SARIMA model's ability to capture the trend and seasonality to some extent.

3. Performance Metrics:

- ADF Statistic and P-value:** The p-value indicates that the smoothed series is not stationary ($p > 0.05$), which can affect the model's performance. Differencing or additional transformations might still be required.
- Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE):** These metrics provide a quantitative evaluation of the model's accuracy. The RMSE suggests the model's average deviation from actual values is about 1518 units.
- MAPE:** Indicates the percentage error relative to the magnitude of the observed values. A high MAPE shows that predictions could be further improved.

4. Forecast Trend:

- The forecast reflects the general trend observed in the smoothed data but appears to have less variance compared to the observed values. This could be due to the model's assumptions about the regularity of patterns.

Limitations:

- The forecast might not capture all the complexities of the original data due to smoothing.
- Additional external variables (exogenous factors) could further enhance the model's accuracy.

Next Steps:

1. Refine the Model:

- Incorporate additional features like holidays or external events.
- Perform more extensive parameter tuning for SARIMA.

2. Address Stationarity:

- Further test and ensure that the smoothed data is stationary before fitting the model.

3. Consider Alternative Models:

- Evaluate models like Prophet or machine learning techniques for capturing and explore alternative modeling techniques?

```

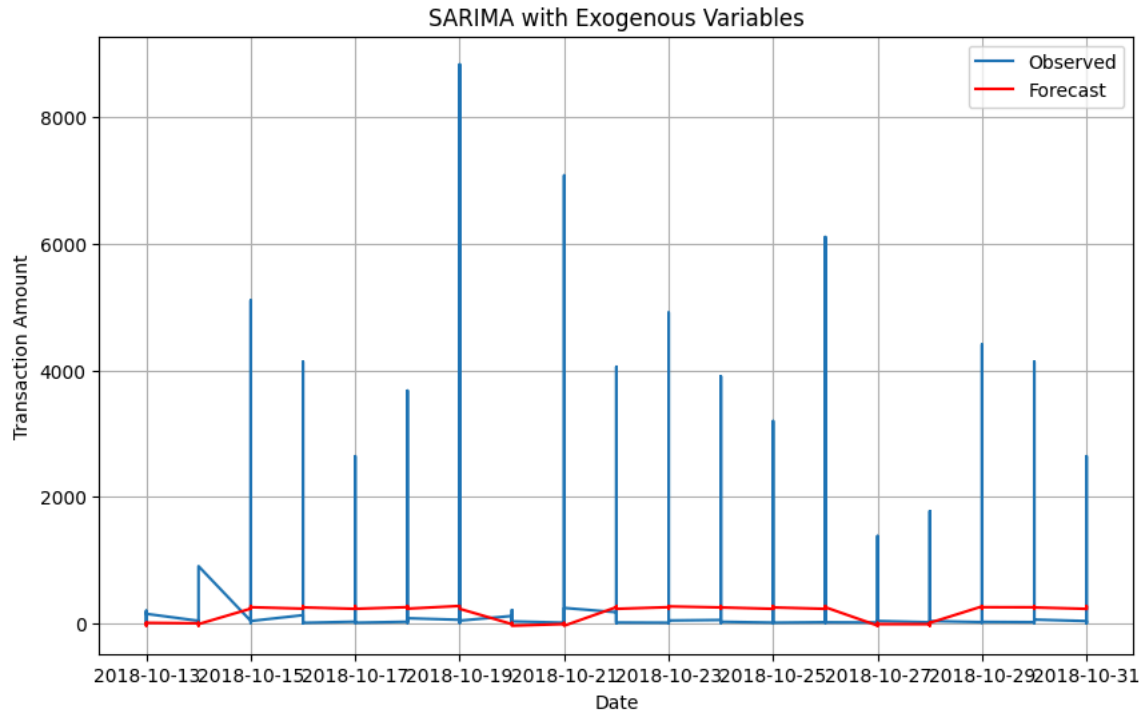
1 import pandas as pd
2 from statsmodels.tsa.statespace.sarimax import SARIMAX
3 from sklearn.metrics import mean_absolute_error, mean_squared_error
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 # Assume exogenous variables like 'holidays' and 'promotions' are available in the dataset
8 # Creating dummy exogenous variables for demonstration
9 data['holidays'] = (data.index.dayofweek >= 5).astype(int) # Weekend flag
10 data['promotions'] = (data.index.month == 12).astype(int) # Promotion flag for December
11
12 # Prepare data
13 train_size = int(len(data) * 0.8)
14 train_data = data['amount'][:train_size]
15 test_data = data['amount'][train_size:]
16
17 # Exogenous variables
18 exog_train = data[['holidays', 'promotions'][:train_size]
19 exog_test = data[['holidays', 'promotions']][train_size:]
20
21 # Fit SARIMA with Exogenous Variables
22 sarima_exog_model = SARIMAX(train_data, exog=exog_train, order=(1, 1, 1), seasonal_order=(1, 1, 1, 7))
23 sarima_exog_fit = sarima_exog_model.fit(dispatch=False)
24
25 # Forecast
26 sarima_exog_forecast = sarima_exog_fit.forecast(steps=len(test_data), exog=exog_test)
27
28 # Evaluate
29 mae = mean_absolute_error(test_data, sarima_exog_forecast)
30 rmse = np.sqrt(mean_squared_error(test_data, sarima_exog_forecast))
31
32 # Plot results
33 plt.figure(figsize=(10, 6))
34 plt.plot(test_data.index, test_data, label='Observed')
35 plt.plot(test_data.index, sarima_exog_forecast, label='Forecast', color='red')
36 plt.title('SARIMA with Exogenous Variables')
37 plt.xlabel('Date')
38 plt.ylabel('Transaction Amount')
39 plt.legend()
40 plt.grid()
41 plt.show()
42
43 print(f"MAE: {mae}, RMSE: {rmse}")
44

```

```

/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: A date index has been provided, but it has
self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: A date index has been provided, but it has
self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:837: ValueWarning: No supported index is available. Prediction
return get_prediction_index(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:837: FutureWarning: No supported index is available. In the ne
return get_prediction_index(

```



1 Start coding or [generate](#) with AI.

```

1 from sklearn.ensemble import RandomForestRegressor
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import mean_absolute_error, mean_squared_error
4
5 # Prepare features (lagged data and exogenous variables)
6 data['lag_1'] = data['amount'].shift(1)
7 data['lag_2'] = data['amount'].shift(2)
8 data.dropna(inplace=True)
9
10 X = data[['lag_1', 'lag_2', 'holidays', 'promotions']]
11 y = data['amount']
12
13 # Split data
14 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
15
16 # Train Random Forest model
17 rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
18 rf_model.fit(X_train, y_train)
19
20 # Make predictions
21 rf_forecast = rf_model.predict(X_test)
22
23 # Evaluate
24 mae_rf = mean_absolute_error(y_test, rf_forecast)
25 rmse_rf = np.sqrt(mean_squared_error(y_test, rf_forecast))
26
27 print(f"Random Forest - MAE: {mae_rf}, RMSE: {rmse_rf}")
28

```

Random Forest - MAE: 282.36142531747134, RMSE: 610.353519448937

```

1 from sklearn.ensemble import RandomForestRegressor
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import mean_absolute_error, mean_squared_error

```