IIITDM
Kancheepuram

(Applied Data Science)

Daily life transaction Dataset

Krishna
Roll no: CS23B2019

Vijay
Roll no: CS23B2017

Varshith B
Roll no: CS23B2015

Sriram B
Roll no: CS23B2032

Project 16

November 10, 2024

Contents

Introduction

This report analyzes a comprehensive dataset on daily financial transactions, detailing spending patterns across various categories, such as groceries, utilities, and entertainment, as well as payment methods like credit cards, cash, and online transfers. By exploring data preprocessing steps, this analysis ensures high data quality, removing duplicates, filling missing values, and encoding categorical data for compatibility with statistical models.

Normalization techniques, including Z-score and Min-Max normalization, are applied to standardize data, ensuring uniformity across different scales. Feature selection, identifies significant factors that most influence spending behavior, making the dataset more manageable and relevant.

For insights, the report leverages visualization techniques—bar graphs to compare expenses across payment methods, pie charts for category spending distribution, and scatter plots to identify correlations. These visualizations reveal spending habits, high-frequency categories, and potential trends.

The analysis emphasizes the importance of data preparation and visualization in uncovering actionable insights. Future research could explore predictive models to forecast or flag unusual spending patterns, potentially aiding budgeting and financial planning.

Data Preprocessing

Data preprocessing ensures data quality for analysis. This project involved loading the dataset, handling missing values, and normalizing the data.

2.1 Loading and Cleaning Data

The dataset was loaded and cleaned with basic operations. Below is a Python code snippet showing data loading:

```
# Load the CSV file
data = pd.read_csv('/content/ADS_DATASET.csv')

data.head(10)
```

Figure 2.1: Loading the dataset.

2.2 Normalization Techniques

Normalization standardizes data for consistency. Techniques used include:

- **Z-score Normalization**: Standardizes data with a mean of zero.
- **Min-Max Normalization**: Scales data to a range of [0, 1].
- **Decimal scaling normalization**: Transforms data by dividing each value by a power of 10 to ensure that the maximum absolute value is less than 1.

```
# 1. Z-score Normalization
z_score_normalized = (numerical_data - numerical_data.mean()) / numerical_data.std()
print("Z-Score Normalized Data:")
print(z_score_normalized.head())
```

(a) Z-score snippet

```
Z-Score Normalized Data:
    Amount
0 -0.217351
1 -0.214954
2 -0.203852
3 -0.218229
4 -0.199698
```

(b) Z-score output

Figure 2.2: Z-Score Normalization.

2.3 Feature Selection

Feature selection was performed by scaling numerical features using StandardScaler. It also selects the top 5 features with the highest significance using ANOVA F-test through SelectKBest. If a target variable exists, it applies feature selection; otherwise, it skips that

step. The transformed data is then displayed, helping prepare the dataset for machine learning tasks.

```python
# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X.select_dtypes(include=['float64', 'int64']))  # Only scale numerical features
# 2. Feature Selection using SelectKBest (if there's a target variable)
if y is not None:
    select_k_best = SelectKBest(score_func=f_classif, k=5)  # Select top 5 features
    X_selected = select_k_best.fit_transform(X_scaled, y)
    selected_features = X.columns[select_k_best.get_support(indices=True)]
    selected_df = pd.DataFrame(X_selected, columns=selected_features)
    print("\nSelected Features based on ANOVA F-value:")
    print(selected_df.head())
else:
    print("\nNo target variable found. Skipping feature selection.")
```

(a) Top image caption

```
No target variable found. Skipping feature selection.
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWar
  and should_run_async(code)
```

(b) Bottom image caption

Figure 2.3: Two images stacked vertically.

Data Visualization Techniques

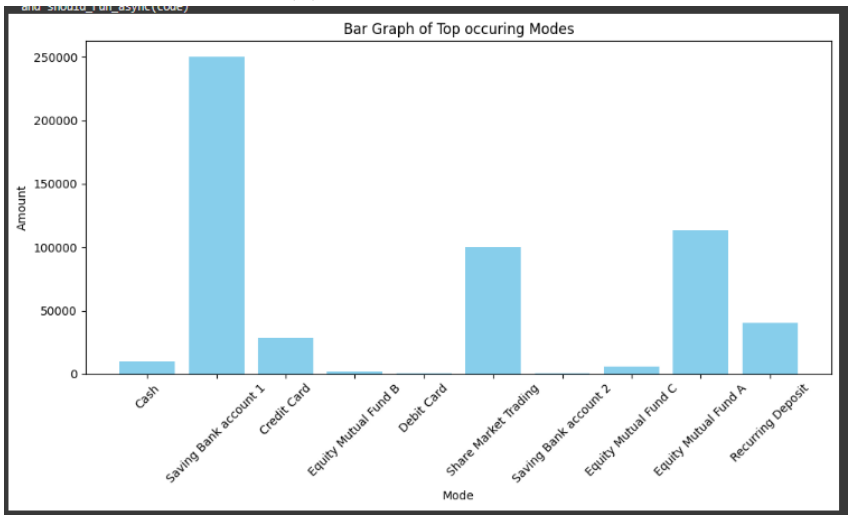
Data visualizations are essential for understanding spending patterns across different categories and payment modes. By representing data visually, trends, outliers, and relationships become much clearer, facilitating better insights.

3.1 Bar Graph

This bar graph is created with "Mode" on the x-axis and "Amount" on the y-axis. The graph provides a visual comparison of the amounts spent across the top modes, allowing for easy identification of major spending channels.

```
# Plot the bar graph
plt.figure(figsize=(10, 6))
plt.bar(filtered_data['Mode'], filtered_data['Amount'], color='skyblue')
plt.xlabel('Mode')
plt.ylabel('Amount')
plt.title('Bar Graph of Top occuring Modes')
plt.xticks(rotation=45)
plt.tight_layout()  # Adjust layout to prevent overlap
```

(a) plot the bar graph



(b) Bar graph visualization

Figure 3.1: Bar graph.

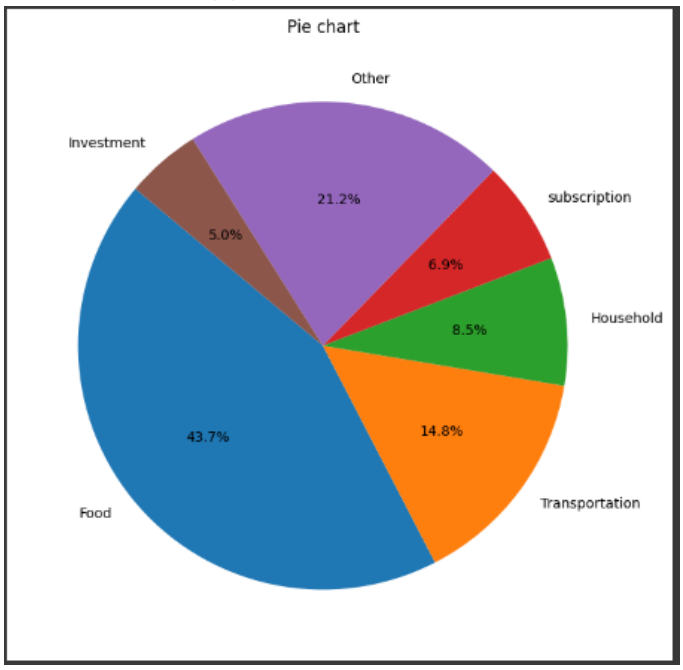
The bar graph shows that Saving Bank Account 1 holds the highest amount, followed by Equity Mutual Fund C and Share Market Trading. Recurring Deposits and Credit Cards have moderate amounts, while Cash, Equity Mutual Fund B, Debit Card, and Saving Bank Account 2 hold smaller amounts. Overall, bank accounts and investments are the preferred modes over cash and credit options.

3.2 Pie Chart

This pie chart is generated for the top 6 categories, including an "Other" category representing the sum of remaining categories. The chart displays the proportions of each category in percentages.

```
# Plot pie chart for the top 10 + "Other" category
plt.figure(figsize=(8, 8))
plt.pie(top_10, labels=top_10.index, autopct='%1.1f%%', startangle=140)
plt.title('Pie chart')
plt.show()
```

(a) plot the Pie chart



(b) Pie chart visualization

Figure 3.2: Pie chart.

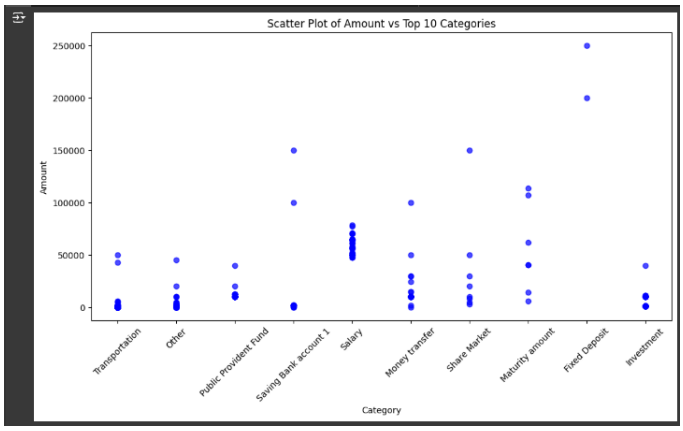
The pie chart shows the distribution of expenses across various categories. The largest portion, 43.7%, is spent on Food, followed by 21.2% on Other expenses and 14.8% on Transportation. Household and Subscription expenses account for 8.5% and 6.9%, respectively, while Investments make up the smallest share at 5.0%.

3.3 Scatter Plot

This scatter plot is created using Matplotlib where the x-axis represents Category and the y-axis represents Amount. The plot uses blue points with some transparency to improve visual clarity. The x-axis labels are rotated by 45 degrees to make them more readable. The scatter plot visually represents how the Amount varies across the Top 10 Categories, allowing you to easily spot any patterns or outliers.

```
# Create scatter plot
plt.figure(figsize=(12, 6))
plt.scatter(top_data['Category'], top_data['Amount'], alpha=0.7, color='blue')
plt.xlabel('Category')
plt.ylabel('Amount')
plt.title('Scatter Plot of Amount vs Top 10 Categories')
plt.xticks(rotation=45)  # Rotate x-axis labels for readability
plt.show()
```

(a) plot the scatter plot



(b) scatter plot visualization

Figure 3.3: scatter plot.

The scatter plot shows the distribution of amounts across the top 10 categories. Fixed Deposit and Salary have the highest amounts, with values reaching over 200,000 units. Other notable categories with varying amounts include Saving Bank Account 1, Public Provident Fund, and Share Market. Lower amounts are observed in categories like Transportation, Other, and Money Transfer.

3.4 Histogram

This code loads a dataset from a CSV file and extracts the first 10 rows. It then displays these rows and generates a histogram for the 'Amount' column of the top 10 entries, showing the frequency distribution of values with 15 bins. The histogram is customized with a title and axis labels, and it is displayed to visualize the spread of the 'Amount' values in the selected data.

```
# Assuming you want to plot a histogram of a specific column, replace 'column_name' with the actual column name
top_10_rows['Amount'].plot(kind='hist', bins=15, edgecolor='black')

# Customize the plot
plt.title('Histogram of Amount(Top 10 rows)')
plt.xlabel('Amount')
plt.ylabel('Frequency')

# Show the plot
plt.show()
```

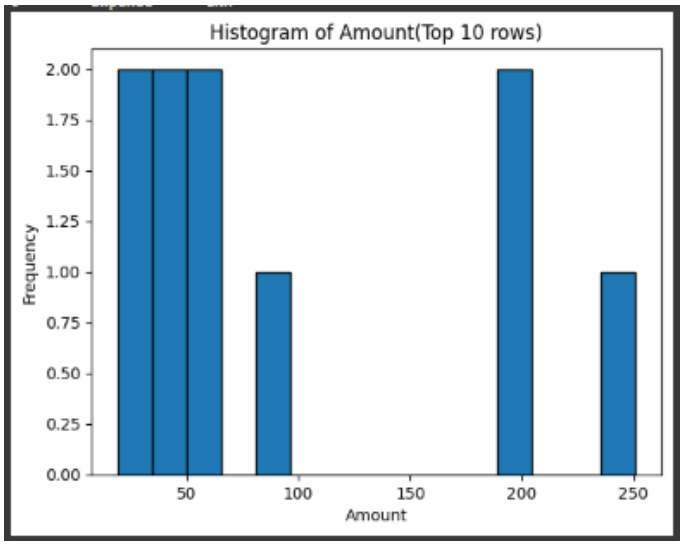
(a) plot the Histogram.

	Date	Mode	Category
0	20-09-2018 12:04	Cash	Transportation
1	20-09-2018 12:03	Cash	Food
2	19-09-2018	Saving Bank account 1	subscription
3	17-09-2018 23:41	Saving Bank account 1	subscription
4	16-09-2018 17:15	Cash	Festivals
5	15-09-2018 06:34	Credit Card	subscription
6	14-09-2018 05:39	Cash	Transportation
7	13-09-2018 21:35	Saving Bank account 1	Transportation
8	13-09-2018 21:01	Credit Card	Other
9	13-09-2018 21:01	Cash	Food

(b) Top 10 data

	Subcategory	Note	Amount
0	Train	2 Place 5 to Place 0	30.0
1	snacks	Idli medu Vada mix 2 plates	60.0
2	Netflix	1 month subscription	199.0
3	Mobile Service Provider	Data booster pack	19.0
4	Ganesh Pujan	Ganesh idol	251.0
5	Tata Sky	Permanent Residence - Tata Play recharge	200.0
6	auto	Place 2 station to Permanent Residence	50.0
7	Train	2 Place 0 to Place 3	40.0
8	NaN	HBR 2 Months subscription	83.0
9	Grocery	1kg atta	46.0

(c) Histogram visualization



(d) Histogram

Figure 3.4: Histogram.

The histogram of the "Amount" column for the top 10 rows highlights that the data has a non-uniform distribution. There are notable peaks around the 50 and 200 values, each with a frequency of 2, indicating these values are more common within the sample. Fewer entries are observed around 100 and 250, with a frequency of 1 for each. This pattern may suggest underlying groupings or categories within the data, pointing to possible clusters or common intervals among the recorded amounts. This distribution could imply that certain amounts are repeated or are significant within this subset.

3.5 Box Plot

This code generates a box plot for the 'Amount' column of the dataset, which provides a visual summary of the data distribution, highlighting the median, interquartile range,

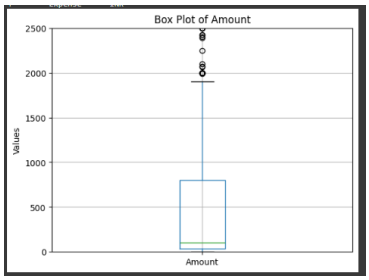
and potential outliers. It customizes the plot by adding a title, labeling the y-axis as "Values," and setting the y-axis range between 0 and 2500 to focus on the relevant data range. The box plot helps to identify the spread of the data, any skewness, and any extreme values (outliers). After customization, the plot is displayed to offer insights into the data's variability and central tendency.

```
# Create the box plot for 'Amount' column
df.boxplot(column='Amount')

# Customize the plot
plt.title('Box Plot of Amount')
plt.ylabel('Values')

# Set the range for the y-axis (replace with your desired min and max values)
plt.ylim(0,2500)  # Example range from 0 to 1000
```

(a) Box plot snippet.

	Date	Mode	Category	Subcategory	Note	Amount	Income/Expense	Currency
0	20-09-2018 12:04	Cash	Transportation	Train	2 Place 5 to Place 0	30.0	Expense	INR
1	20-09-2018 12:03	Cash	Food	snacks	Idli medu Vada mix 2 plates	60.0	Expense	INR
2	19-09-2018	Saving Bank account 1	subscription	Netflix	1 month subscription	199.0	Expense	INR
3	17-09-2018 23:41	Saving Bank account 1	subscription	Mobile Service Provider	Data booster pack	19.0	Expense	INR
4	16-09-2018 17:15	Cash	Festivals	Ganesh Pujan	Ganesh idol	251.0	Expense	INR

(b) Box plot output



(c) Box plot for amounts

Figure 3.5: Box plot.

The box plot of the "Amount" data shows a skewed distribution with a large number of outliers above 2000. The main data points are concentrated below 500, as indicated by the interquartile range (IQR) box, with the median close to the lower end. The presence of outliers suggests that there are several significantly high values, which pull the distribution upwards, making the data highly variable. This spread and the clustering of most values below 500 highlight the skewed nature of the dataset.

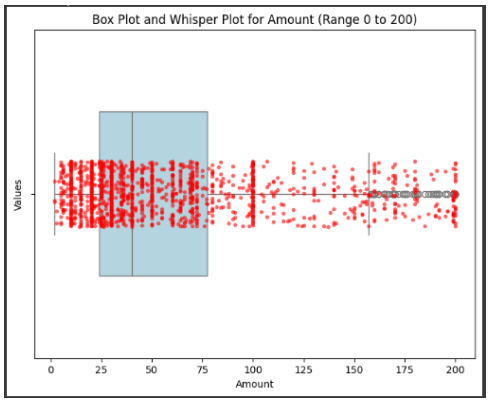
The box and whisker plot for "Amount" in the range of 0 to 200 shows a dense concentration of values around 25 to 75, with the majority of data points falling within this interquartile range. There is a scattering of values throughout the range up to 200, with a few outliers appearing near the upper end. The spread and clustering indicate that while most amounts are low to moderate, there is some variability with occasional higher values extending closer to the 200 mark.

```python
# Filter the dataframe to include only values within the desired range
filtered_df = df[(df['Amount'] >= min_value) & (df['Amount'] <= max_value)]

# Create the box plot for 'Amount' column within the specified range
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size
sns.boxplot(data=filtered_df, x='Amount', color='lightblue', width=0.5)

# Add whisper plot (strip plot) to show individual points
sns.stripplot(data=filtered_df, x='Amount', color='red', jitter=True, size=4, alpha=0.6)

# Customize the plot
plt.title(f'Box Plot and Whisper Plot for Amount (Range {min_value} to {max_value})')
plt.xlabel('Amount')
plt.ylabel('Values')
```

(a) Box plot snippet.

	Date	Mode	Category	Subcategory	Note	Amount	Income/Expense	Currency
0	20-09-2018 12:04	Cash	Transportation	Train	2 Place 5 to Place 0	30.0	Expense	INR
1	20-09-2018 12:03	Cash	Food	snacks	Idli medu Vada mix 2 plates	60.0	Expense	INR
2	19-09-2018	Saving Bank account 1	subscription	Netflix	1 month subscription	199.0	Expense	INR
3	17-09-2018 23:41	Saving Bank account 1	subscription	Mobile Service Provider	Data booster pack	19.0	Expense	INR
4	16-09-2018 17:15	Cash	Festivals	Ganesh Pujan	Ganesh idol	251.0	Expense	INR

(b) Box plot output



(c) Box plot and whisker plot for amounts

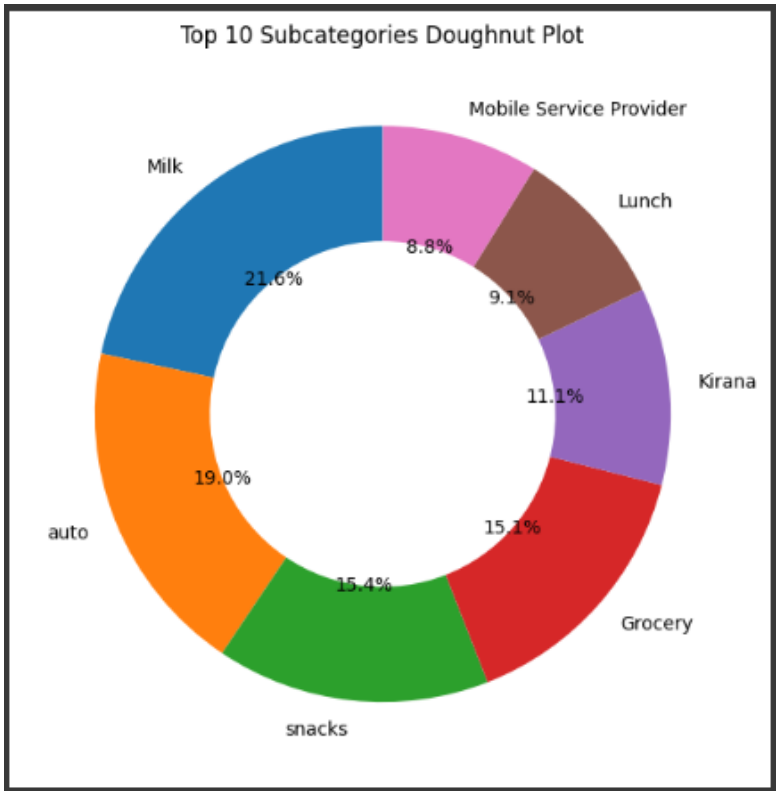
Figure 3.6: Box plot.

3.6 Doughnut Plot

This code counts the occurrences of each 'Subcategory' in the dataset, selects the top 7, and creates a doughnut plot to visualize their distribution. The plot displays the percentage share of each subcategory, providing a clear visual representation of the most common subcategories. It is customized with a title and designed with a 40% hole in the center for a doughnut-style appearance. This visualization helps identify the dominant subcategories in the dataset.

```python
# Select the top 10 subcategories
top_10_subcategories = subcategory_counts.head(7)

# Plotting the doughnut plot
plt.figure(figsize=(7, 7))
plt.pie(top_10_subcategories, labels=top_10_subcategories.index, autopct='%1.1f%%', startangle=90, wedgeprops={'width': 0.4})
plt.title('Top 10 Subcategories Doughnut Plot')
plt.show()
```

(a) Doughnut plot snippet



(b) Doughnut plot visualization

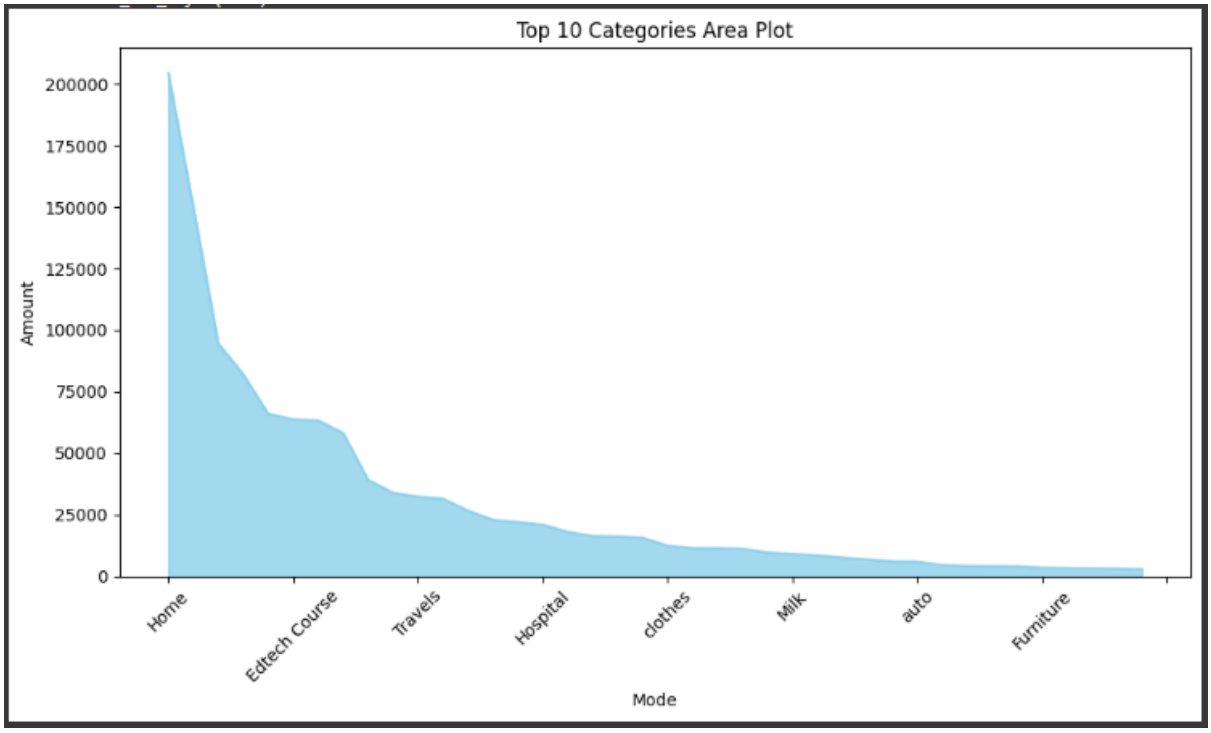
Figure 3.7: Doughnut plot.

3.7 Area Plot

This code groups the data by 'Subcategory' and calculates the sum of the 'Amount' for each subcategory, selecting the top 40 based on the highest sums. It then creates an area plot to visualize the cumulative values of these top subcategories over their index. The plot is customized with a title, labeled axes, and rotated x-axis labels for better readability. The area under the plot is filled with a light blue color to enhance the visual effect. Finally, the plot is displayed with a tight layout to ensure everything fits well. This visualization highlights the contribution of each subcategory to the total amount.

```
# Plot the area plot
plt.figure(figsize=(10, 6))
top_10.plot(kind='area', color='skyblue', alpha=0.7)
plt.title('Top 10 Categories Area Plot')
plt.xlabel('Mode')
plt.ylabel('Amount')
plt.xticks(rotation=45)

# Fill the area between the lines
plt.gca().fill_between(top_10.index, top_10.values, color='skyblue', alpha=0.2)
```

(a) Area plot snippet



(b) Area plot visualization

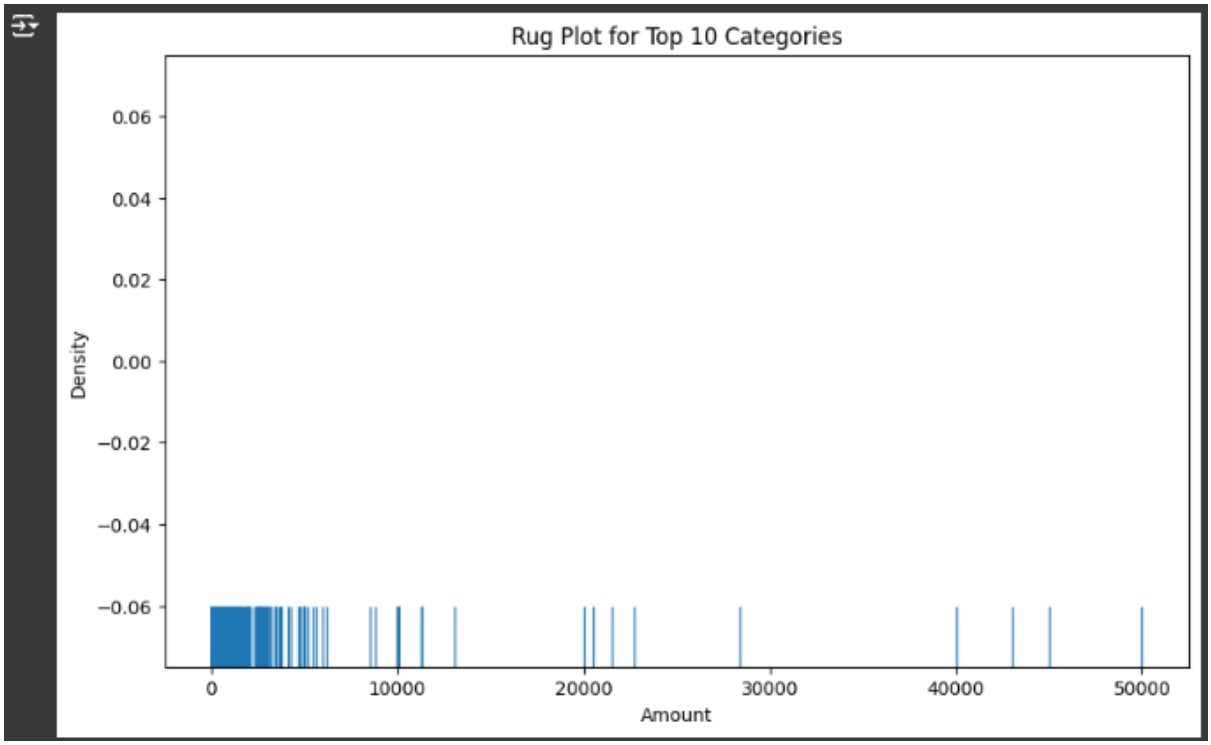
Figure 3.8: Area plot.

3.8 Rug Plot

This code groups the data by 'Subcategory' and calculates the sum of the 'Amount' for each subcategory, selecting the top 40 based on the highest sums. It then creates an area plot to visualize the cumulative values of these top subcategories over their index. The plot is customized with a title, labeled axes, and rotated x-axis labels for better readability. The area under the plot is filled with a light blue color to enhance the visual effect. Finally, the plot is displayed with a tight layout to ensure everything fits well. This visualization highlights the contribution of each subcategory to the total amount.

```
# Filter the DataFrame to include only the top 10 categories
top_10_data = data[data['Category'].isin(top_10_categories)]

# Plotting the rug plot
plt.figure(figsize=(10, 6))
sns.rugplot(x='Amount', data=top_10_data, height=0.1)  # Adjust 'Amount' to the numeric column you want to analyze

# Set the x-axis range (adjust these values as needed)
#plt.xlim(0, 10000)  # Replace 0 and 1000 with your desired lower and upper limits

plt.title('Rug Plot for Top 10 Categories')
plt.xlabel('Amount')
plt.ylabel('Density')
```

(a) Rug plot snippet



(b) Rug plot visualization

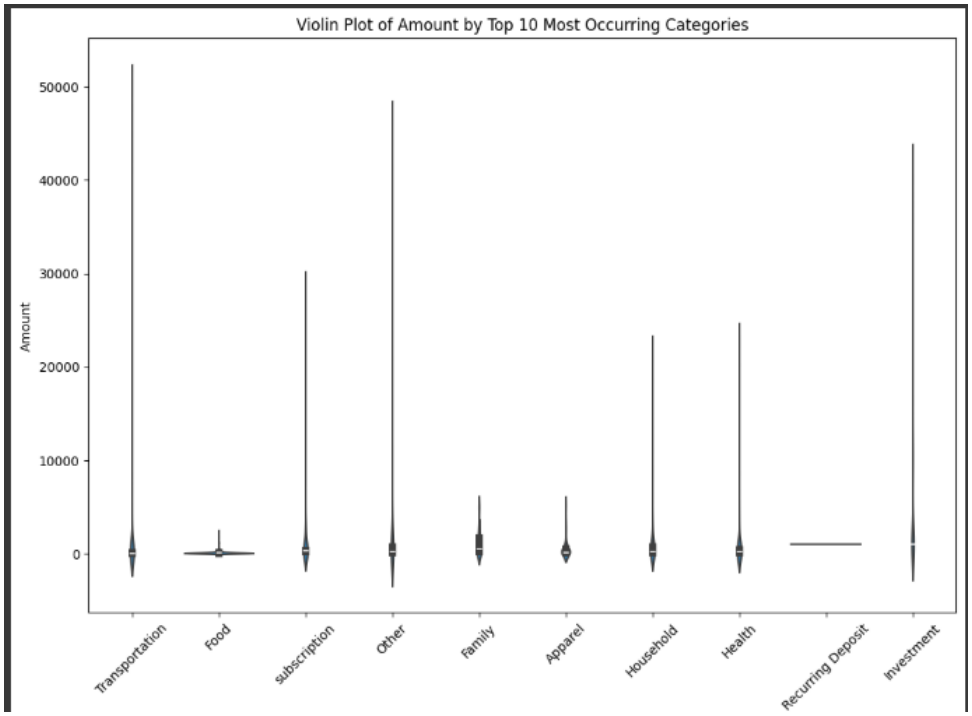
Figure 3.9: Rug plot.

3.9 Violin Plot

This code identifies the top 10 most frequent categories in the 'Category' column and filters the dataset to include only rows corresponding to those categories. A violin plot is then created to visualize the distribution of the 'Amount' column across these top categories. The plot shows the range, median, and density of 'Amount' values for each category, providing a detailed view of the data distribution. The plot is customized with a title, axis labels, and rotated category labels for better readability. This visualization helps compare the distribution of 'Amount' across the most common categories.

```
# Plotting the violin plot for the top 10 categories
plt.figure(figsize=(12, 8))
sns.violinplot(x='Category', y='Amount', data=top_10_data)

# Add labels and title
plt.title('Violin Plot of Amount by Top 10 Most Occurring Categories')
plt.xlabel('Category')
plt.ylabel('Amount')
```

(a) Violin plot snippet



(b) Violin plot visualization

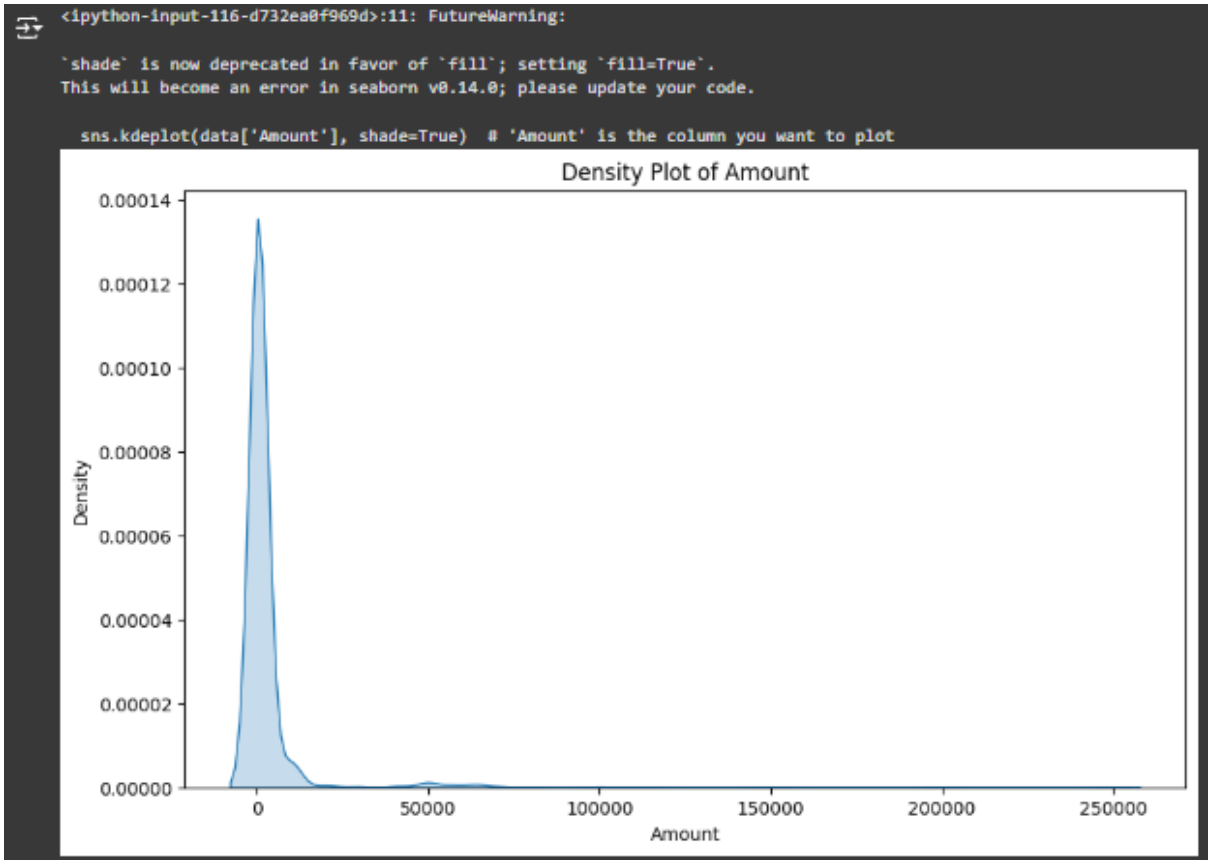
Figure 3.10: Violin plot.

3.10 Density Plot

This code identifies the top 10 most frequent This code generates a kernel density estimate (KDE) plot for the 'Amount' column, which visualizes the distribution of the 'Amount' values as a smooth curve. The plot is shaded to highlight the density areas, giving a clear view of how the data is spread. The plot includes a title, and labels for the x-axis (Amount) and y-axis (Density). This visualization helps in understanding the overall distribution and concentration of 'Amount' values across the dataset.

```
# Plotting the density plot
# Replace 'Amount' with the relevant numeric column in your CSV
plt.figure(figsize=(10, 6))
sns.kdeplot(data['Amount'], shade=True)  # 'Amount' is the column you want to plot

# Add labels and title
plt.title('Density Plot of Amount')
plt.xlabel('Amount')
plt.ylabel('Density')
```

(a) Density plot snippet



(b) Density plot visualization

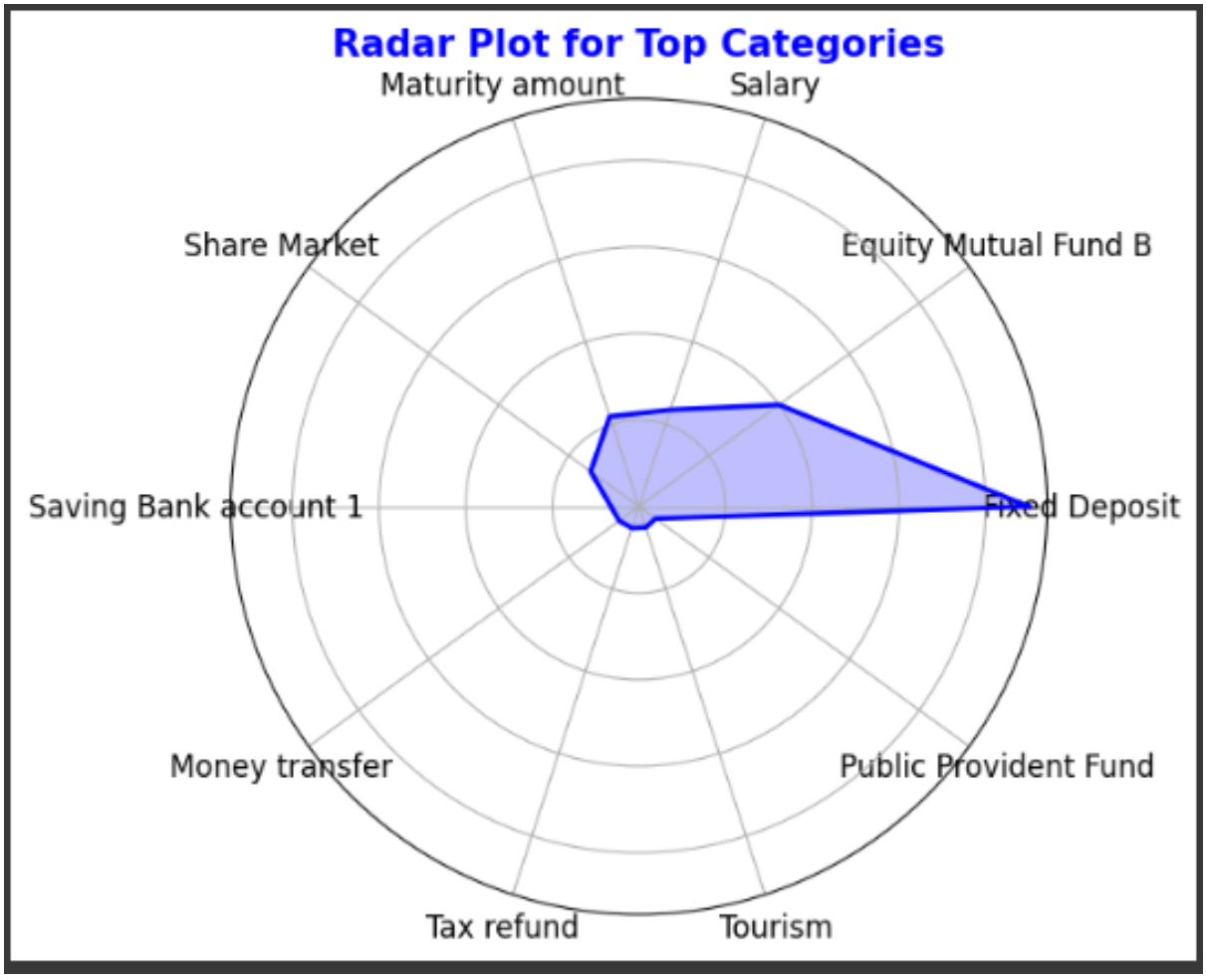
Figure 3.11: Density plot.

3.11 Radar Plot

This code generates a radar plot to visualize the top 10 categories with the highest average 'Amount'. It first calculates the mean 'Amount' for each category, selects the top 10, and prepares the data for plotting. The radar plot is set up with angles corresponding to each category, and the values are plotted around the circle, with the first value repeated to close the circle. The plot is filled with a transparent blue color, and a line connects the data points for each category. The category labels are displayed around the plot, and the radial ticks are removed for better clarity. This visualization provides a clear comparative view of the top categories, making it easy to identify which categories have higher average amounts and how they relate to each other.

```python
# Sort the data if needed, for example, select top 5 categories
top_categories = category_data.nlargest(10).index
top_category_data = category_data[top_categories]

# Number of categories (this is the number of axes for the radar plot)
categories = top_category_data.index
values = top_category_data.values

# Set up the radar plot
angles = np.linspace(0, 2 * np.pi, len(categories), endpoint=False).tolist()

# The radar plot needs to close, so repeat the first value to close the circle
values = np.concatenate((values, [values[0]]))
angles += angles[:1]

# Create the radar plot
fig, ax = plt.subplots(figsize=(6, 6), dpi=100, subplot_kw=dict(polar=True))
ax.fill(angles, values, color='blue', alpha=0.25)
ax.plot(angles, values, color='blue', linewidth=2)  # Line connecting the points

# Set the labels
ax.set_yticklabels([])  # Remove the radial ticks if you want
ax.set_xticks(angles[:-1])  # Remove the last value for x-ticks, which is a repeat
ax.set_xticklabels(categories, fontsize=12)
```

(a) Radar plot snippet



(b) Radar plot visualization

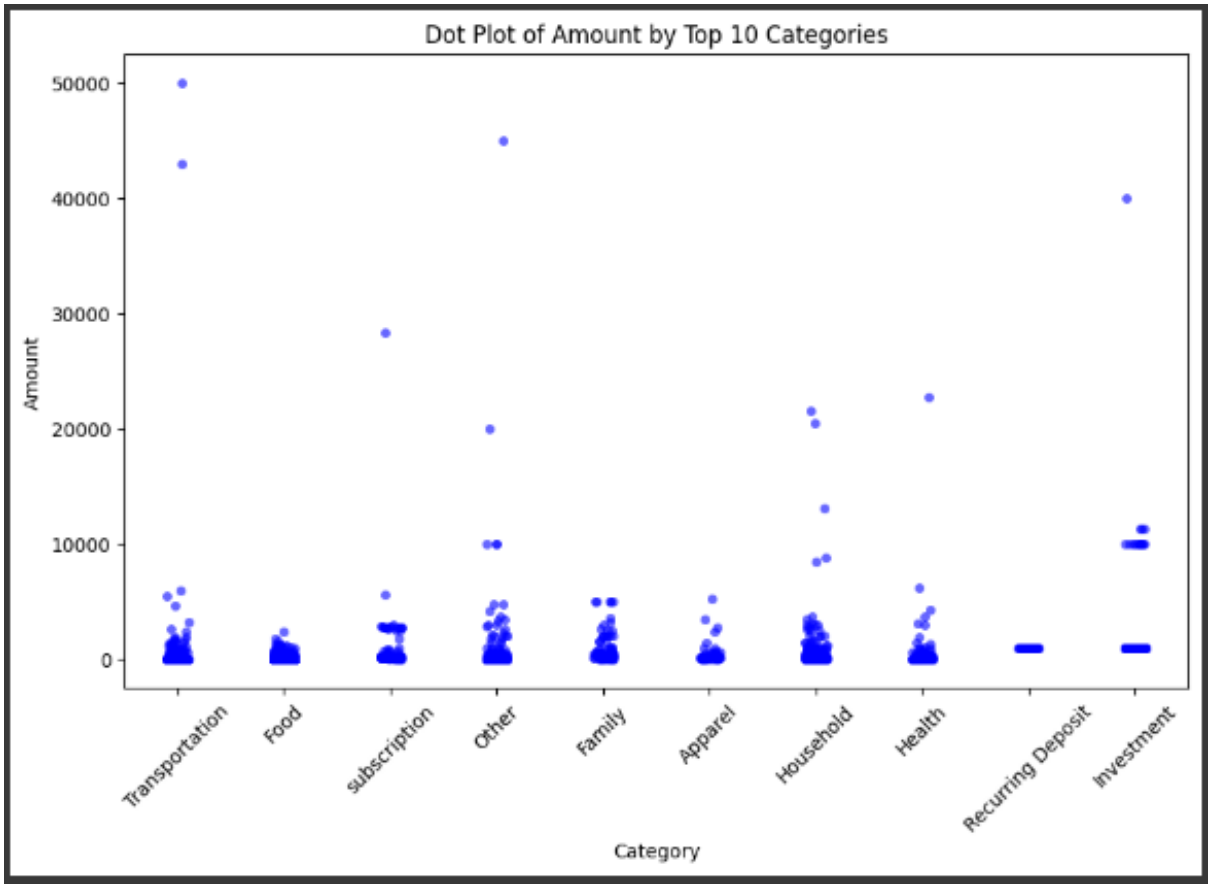
Figure 3.12: Radar plot.

3.12 Dot Plot

This code generates a dot plot to visualize the distribution of 'Amount' values for the top 10 most frequent categories. It first identifies the top 10 categories based on their frequency in the 'Category' column and filters the dataset accordingly. A strip plot (dot plot) is then created, where each data point is represented by a dot. The dots are jittered to avoid overlap, making the distribution clearer. The plot is customized with a title, labels for the x and y axes, and rotated x-axis labels for better readability. This visualization helps to see the spread and density of 'Amount' values within the top categories.

```
# Filter the data to include only the top categories
top_category_data = data[data['Category'].isin(top_categories)]

# Create a dot plot for the top categories
plt.figure(figsize=(10, 6))
sns.stripplot(x='Category', y='Amount', data=top_category_data, jitter=True, color='blue', alpha=0.6)

# Add labels and title
plt.title('Dot Plot of Amount by Top 10 Categories')
plt.xlabel('Category')
plt.ylabel('Amount')
```

(a) Dot plot snippet



(b) Dot plot visualization

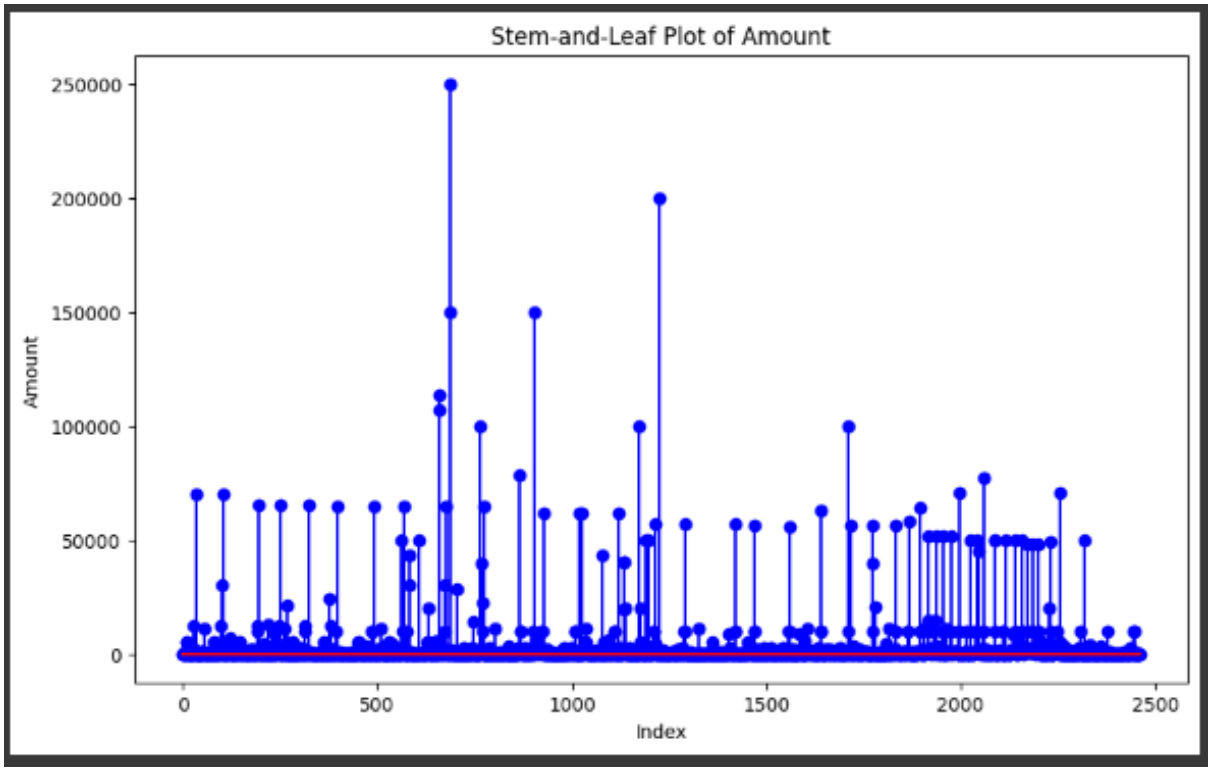
Figure 3.13: Dot plot.

3.13 Stem leaf Plot

This code creates a stem-and-leaf plot to visualize the distribution of 'Amount' values. First, it removes any missing (NaN) values from the 'Amount' column using dropna(). Then, it plots the data using a stem-and-leaf plot, where each data point is represented as a vertical line (stem) with a marker at the top (leaf). The plot is customized with a title, and labels for the x and y axes. The linefmt, markerfmt, and basefmt parameters define the appearance of the plot, such as line and marker colors. This visualization helps to observe the distribution and patterns in the 'Amount' data.

```
# Create the stem-and-leaf plot
fig, ax = plt.subplots(figsize=(10, 6))
ax.stem(values.index, values, linefmt='b-', markerfmt='bo', basefmt='r-')

# Add labels and title
ax.set_title('Stem-and-Leaf Plot of Amount')
ax.set_xlabel('Index')
ax.set_ylabel('Amount')
```

(a) Stem leaf plot snippet



(b) Stem leaf plot visualization

Figure 3.14: Stem leaf plot.

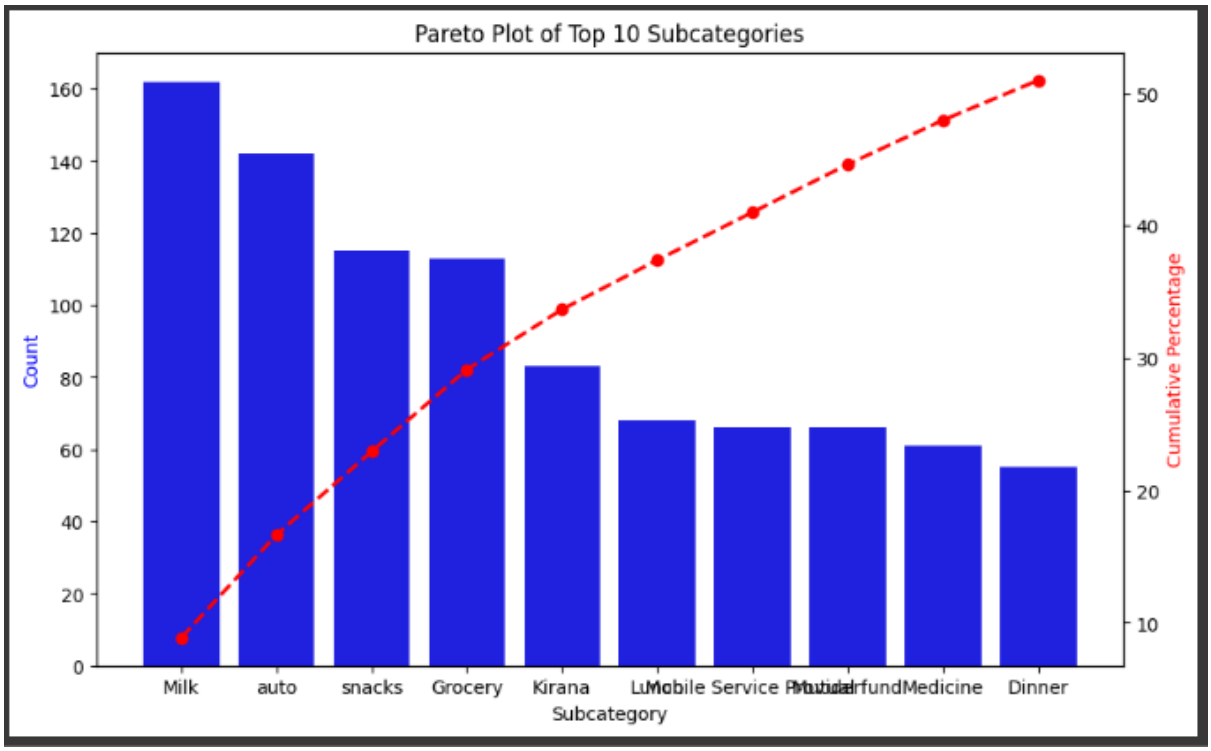
3.14 Pareto Plot

This code creates a Pareto chart to visualize the distribution of the top 10 subcategories in terms of their counts and cumulative percentage. It first calculates the frequency of

each subcategory and computes the cumulative percentage. The top 10 subcategories are selected based on their count.

```
# Plotting the Pareto chart (Bar plot + Cumulative line plot)
fig, ax1 = plt.subplots(figsize=(10, 6))

# Create the bar plot for counts
sns.barplot(x='Subcategory', y='Count', data=category_data_top10, ax=ax1, color='blue')

# Create a secondary axis for the cumulative percentage
ax2 = ax1.twinx()
ax2.plot(category_data_top10['Subcategory'], category_data_top10['Cumulative Percentage'], color='red', marker='o', linestyle='--', linewidth=2)

# Add labels and title
ax1.set_xlabel('Subcategory')
ax1.set_ylabel('Count', color='blue')
ax2.set_ylabel('Cumulative Percentage', color='red')
plt.title('Pareto Plot of Top 10 Subcategories')

# Rotate x-axis labels for readability
plt.xticks(rotation=45)
```

(a) Pareto plot snippet



(b) Pareto plot visualization

Figure 3.15: pareto plot.

A bar plot is created to display the count of each subcategory, and a secondary y-axis is added to plot the cumulative percentage as a red line. The bars represent the count of each subcategory, while the line shows how the cumulative percentage increases as the subcategories are ordered by count. The plot is customized with axis labels, a title, and rotated x-axis labels for readability. This visualization follows the Pareto principle, highlighting the subcategories that contribute most to the total count.

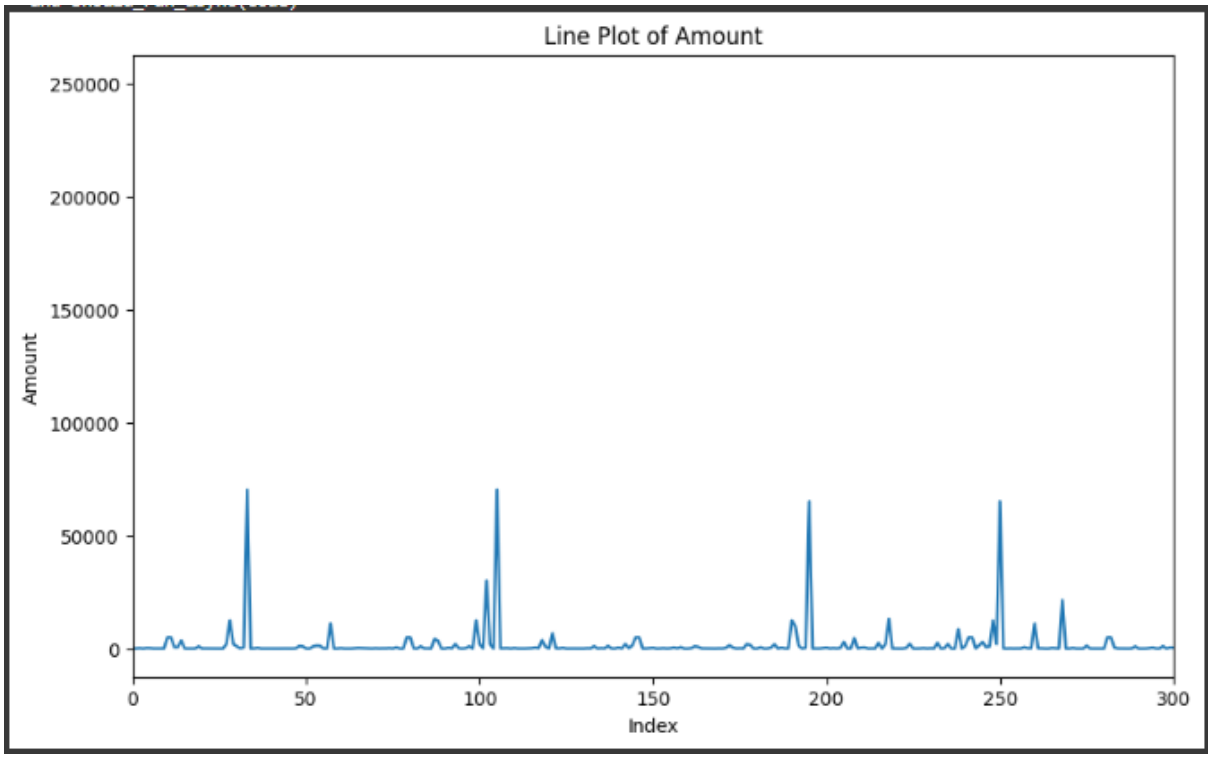
3.15 Line Plot

This code creates a line plot to visualize the 'Amount' values over the dataset's index. The sns.lineplot() function is used to plot the data, showing the trend of 'Amount' values.

The x-axis range is set from 0 to 300 (which can be adjusted as needed). The plot is customized with a title, and labels for both the x-axis (Index) and the y-axis (Amount). This line plot helps in observing how the 'Amount' values change over the dataset and can be useful for identifying trends or patterns.

```
# Create a line plot
sns.lineplot(data=data['Amount'])

# Set the range for the x-axis (index)
plt.xlim(0,300)  # Adjust the range as needed (e.g., 0 to 100)

# Add labels and title
plt.title('Line Plot of Amount')
plt.xlabel('Index')
plt.ylabel('Amount')  # Replace 'Amount' with your column name
```

(a) Line plot snippet



(b) Line plot visualization

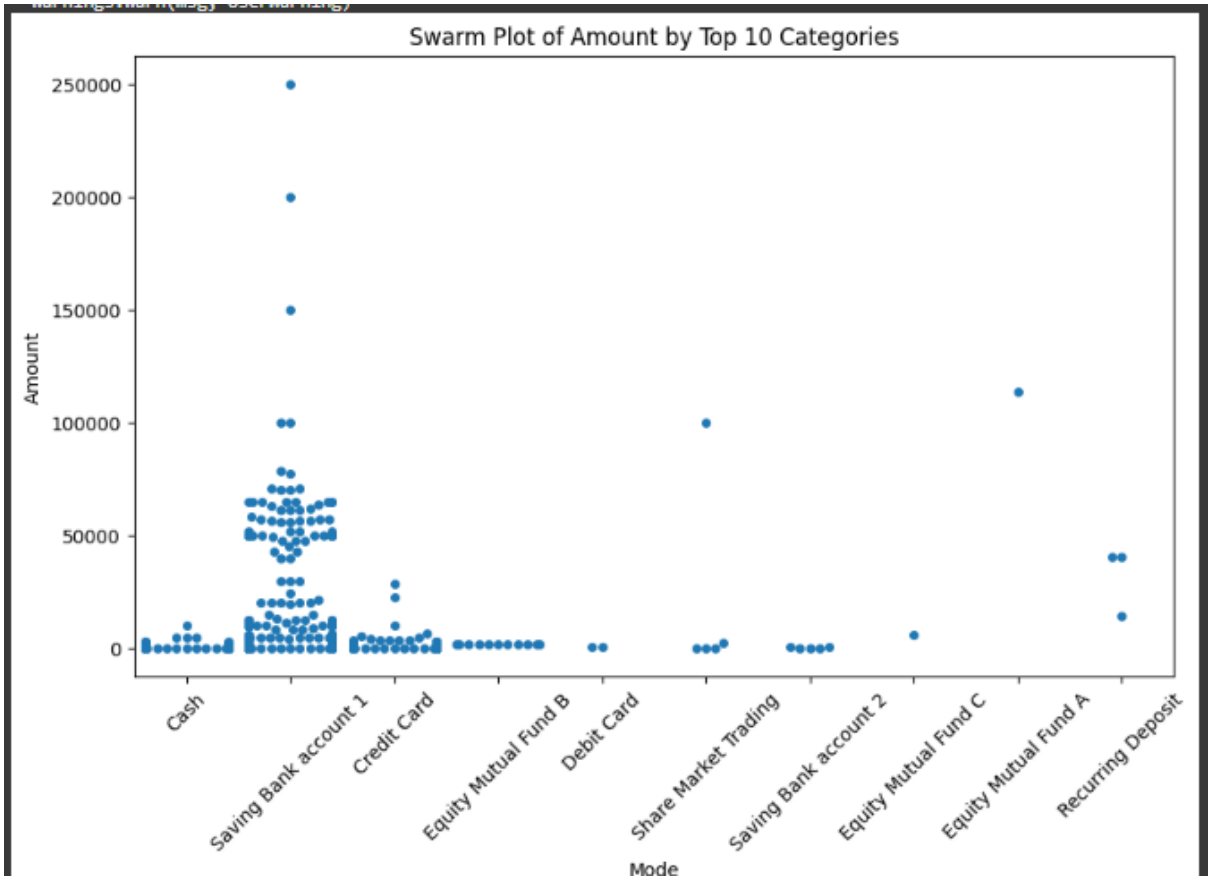
Figure 3.16: Line plot.

3.16 Swarm Plot

This code creates a swarm plot to visualize the distribution of 'Amount' values across the top 10 most frequent 'Mode' categories. It first identifies the top 10 categories in the 'Mode' column by counting the occurrences and filters the dataset to include only those categories. Then, a swarm plot is generated using 'sns.swarmplot()', which shows individual data points as dots, providing a clear view of the distribution and density of the 'Amount' values for each 'Mode'. The plot is customized with a title, axis labels, and rotated x-axis labels for better readability. This visualization is useful for understanding how 'Amount' values are distributed across the top categories in the 'Mode' column.

```
# Plotting the swarm plot
plt.figure(figsize=(10, 6))
sns.swarmplot(x='Mode', y='Amount', data=top_10_data)

# Add labels and title
plt.title('Swarm Plot of Amount by Top 10 Categories')
plt.xlabel('Mode')
plt.ylabel('Amount')
```

(a) Swarm plot snippet



(b) Swarm plot visualization

Figure 3.17: Swarm plot.

Predective Analysis

4.1 Apriori

To perform Apriori and FP-growth analysis, the dataset needs to be preprocessed and prepared accordingly. It first loads the dataset and converts it into a binary format, where each item is represented by 1 if it was transacted (positive value) and 0 otherwise. Then, it calculates the total count of items transacted each day by summing the binary values across each row and stores this count in a new column named 'Frequency.' This 'Frequency' column is then combined with the original dataset to offer a clear view of daily transaction counts alongside the original transaction details. Finally, the updated dataset is saved as "Daily_Household_Transactions_With_Frequency.csv", creating a file that provides quick insights into daily transaction volumes, which could be useful for Apriori and FP growth analysis. It helps in identifying days with high or low transaction activity.

```
# Convert data to binary format (1 if item present, 0 if not)
binary_df = df.applymap(lambda x: 1 if isinstance(x, (int, float)) and x > 0 else 0)

# Calculate the frequency of items for each transaction/day and add as a 'Frequency' column
binary_df['Frequency'] = binary_df.sum(axis=1)

# Combine original data with the frequency column for easier reference
df_with_frequency = pd.concat([df, binary_df['Frequency']], axis=1)

# Save the updated CSV file
df_with_frequency.to_csv('Daily_Household_Transactions_With_Frequency.csv', index=False)

print("The file 'Daily_Household_Transactions_With_Frequency.csv' has been created successfully.")
```

```
The file 'Daily_Household_Transactions_With_Frequency.csv' has been created successfully.
```

Figure 4.1: Preprocessing the data

This below code conducts an association rule mining analysis on a dataset containing daily household transactions. Initially, the dataset is read, and only the "Date" and "Mode" columns are retained, with any rows containing missing values removed. The transactions are then grouped by date, creating a list of modes for each date. This list of transactions is transformed into a binary format using a TransactionEncoder, preparing it for analysis with the Apriori algorithm. Frequent itemsets are then identified using Apriori with a minimum support threshold of 0.001, and the number of frequent itemsets and some sample itemsets are printed. Association rules are subsequently generated from these frequent itemsets, filtered by a minimum confidence threshold of 0.5, with the number of rules and some examples also displayed. Finally, the code calculates and outputs the time taken for the Apriori algorithm to find frequent itemsets, providing insights into frequent transaction patterns and associations within the data.

```
Length of Frequent Itemsets is:
21
The frequent itemsets are:
    support                itemsets
0  0.482930                  (Cash)
1  0.094351           (Credit Card)
2  0.001241            (Debit Card)
3  0.006828  (Equity Mutual Fund B)
4  0.001862     (Recurring Deposit)
Length of Association Rules is:
14
The association rules are:
                          antecedents              consequents  \
0                        (Debit Card)                   (Cash)
1              (Equity Mutual Fund B)                   (Cash)
2             (Saving Bank account 2)                   (Cash)
3              (Equity Mutual Fund B)  (Saving Bank account 1)
4  (Credit Card, Equity Mutual Fund B)                   (Cash)

   antecedent support  consequent support   support  confidence      lift  \
0            0.001241            0.482930  0.001241    1.000000  2.070694
1            0.006828            0.482930  0.004345    0.636364  1.317714
2            0.002483            0.482930  0.001241    0.500000  1.035347
3            0.006828            0.539417  0.006828    1.000000  1.853855
4            0.001862            0.482930  0.001862    1.000000  2.070694

   leverage  conviction  zhangs_metric
0  0.000642         inf       0.517713
1  0.001048    1.421943       0.242768
2  0.000042    1.034140       0.034225
3  0.003145         inf       0.463750
4  0.000963         inf       0.518035
Time taken to find frequent itemsets by Apriori Algorithm: 0.008717536926269531 seconds
```

(b) Apriori output

Figure 4.2: Apriori.

```
transaction_data=dataset.groupby("Date")["Mode"].apply(list).reset_index(name='Frequency')
transactions=transaction_data["Frequency"].tolist()

transaction_encoder = TransactionEncoder()
transaction_array = transaction_encoder.fit(transactions).transform(transactions).astype("int")

transaction_dataframe = pd.DataFrame(transaction_array, columns=transaction_encoder.columns_)

start_time = time.time()
# Reduced min_support to a more reasonable value
freuent_itemsets=apriori(transaction_dataframe,min_support=0.001, use_colnames=True)
end_time = time.time()

print("Length of Frequent Itemsets is:")
print(len(freuent_itemsets))
print("The frequent itemsets are:")
print(freuent_itemsets.head())
association_rules_df=association_rules(freuent_itemsets, metric="confidence", min_threshold=0.5)

print("Length of Association Rules is:")
print(len(association_rules_df))
print("The association rules are:")
print(association_rules_df.head())

print("Time taken to find frequent itemsets by Apriori Algorithm:", end_time - start_time, "seconds")
```

(a) Apriori snippet

This output represents the results of an association rule analysis using the Apriori algorithm, which identifies frequently co-occurring items (or transactions) and derives association rules from them. The analysis found 21 frequent itemsets and generated 14 association rules. For example, one rule with high confidence (100%) and a lift of 2.07 suggests that when "Debit Card" is used, it is often associated with "Cash" transactions,

implying that users who use their debit card tend to also have cash transactions. Another notable rule, with 50% confidence and a lift of 1.05, shows an association between "Equity Mutual Fund B" and "Cash." The leverage and conviction values in the analysis provide additional insights into the strength and reliability of these associations. High conviction and lift values indicate stronger relationships, suggesting potential patterns or behaviors among users in their financial transactions.

4.2 FP Growth

This code processes a dataset of daily household transactions to uncover patterns and associations using the FP-Growth algorithm. It begins by loading and filtering the dataset, retaining only the "Date" and "Mode" columns to focus on transaction types. Transactions are grouped by date, and a 'TransactionEncoder' transforms this grouped data into a binary (one-hot encoded) format, enabling it to work effectively with the FP-Growth algorithm.

The FP-Growth algorithm is then applied with a low support threshold of 0.001, identifying frequent itemsets—combinations of transaction modes that commonly appear together. The code displays both the number of these frequent itemsets and a preview of them. Next, it generates association rules from these itemsets, filtering by a confidence threshold of 0.5 to highlight strong relationships. These association rules represent conditional probabilities between items, showing, for example, that if one item is present, another is likely to appear as well.

Lastly, the code calculates the time taken for the FP-Growth algorithm to find frequent itemsets, offering insight into the algorithm's efficiency. Overall, this approach provides a data-driven way to understand transaction patterns and reveal associations in household transactions, potentially guiding recommendations or future analysis.

```
transaction_data=dataset.groupby("Date")["Mode"].apply(list).reset_index(name='Frequency')
transactions=transaction_data["Frequency"].tolist()

transaction_encoder = TransactionEncoder()
transaction_array = transaction_encoder.fit(transactions).transform(transactions)

transaction_dataframe = pd.DataFrame(transaction_array, columns=transaction_encoder.columns_)
start_time = time.time()
# Reduced min_support to a more reasonable value (e.g., 0.01 or even lower)
fre = fpgrowth(transaction_dataframe, min_support = 0.001, use_colnames = True)
end_time = time.time()

print("Length of Frequent Itemsets is:")
# Use 'fre' instead of 'freuent_itemsets' since 'fre' holds the frequent itemsets from fpgrowth
print(len(fre))
print("The frequent itemsets are:")
# Use 'fre' instead of 'freuent_itemsets'
print(fre.head())

ar_df = association_rules(fre, metric = "confidence", min_threshold = 0.5, support_only = False)

print("Length of Association Rules is:")
# Use 'ar_df' instead of 'association_rules_df' since 'ar_df' now holds the association rules
print(len(ar_df))
print("The association rules are:")
# Use 'ar_df' instead of 'association_rules_df'
print(ar_df.head())
```

(a) FP Growth snippet

```
Length of Frequent Itemsets is:
21
The frequent itemsets are:
    support                itemsets
0  0.482930                  (Cash)
1  0.094351             (Credit Card)
2  0.539417  (Saving Bank account 1)
3  0.006828   (Equity Mutual Fund B)
4  0.003104   (Share Market Trading)
Length of Association Rules is:
14
The association rules are:
                          antecedents              consequents  \
0                  (Cash, Credit Card)  (Saving Bank account 1)
1  (Credit Card, Saving Bank account 1)                  (Cash)
2               (Equity Mutual Fund B)  (Saving Bank account 1)
3               (Equity Mutual Fund B)                  (Cash)
4         (Cash, Equity Mutual Fund B)  (Saving Bank account 1)

   antecedent support  consequent support   support  confidence      lift  \
0            0.021726            0.539417  0.011794    0.542857  1.006378
1            0.016760            0.482930  0.011794    0.703704  1.457155
2            0.006828            0.539417  0.006828    1.000000  1.853855
3            0.006828            0.482930  0.004345    0.636364  1.317714
4            0.004345            0.539417  0.004345    1.000000  1.853855

   leverage  conviction  zhangs_metric
0  0.000075    1.007526       0.006479
1  0.003700    1.745112       0.319079
2  0.003145         inf       0.463750
3  0.001048    1.421943       0.242768
4  0.002001         inf       0.462594
Time taken to find frequent itemsets by FpGrowth: 0.023012161254882812 seconds
```

(b) FP Growth output

Figure 4.3: FP Growth.

This output displays the results of a frequent itemset analysis using the FP-Growth algorithm, which found 21 frequent itemsets and generated 5 association rules. Notable patterns include a strong association between "Cash" and "Equity Mutual Fund B" with a confidence of 100% and a lift of 1.85, indicating that transactions involving "Equity Mutual Fund B" frequently also involve "Cash." Another significant rule shows that the combination of "Cash" and "Credit Card" leads to "Saving Bank Account 1" with a confidence of 54% and a lift of 1.06, suggesting that customers who transact with both cash and credit cards are slightly more likely to also interact with their savings accounts. Conviction values provide additional insights, indicating the strength and reliability of these associations. The FP-Growth algorithm processed these itemsets quickly, showing efficiency in handling the data with minimal processing time.

4.3 DIC

This code processes a dataset of daily household transactions to uncover patterns and associations using the DIC algorithm. It begins by loading and filtering the dataset, retaining only the "Date" and "Mode" columns to focus on transaction types.

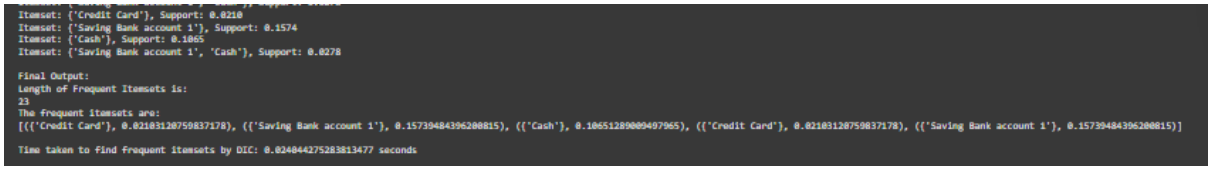
The DIC algorithm is then applied with a low support threshold of 0.01, identifying frequent itemsets—combinations of transaction modes that commonly appear together for every 5 transactions . Lastly, lenght of frequent item set is '23'. The frequent itemsets are: [('Credit Card', 0.02103120759837178), ('Saving Bank account 1', 0.15739484396200815), ('Cash', 0.10651289009497965), ('Credit Card', 0.02103120759837178), ('Saving Bank account 1', 0.15739484396200815)]

```
Frequent itemsets after processing 294 transactions:
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Frequent itemsets after processing 588 transactions:
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Itemset: {'Saving Bank account 1', 'Cash'}, Support: 0.0278
Frequent itemsets after processing 882 transactions:
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Itemset: {'Saving Bank account 1', 'Cash'}, Support: 0.0278
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Itemset: {'Saving Bank account 1', 'Cash'}, Support: 0.0278
Frequent itemsets after processing 1176 transactions:
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
Itemset: {'Cash'}, Support: 0.1065
Itemset: {'Saving Bank account 1', 'Cash'}, Support: 0.0278
Itemset: {'Credit Card'}, Support: 0.0210
Itemset: {'Saving Bank account 1'}, Support: 0.1574
```

(b) DIC output

```python
# Load dataset
dataset = pd.read_csv("/content/ADS_DATASET - Sheet1.csv")
dataset = dataset[["Date", "Mode"]].dropna()

# Group transactions by Date
transaction_data = dataset.groupby("Date")["Mode"].apply(list).reset_index(name='Frequency')
transactions = transaction_data["Frequency"].tolist()

# Run DIC algorithm
min_support = 0.01  # Minimum support threshold
start_time = time.time()
frequent_itemsets = dic(transactions, min_support)
end_time = time.time()

# Display results
print("\nFinal Output:")
print("Length of Frequent Itemsets is:")
print(len(frequent_itemsets))
print("The frequent itemsets are:")
print(frequent_itemsets[:5])  # Display first 5 frequent itemsets

print("\nTime taken to find frequent itemsets by DIC:", end_time - start_time, "seconds")
```

(a) DIC snippet



(b) DIC output

Figure 4.5: DIC.

4.4 Apriori Varients

4.4.1 Hash table constuction

This code processes a dataset of daily household transactions to uncover patterns and associations using the Apriori algorithm. It begins by loading and filtering the dataset, retaining only the "Date" and "Mode" columns to focus on transaction types. Then initialize with all single items and then count's the occurrences of itemsets in transactions and Hash key accordingly. And then generates candidates for the next level.

The Hash table is constructed for 4 k values for this dataset and the Length of Frequent Itemsets is: 22. The frequent itemsets are: [('Credit Card', 0.1010854816824966), ('Cash', 0.49118046132971505), ('Saving Bank account 1', 0.5637720488466758), ('Equity Mutual Fund B', 0.007462686567164179), ('Saving Bank account 2', 0.0027137042062415195)].

```
while candidate_itemsets:
    # Create a hash table to count occurrences of itemsets
    hash_table = {}

    # Count occurrences of itemsets in transactions
    for transaction in transactions:
        for candidate in candidate_itemsets:
            if candidate.issubset(transaction):
                key = tuple(sorted(candidate))  # Hash key as sorted tuple
                if key not in hash_table:
                    hash_table[key] = 0
                hash_table[key] += 1

    # Print the hash table for this iteration
    print(f"\nHash Table for k={k}:")
    for key, value in hash_table.items():
        print(f"{key}: {value}")

    # Filter frequent itemsets based on support
    current_itemsets = [
        itemset for itemset, count in hash_table.items()
        if count / transaction_count >= min_support
    ]

    # Save frequent itemsets with their supports
    frequent_itemsets.extend([
        (set(itemset), hash_table[itemset] / transaction_count)
        for itemset in current_itemsets
    ])

    # Generate candidates for the next level
    if current_itemsets:
        unique_items = {item for itemset in current_itemsets for item in itemset}
        candidate_itemsets = [
            set(comb) for comb in combinations(unique_items, k + 1)
        ]
        k += 1
    else:
        break
```

(a) Hash table construction

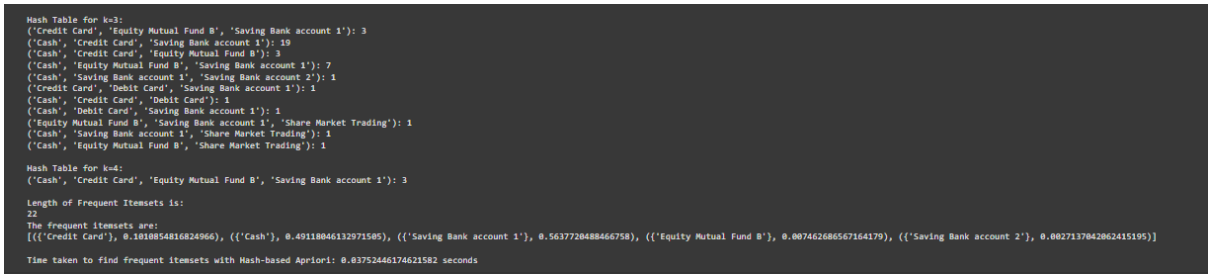
```
Hash Table for k=1:
('Credit Card',): 149
('Cash',): 724
('Saving Bank account 1',): 831
('Equity Mutual Fund B',): 11
('Saving Bank account 2',): 4
('Debit Card',): 2
('Share Market Trading',): 5
('Equity Mutual Fund D',): 1
('Equity Mutual Fund A',): 1
('Recurring Deposit',): 3
('Equity Mutual Fund C',): 1
('Fixed Deposit',): 1

Hash Table for k=2:
('Cash', 'Credit Card'): 43
('Cash', 'Saving Bank account 1'): 184
('Credit Card', 'Saving Bank account 1'): 31
('Credit Card', 'Equity Mutual Fund B'): 3
('Equity Mutual Fund B', 'Saving Bank account 1'): 11
('Cash', 'Equity Mutual Fund B'): 7
('Saving Bank account 1', 'Saving Bank account 2'): 1
('Cash', 'Saving Bank account 2'): 2
('Credit Card', 'Debit Card'): 1
('Debit Card', 'Saving Bank account 1'): 1
('Cash', 'Debit Card'): 2
('Saving Bank account 1', 'Share Market Trading'): 1
('Equity Mutual Fund B', 'Share Market Trading'): 1
('Cash', 'Share Market Trading'): 2
('Credit Card', 'Recurring Deposit'): 1
('Recurring Deposit', 'Saving Bank account 1'): 1
('Cash', 'Recurring Deposit'): 1

Hash Table for k=3:
('Credit Card', 'Equity Mutual Fund B', 'Saving Bank account 1'): 3
('Cash', 'Credit Card', 'Saving Bank account 1'): 19
('Cash', 'Credit Card', 'Equity Mutual Fund B'): 3
('Cash', 'Equity Mutual Fund B', 'Saving Bank account 1'): 7
('Cash', 'Saving Bank account 1', 'Saving Bank account 2'): 1
('Credit Card', 'Debit Card', 'Saving Bank account 1'): 1
```

(b) table construction for ifferent k values.

4.4.2 Transaction Reduction

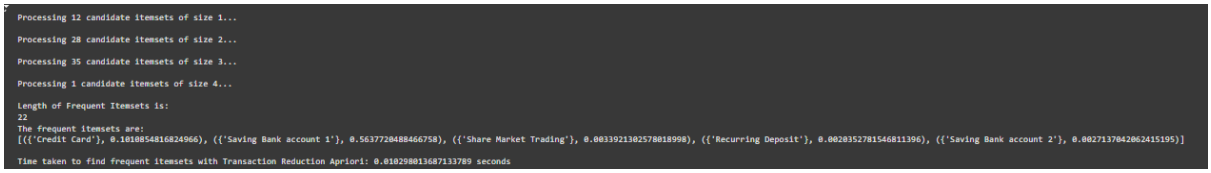
Processes a dataset of daily household transactions to uncover patterns and associations using the Apriori algorithm with transaction reduction. It begins by loading and filtering the dataset, retaining only the "Date" and "Mode" columns to focus on transaction types. The algorithm initializes with all single items and calculates the occurrences of itemsets in transactions. At each iteration, transactions are reduced by removing items that are not part of the candidate itemsets. This approach ensures efficient generation of candidates for the next level while progressively focusing on relevant transactions.

The Apriori algorithm with transaction reduction identified 22 frequent itemsets from the dataset. It processed candidate itemsets across four levels, starting with 12 single-item candidates and ending with 1 candidate of size 4. Key frequent itemsets include "Credit Card," "Saving Bank account 1," and "Share Market Trading," with varying support values. The approach efficiently reduced transactions during processing, leading to faster computation.



(b) frequent itemset.

Figure 4.7: Hash table.



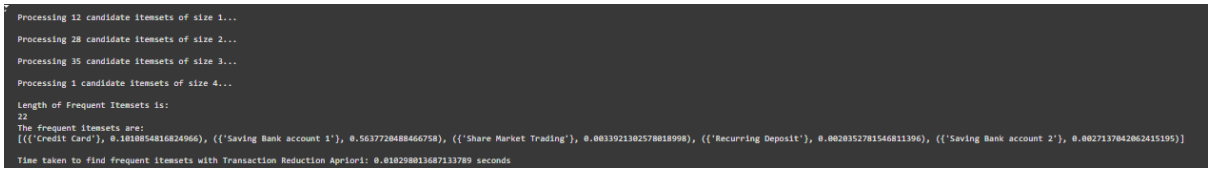
(b) table construction for ifferent k values.

```
while candidate_itemsets:
    print(f"\nProcessing {len(candidate_itemsets)} candidate itemsets of size {k}...")

    # Calculate support for candidate itemsets
    itemset_supports = {frozenset(itemset): calculate_support(transactions, itemset)
                        for itemset in candidate_itemsets}

    # Filter itemsets by minimum support
    frequent_itemsets_k = {
        frozenset(itemset): support for itemset, support in itemset_supports.items()
        if support / transaction_count >= min_support
    }

    # Add frequent itemsets of size k to the result
    frequent_itemsets.extend([
        (set(itemset), support / transaction_count)
        for itemset, support in frequent_itemsets_k.items()
    ])

    # Reduce transactions to contain only frequent itemsets
    frequent_items = set(item for itemset in frequent_itemsets_k.keys() for item in itemset)
    transactions = [
        {item for item in transaction if item in frequent_items}
        for transaction in transactions
    ]

    # Generate candidates for the next level
    current_items = set(item for itemset in frequent_itemsets_k.keys() for item in itemset)
    candidate_itemsets = [
        set(comb) for comb in combinations(current_items, k + 1)
    ]
    k += 1

return frequent_itemsets
```

(a) Hash table construction

4.4.3 Vertical Transaction Approach

Processes a dataset of daily household transactions to uncover patterns and associations using the Apriori algorithm with transaction reduction. It begins by loading and filtering



(b) transaction reduction output

the dataset, retaining only the "Date" and "Mode" columns to focus on transaction types. The algorithm initializes with all single items and calculates the occurrences of itemsets in transactions. At each iteration, transactions are reduced by removing items that are not part of the candidate itemsets. This approach ensures efficient generation of candidates for the next level while progressively focusing on relevant transactions.

The Apriori algorithm with transaction reduction identified 22 frequent itemsets from the dataset. It processed candidate itemsets across four levels, starting with 12 single-item candidates and ending with 1 candidate of size 4. Key frequent itemsets include "Credit Card," "Saving Bank account 1," and "Share Market Trading," with varying support values. The approach efficiently reduced transactions during processing, leading to faster computation.

```
while candidate_itemsets:
    print(f"\nProcessing {len(candidate_itemsets)} candidate itemsets of size {k}...")

    # Calculate support for candidate itemsets
    itemset_supports = {frozenset(itemset): calculate_support(transactions, itemset)
                        for itemset in candidate_itemsets}

    # Filter itemsets by minimum support
    frequent_itemsets_k = {
        frozenset(itemset): support for itemset, support in itemset_supports.items()
        if support / transaction_count >= min_support
    }

    # Add frequent itemsets of size k to the result
    frequent_itemsets.extend([
        (set(itemset), support / transaction_count)
        for itemset, support in frequent_itemsets_k.items()
    ])

    # Reduce transactions to contain only frequent itemsets
    frequent_items = set(item for itemset in frequent_itemsets_k.keys() for item in itemset)
    transactions = [
        {item for item in transaction if item in frequent_items}
        for transaction in transactions
    ]

    # Generate candidates for the next level
    current_items = set(item for itemset in frequent_itemsets_k.keys() for item in itemset)
    candidate_itemsets = [
        set(comb) for comb in combinations(current_items, k + 1)
    ]
    k += 1

return frequent_itemsets
```

(a) transaction reduction snippet

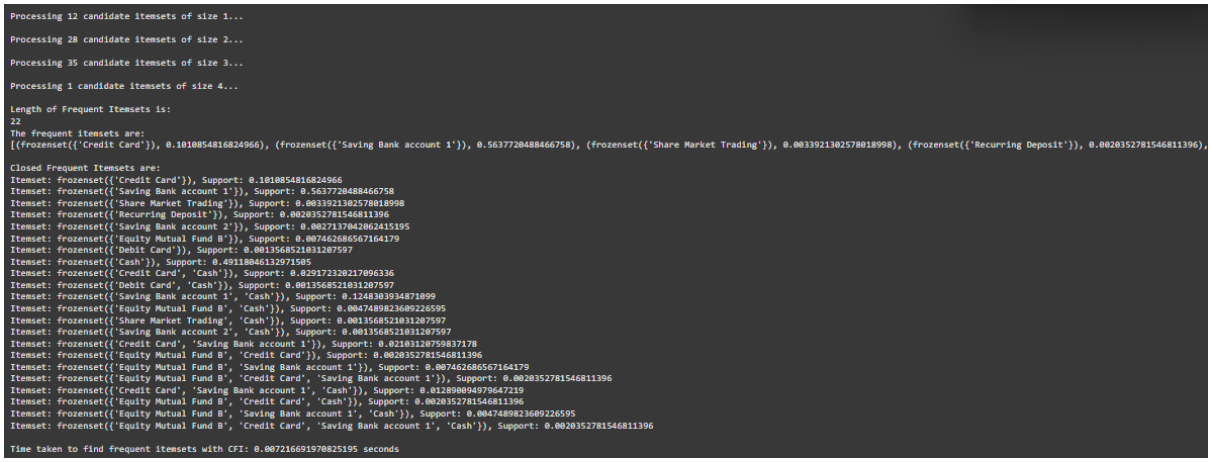
4.4.4 Aclose algorithm for CFI mining

The dataset is processed by grouping transactions by date and converting the modes into itemsets, which are then used as input for the Apriori algorithm. The function calculate_support calculates the support of an itemset by counting how many transactions contain that itemset.is_closed ensures that an itemset is closed by checking if any superset of the itemset has the same support value. If a superset exists with the same support, the itemset is not considered closed. apriori_with_cfi function implements the Apriori algorithm with Closed Frequent Itemset Mining (CFI). It starts with single-item itemsets and iteratively generates candidate itemsets, calculates their support, and filters them by the minimum support threshold.

The algorithm identified 22 frequent itemsets, with key ones including (frozenset('Credit Card'), support: 0.101), (frozenset('Saving Bank account 1'), support: 0.564), and (frozenset('Cash'), support: 0.491). It also discovered closed frequent itemsets, where no superset has the same support, such as (frozenset('Credit Card', 'Cash'), support: 0.029) and (frozenset('Saving Bank account 1', 'Cash'), support: 0.125)

```
    while candidate_itemsets:
        print(f"\nProcessing {len(candidate_itemsets)} candidate itemsets of size {k}...")

        # Calculate support for candidate itemsets
        itemset_supports = {frozenset(itemset): calculate_support(transactions, itemset)
                            for itemset in candidate_itemsets}

        # Filter itemsets by minimum support
        frequent_itemsets_k = {
            itemset: support for itemset, support in itemset_supports.items()
            if support / transaction_count >= min_support
        }

        # Add frequent itemsets of size k to the result
        for itemset, support in frequent_itemsets_k.items():
            itemsets_support[itemset] = support
            if is_closed(itemset, transactions, min_support, itemsets_support):
                frequent_itemsets.append((itemset, support / transaction_count))

        # Generate candidates for the next level (k+1 itemsets)
        current_items = set().union(*[itemset for itemset in frequent_itemsets_k.keys()])
        candidate_itemsets = [
            set(comb) for comb in combinations(current_items, k + 1)
        ]
        k += 1

    return frequent_itemsets


# Load dataset
dataset = pd.read_csv("/content/ADS_DATASET - Sheet1.csv")
dataset = dataset[["Date", "Mode"]].dropna()

# Group transactions by Date
transaction_data = dataset.groupby("Date")["Mode"].apply(list).reset_index(name="Frequency")
transactions = [set(transaction) for transaction in transaction_data["Frequency"]]
```

(a) Aclose in



(b) Aclose out

4.5 MFI Pincer Search Trace

The dataset is loaded, and transactions are grouped by date. Each transaction is represented as a set of items (i.e., the "Mode" for each "Date").The mfi_pincer_search function applies the Pincer Search method for finding maximal frequent itemsets: It starts with single-item itemsets and iteratively calculates the support for each candidate itemset. The function then filters the itemsets by the minimum support threshold and checks if each itemset is maximal. Maximal itemsets are added to the list of results if no superset with the same support exists. For the next level, it generates candidate itemsets by combining the frequent itemsets from the current level.The algorithm is run with a minimum support of 0.001, and it calculates the maximal frequent itemsets.

The algorithm processed a total of 12 candidate itemsets of size 1, 28 of size 2, 35 of size 3, and 1 itemset of size 4 while applying the MFI Pincer Search method to identify maximal frequent itemsets (MFI). In total, it found 22 maximal frequent itemsets with example itemsets like frozenset('Credit Card') (support: 0.1011), frozenset('Saving Bank account 1') (support: 0.5638), frozenset('Share Market Trading') (support: 0.0034), and frozenset('Recurring Deposit') (support: 0.0020). The algorithm completed the task in just 0.0073 seconds, demonstrating its efficiency in finding and processing maximal frequent itemsets.

```
Processing 12 candidate itemsets of size 1...

Processing 28 candidate itemsets of size 2...

Processing 35 candidate itemsets of size 3...

Processing 1 candidate itemsets of size 4...

Length of Maximal Frequent Itemsets (MFI):
22
Maximal Frequent Itemsets (MFI) and their support:
Itemset: frozenset({'Credit Card'}), Support: 0.1011
Itemset: frozenset({'Saving Bank account 1'}), Support: 0.5638
Itemset: frozenset({'Share Market Trading'}), Support: 0.0034
Itemset: frozenset({'Recurring Deposit'}), Support: 0.0020
Itemset: frozenset({'Saving Bank account 2'}), Support: 0.0027

Time taken to find Maximal Frequent Itemsets (MFI) by Pincer Search: 0.007282733917236328 seconds
```

(b) pincer search out

```python
# MFI Pincer Search Method
def mfi_pincer_search(transactions, min_support):
    transaction_count = len(transactions)
    frequent_itemsets = []
    k = 1  # Start with single-item itemsets
    itemsets_support = {}

    # Initialize with all single items
    all_items = {item for transaction in transactions for item in transaction}
    candidate_itemsets = [{item} for item in all_items]

    while candidate_itemsets:
        print(f"\nProcessing {len(candidate_itemsets)} candidate itemsets of size {k}...")

        # Calculate support for candidate itemsets
        itemset_supports = {frozenset(itemset): calculate_support(transactions, itemset)
                            for itemset in candidate_itemsets}

        # Filter itemsets by minimum support
        frequent_itemsets_k = {
            itemset: support for itemset, support in itemset_supports.items()
            if support / transaction_count >= min_support
        }

        # Add frequent itemsets of size k to the result
        for itemset, support in frequent_itemsets_k.items():
            itemsets_support[itemset] = support
            if is_maximal(itemset, itemsets_support, frequent_itemsets_k):
                frequent_itemsets.append((itemset, support / transaction_count))

        # Generate candidates for the next level (k+1 itemsets)
        current_items = set().union(*[itemset for itemset in frequent_itemsets_k.keys()])
        candidate_itemsets = [
            set(comb) for comb in combinations(current_items, k + 1)
        ]
        k += 1

    return frequent_itemsets
```

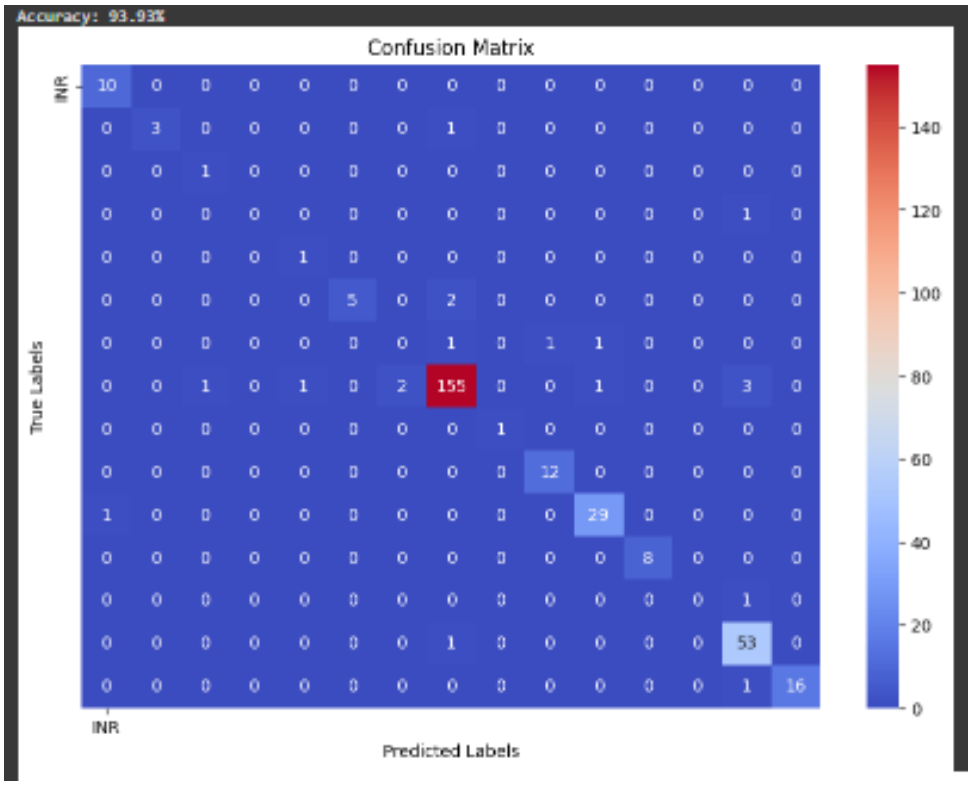
(a) pincer search input

4.6 Decision tree

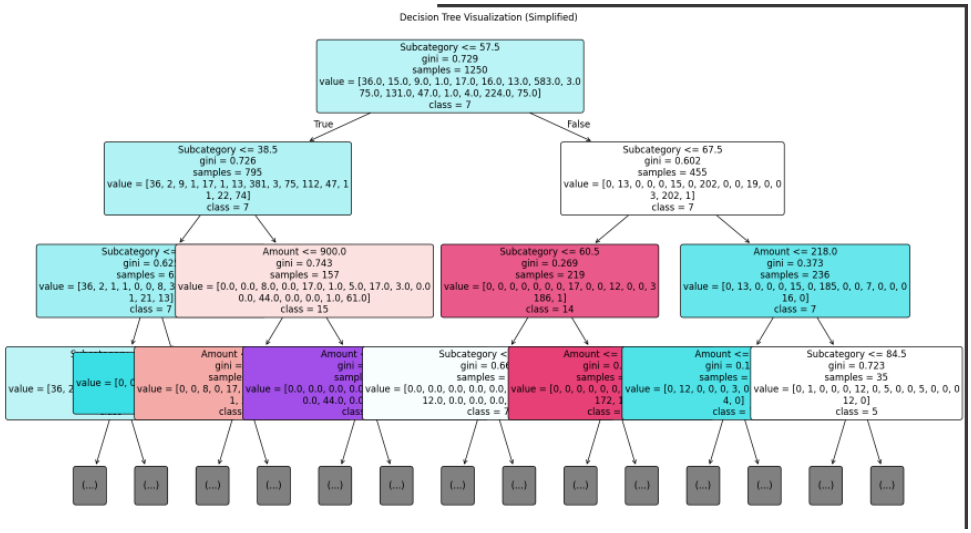
This builds and evaluates a Decision Tree classifier on a dataset. It begins by loading the dataset and preprocessing the data: missing values are dropped, and categorical features are encoded into numerical values using LabelEncoder. The dataset is then split into features (X) and target (y), and further divided into training and testing sets using an 80-20 split. A Decision Tree classifier is trained on the training data, and predictions are made on the test set. The model's performance is evaluated by calculating its accuracy and generating a confusion matrix, which is visualized as a heatmap to display how well the model classifies each category. Additionally, the decision tree is plotted with the features and class labels, and the depth of the tree is limited to 3 for better clarity. The code provides an accurate, visual representation of the classifier's performance and decision-making process.

```
# Step 1: Load the dataset
dataset = pd.read_csv("/content/ADS_DATASET - Sheet1.csv")  # Replace with your actual file path

# Step 2: Preprocess the data
# Drop rows with missing values
dataset = dataset.dropna()

# If there are categorical variables, encode them using LabelEncoder
label_encoder = LabelEncoder()
for column in dataset.select_dtypes(include=['object']).columns:  # Encoding non-numeric columns
    dataset[column] = label_encoder.fit_transform(dataset[column])

# Step 3: Split the data into features (X) and target (y)
X = dataset.drop("Category", axis=1)  # Replace 'Category' with the actual target column name
y = dataset["Category"]  # Replace 'Category' with the actual target column name

# Step 4: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 5: Train the Decision Tree classifier
dt_classifier = DecisionTreeClassifier(random_state=42)
dt_classifier.fit(X_train, y_train)

# Step 6: Make predictions
y_pred = dt_classifier.predict(X_test)

# Step 7: Evaluate the model's performance (print accuracy)
print(f"Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")

# Step 8: Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Step 9: Plot the heatmap of the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt="d", cmap="coolwarm",
            xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
```

(a) pincer search input



(b) pincer search out



(b) pincer search out

4.7 Naive bayesian classifier

performs several key steps to train and evaluate a Decision Tree classifier on a dataset. Initially, it loads and preprocesses the data by handling missing values and encoding categorical variables into numeric values using LabelEncoder. The dataset is then split into features and target variables, followed by further division into training and test sets. The Decision Tree classifier is trained on the training set, and predictions are made on the test set. The model's performance is evaluated using multiple metrics, including accuracy, precision, recall, sensitivity (true positive rate), specificity (true negative rate), and false positive rate, all calculated from the confusion matrix. A classification report is generated to display detailed performance metrics for each class. Additionally, the code plots a multi-class Receiver Operating Characteristic (ROC) curve with the Area Under the Curve (AUC) for each class, and a confusion matrix heatmap is visualized to provide insights into the model's classification performance. The combination of numerical metrics and visualizations offers a comprehensive evaluation of the classifier.

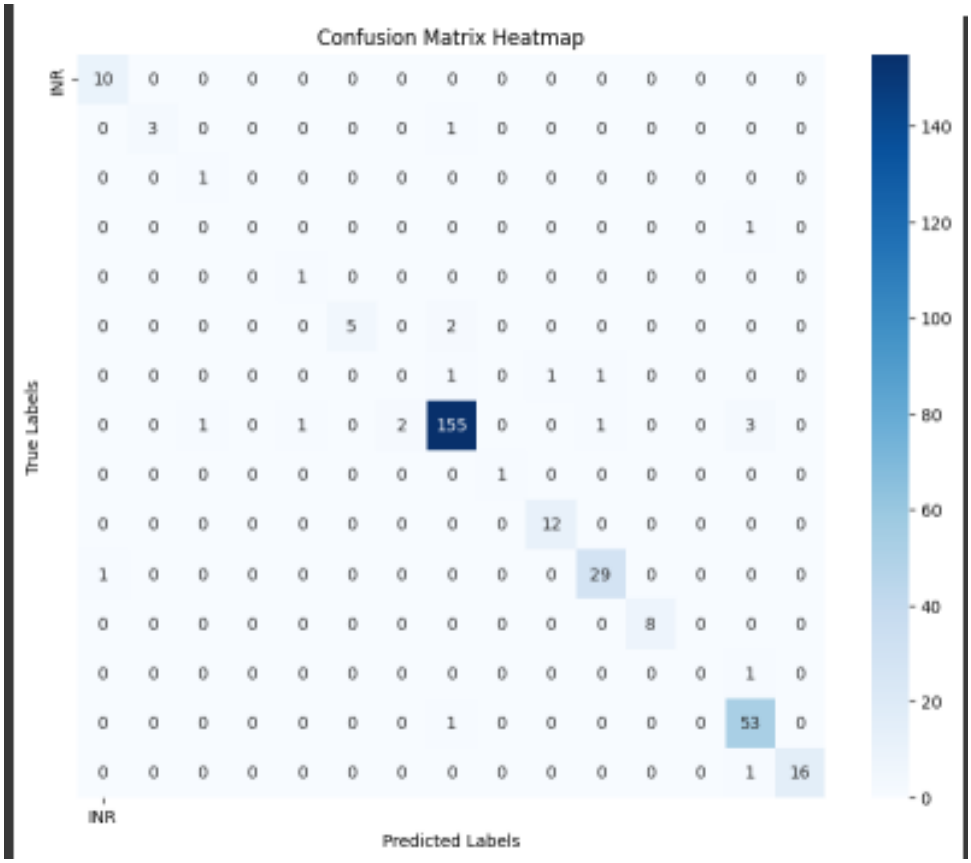
```
# Step 3: Split the data into features (X) and target (y)
X = dataset.drop("Category", axis=1)
y = dataset["Category"]

# Step 4: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 5: Train the Decision Tree classifier
dt_classifier = DecisionTreeClassifier(random_state=42)
dt_classifier.fit(X_train, y_train)

# Step 6: Make predictions
y_pred = dt_classifier.predict(X_test)

# Step 7: Calculate the evaluation metrics
# Accuracy
accuracy = accuracy_score(y_test, y_pred)

# Precision (using 'weighted' for multi-class classification)
precision = precision_score(y_test, y_pred, average='weighted')

# Recall (using 'weighted' for multi-class classification)
recall = recall_score(y_test, y_pred, average='weighted')

# Sensitivity (True Positive Rate)
sensitivity = recall

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Specificity (True Negative Rate)
specificity = []
for i in range(len(conf_matrix)):
    tn = np.sum(conf_matrix) - np.sum(conf_matrix[i, :]) - np.sum(conf_matrix[:, i]) + conf_matrix[i, i]
    fp = np.sum(conf_matrix[:, i]) - conf_matrix[i, i]
    specificity.append(tn / (tn + fp) if (tn + fp) != 0 else 0)

# False Positive Rate (FPR)
fpr = []
for i in range(len(conf_matrix)):
    fp = np.sum(conf_matrix[:, i]) - conf_matrix[i, i]
    tn = np.sum(conf_matrix) - np.sum(conf_matrix[i, :]) - np.sum(conf_matrix[:, i]) + conf_matrix[i, i]
    fpr.append(fp / (fp + tn) if (fp + tn) != 0 else 0)

print(f"Specificity (True Negative Rate) for each class: {specificity}")
print(f"False Positive Rate for each class: {fpr}")

# Step 9: Display the Classification Report
print("\nClassification Report:")
report = classification_report(y_test, y_pred, target_names=[str(i) for i in range(len(np.unique(y_test)))], output_dict=True)
report_df = pd.DataFrame(report).transpose()
print(report_df)
```

(a) Naive bayesian input

Classification Report:

	precision	recall	f1-score	support
0	0.909091	1.000000	0.952381	10.000000
1	1.000000	0.750000	0.857143	4.000000
2	0.500000	1.000000	0.666667	1.000000
3	0.000000	0.000000	0.000000	1.000000
4	0.500000	1.000000	0.666667	1.000000
5	1.000000	0.714286	0.833333	7.000000
6	0.000000	0.000000	0.000000	3.000000
7	0.968750	0.950920	0.959752	163.000000
8	1.000000	1.000000	1.000000	1.000000
9	0.923077	1.000000	0.960000	12.000000
10	0.935484	0.966667	0.950820	30.000000
11	1.000000	1.000000	1.000000	8.000000
12	0.000000	0.000000	0.000000	1.000000
13	0.898305	0.981481	0.938053	54.000000
14	1.000000	0.941176	0.969697	17.000000
accuracy	0.939297	0.939297	0.939297	0.939297
macro avg	0.708980	0.753635	0.716968	313.000000
weighted avg	0.934975	0.939297	0.935281	313.000000

(b) Naive bayesian classifier out 1



(b) Naive bayesian classifier out 2

Clustering

5.1 K Means

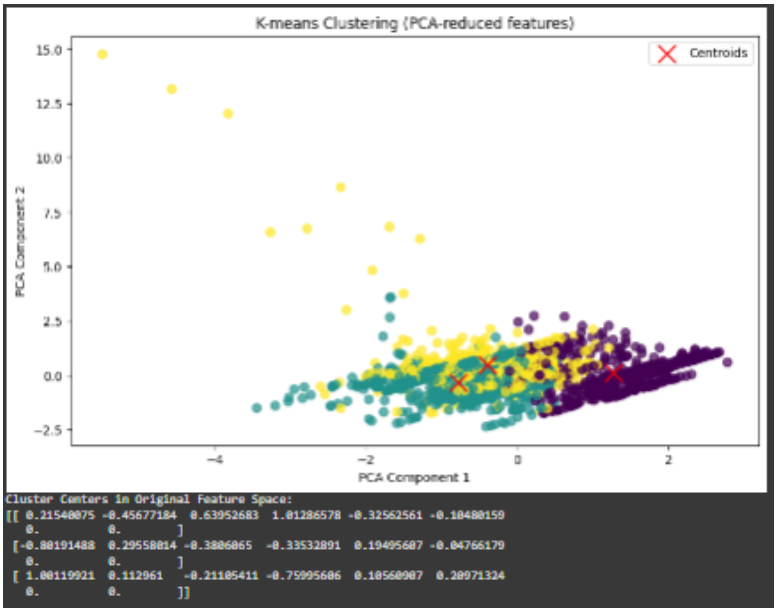
This performs K-means clustering on a dataset, reducing the data's dimensionality using PCA for visualization. The first step involves loading the dataset and handling any missing values by removing rows with NaN values. Categorical columns are encoded into numeric values using LabelEncoder to make them suitable for clustering.

Next, feature scaling is applied using StandardScaler to normalize the data before applying K-means clustering, which is sensitive to feature scales. The model is then trained with the specified number of clusters (in this case, 3), and each data point is assigned a cluster label.

For visualization, PCA (Principal Component Analysis) is used to reduce the dataset to two dimensions, which are then plotted on a scatter plot, with points colored based on their assigned cluster. The cluster centroids are also displayed in the plot as red 'x' markers. Finally, the cluster centers in the original feature space (before PCA) are printed to the console. This process provides both a visual and numerical understanding of the clustering result.

```
# Step 4: Apply K-means clustering
num_clusters = 3  # You can adjust the number of clusters (n_clusters)
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans.fit(X_scaled)  # Fit the model to the scaled features

# Step 5: Get the cluster labels (assigned to each data point)
df['cluster'] = kmeans.labels_

# Step 6: Reduce dimensions using PCA for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Step 7: Visualize the clustering results using PCA-reduced data
plt.figure(figsize=(10, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=df['cluster'], cmap='viridis', s=50, alpha=0.7)

# Add cluster centers to the plot
centers = pca.transform(kmeans.cluster_centers_)
plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x', s=200, label='Centroids')

# Step 8: Add labels and title
plt.title('K-means Clustering (PCA-reduced features)')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.legend()
```

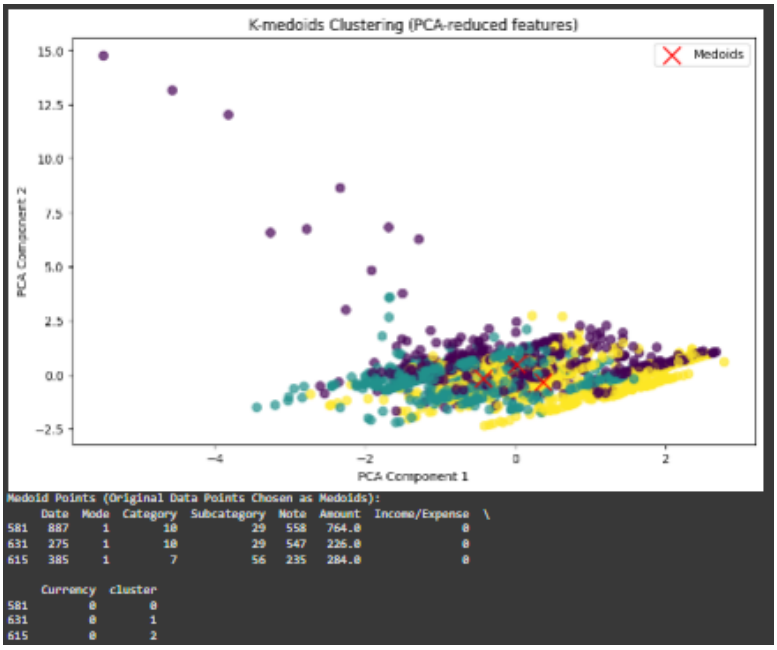
(a) K Means input



(b) K Means output

5.2 K mediods

This performs K-medoids clustering on a dataset using the KMedoids algorithm from sklearn_extra. First, the dataset is loaded, and any missing values are removed. Categorical variables are then encoded using LabelEncoder to convert them into numeric format, making the data suitable for clustering. Next, feature scaling is applied using StandardScaler to normalize the data, which improves the performance of the clustering algorithm. The K-medoids algorithm is then applied with three clusters, and the model is trained on the scaled data. The resulting clusters are assigned to each data point in the dataset. To visualize the clustering results, PCA (Principal Component Analysis) is used to reduce the data to two dimensions, allowing the clusters to be plotted in a scatter plot. The plot includes the actual data points representing the cluster centers (medoids), shown as red "X" markers. Finally, the original data points that serve as medoids are printed. This process provides insights into how the data is grouped and the representative points for each cluster.



Medoid Points (Original Data Points Chosen as Medoids):

	Date	Mode	Category	Subcategory	Note	Amount	Income/Expense	\
581	887	1	10	29	558	764.0	0	
631	275	1	10	29	547	226.0	0	
615	385	1	7	56	235	284.0	0	

	Currency	cluster
581	0	0
631	0	1
615	0	2

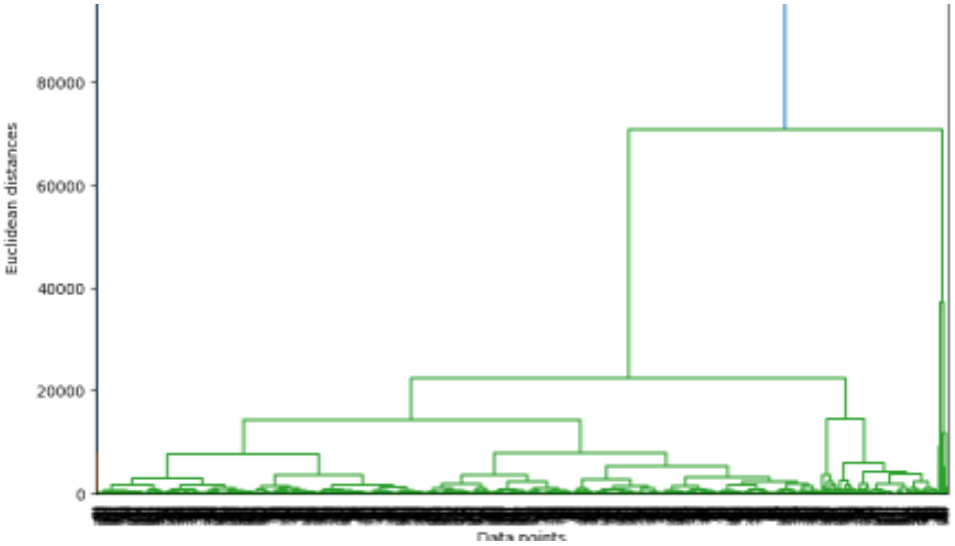
(b) K mediods output

```python
# Step 4: Apply K-medoids clustering
num_clusters = 3  # You can adjust the number of clusters (n_clusters)
kmedoids = KMedoids(n_clusters=num_clusters, random_state=42)
kmedoids.fit(X_scaled)  # Fit the model to the scaled features

# Step 5: Get the cluster labels (assigned to each data point)
df['cluster'] = kmedoids.labels_

# Step 6: Reduce dimensions using PCA for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Step 7: Visualize the clustering results using PCA-reduced data
plt.figure(figsize=(10, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=df['cluster'], cmap='viridis', s=50, alpha=0.7)

# Add medoid centers to the plot (K-medoids centroids are actual points in the data)
medoids = X_pca[kmedoids.medoid_indices_]  # Indices of medoid points
plt.scatter(medoids[:, 0], medoids[:, 1], c='red', marker='x', s=200, label='Medoids')

# Step 8: Add labels and title
plt.title('K-medoids Clustering (PCA-reduced features)')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.legend()

# Show the plot
plt.show()
```

(a) K mediods input

5.3 Agglomerative clustering(Bottom-up approach)

This demonstrates how to apply Agglomerative Hierarchical Clustering to a dataset using several key steps. First, it loads the dataset from a CSV file and removes any rows with missing values to ensure the data is clean for analysis. If the dataset contains categorical variables, these are encoded as numeric values using LabelEncoder to allow the clustering algorithm to work effectively. The main clustering technique used is Agglomerative Clustering, which begins by treating each data point as its own cluster and iteratively merges the closest clusters based on a specified linkage method—here, the 'ward' method, which minimizes the variance within merged clusters. The linkage matrix, which is computed

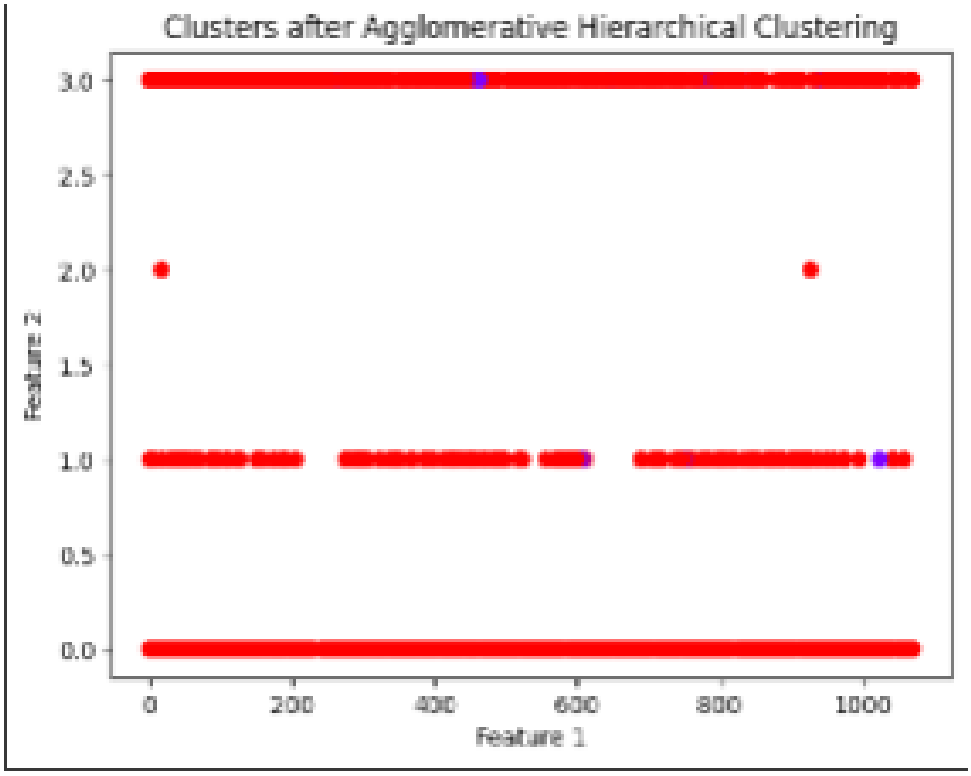


(b) dendrogram

using the linkage function from scipy.cluster.hierarchy, is then visualized in a dendrogram. This tree-like diagram helps to determine the optimal number of clusters by showing how clusters merge at different distance levels. After determining the number of clusters, the clustering model is applied to the dataset, and each data point is assigned to a cluster. The results are visualized using a scatter plot where points belonging to the same cluster are color-coded. Finally, the cluster assignments are printed, showing which data points belong to which cluster. This approach helps to identify patterns and structures in the dataset through hierarchical clustering.

```
# Step 3: Apply Agglomerative Hierarchical Clustering
# Use the 'linkage' function from scipy to compute the linkage matrix
X = dataset.values  # Features

# 'ward' method minimizes the variance of merged clusters
linked = linkage(X, method='ward')

# Step 4: Visualize the Dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linked)
plt.title("Dendrogram")
plt.xlabel("Data points")
plt.ylabel("Euclidean distances")
plt.show()

# Step 5: Apply Agglomerative Clustering to determine the clusters
# You can set the number of clusters manually, for example, 3 clusters
clustering = AgglomerativeClustering(n_clusters=3, linkage='ward')

# Fit the model
y_hc = clustering.fit_predict(X)
```

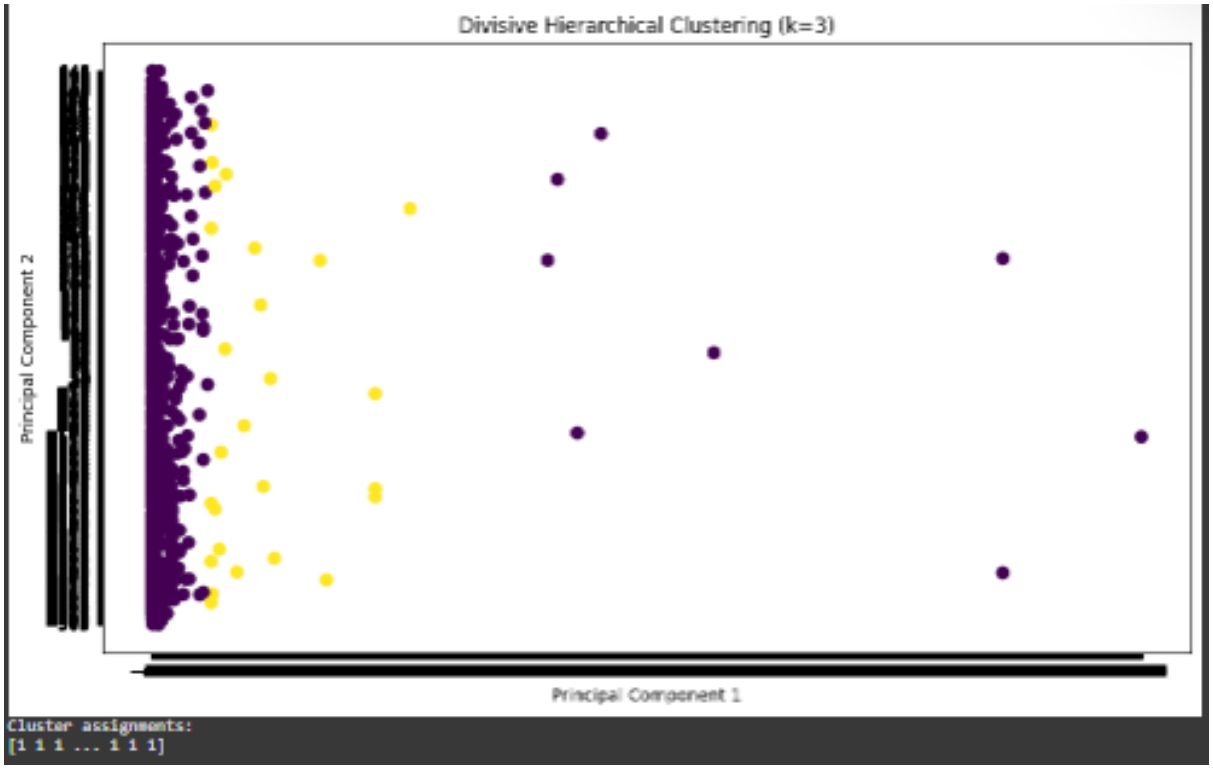
(a) Agglomerative clustering input



(b) Agglomerative clustering out

5.4 Divisive clustering(bottom-up approach)

This implements a Divisive Hierarchical Clustering approach, which is a top-down method for grouping data points into clusters. The dataset is first loaded and preprocessed, with missing values dropped and categorical features encoded into numerical values using LabelEncoder. To enhance visualization, the code applies Principal Component Analysis (PCA) to reduce the dataset's dimensions to two, making it easier to plot the clusters. The Divisive Hierarchical Clustering begins by treating all data points as a single cluster and iteratively splits the largest cluster into two smaller clusters using KMeans until the desired number of clusters is achieved. The resulting clusters are visualized in a scatter plot, where each point is colored according to its assigned cluster, and the principal components are used for the x and y axes. The final cluster labels are also printed, showing which data points belong to which cluster. This approach enables clear and interpretable clustering results while leveraging dimensionality reduction for better visualization.



(b) Divisive clustering output

```python
def divisive_clustering(X, n_clusters=3):
    # Start with all data points in one cluster
    clusters = [X]
    labels = np.zeros(len(X), dtype=int)  # Initialize labels for all data points

    while len(clusters) < n_clusters:
        # Split the largest cluster using KMeans
        largest_cluster_index = np.argmax([len(cluster) for cluster in clusters])
        largest_cluster = clusters.pop(largest_cluster_index)

        # Apply KMeans to split it into two sub-clusters
        kmeans = KMeans(n_clusters=2, random_state=42)
        cluster_labels = kmeans.fit_predict(largest_cluster)

        # Update labels for data points in the split cluster
        labels_in_largest_cluster = labels[np.isin(X, largest_cluster).all(axis=1)]
        labels_in_largest_cluster[cluster_labels == 0] = len(clusters)  # Assign new cluster label
        labels_in_largest_cluster[cluster_labels == 1] = len(clusters) + 1  # Assign next new label

        labels[np.isin(X, largest_cluster).all(axis=1)] = labels_in_largest_cluster  # Update original labels

        # Add the two sub-clusters
        clusters.extend([largest_cluster[cluster_labels == 0], largest_cluster[cluster_labels == 1]])

    return labels

n_clusters = 3  # Desired number of clusters
divisive_labels = divisive_clustering(X_reduced, n_clusters)
```

(a) Divisive clustering input

5.5 DBSCAN clustering

This performs clustering on a dataset using the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm. It first loads and preprocesses the dataset, handling missing values by dropping them, and encoding any categorical variables using LabelEncoder.

Next, the features of the dataset are standardized using StandardScaler to ensure that all features are on the same scale, which is important for DBSCAN as it is sensitive to the scale of the data.

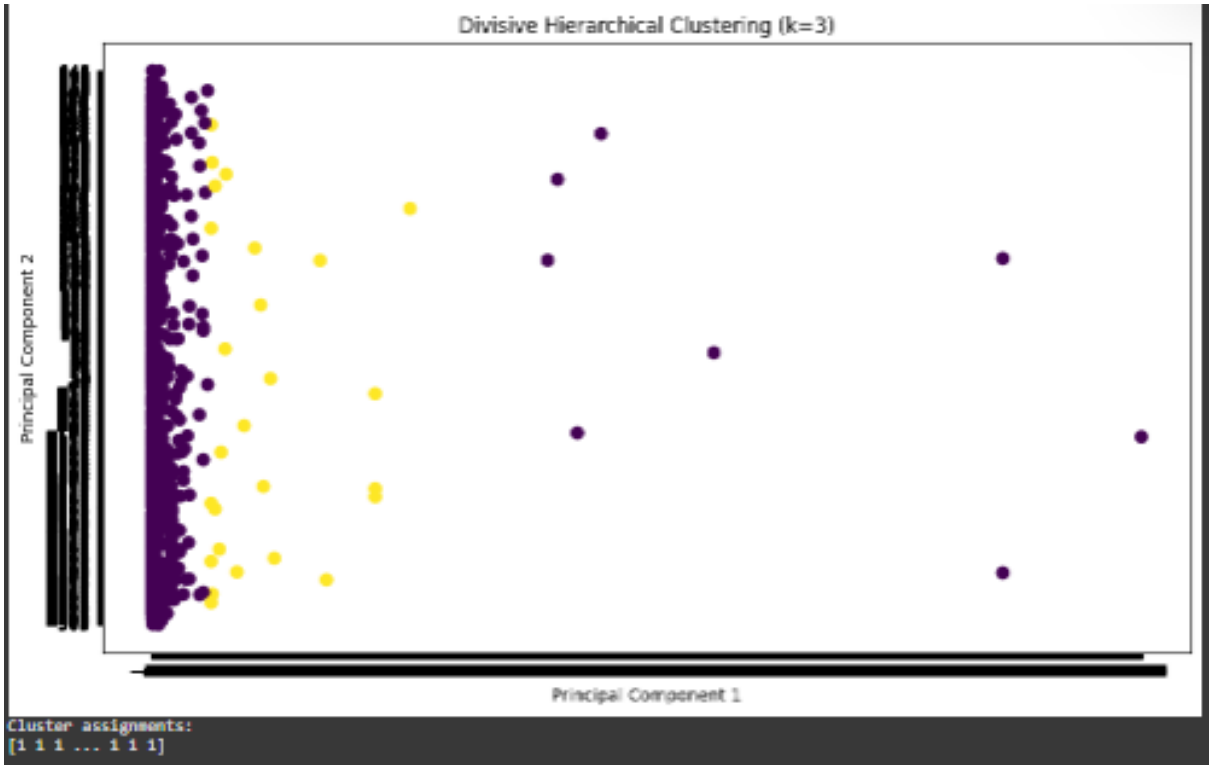
DBSCAN is then applied to the standardized data, with eps set to 0.5 (the maximum distance between two points to be considered neighbors) and min_samples set to 5 (the minimum number of points required to form a dense region, i.e., a cluster). The fit_predict method of DBSCAN assigns a cluster label to each data point, where core points are labeled with positive integers representing their respective clusters, and noise points (which do not belong to any cluster) are labeled with -1.

If the dataset contains more than two dimensions, PCA (Principal Component Analysis) is used to reduce the feature space to two dimensions for easier visualization. The resulting 2D data is then plotted with the cluster labels displayed in different colors, providing a clear visual representation of how DBSCAN has grouped the data points. The cluster assignments are printed at the end, allowing users to see the label for each data point.

This method provides an unsupervised learning approach to identifying clusters of varying shapes, while also handling noise points that do not belong to any cluster.

```
# Step 3: Standardize the features
X = dataset.values  # Features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Scaling features

# Step 4: Apply DBSCAN clustering
# You may need to tune the eps (neighborhood size) and min_samples (minimum points for a cluster) parameters
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan_labels = dbscan.fit_predict(X_scaled)

# Step 5: Visualize the clusters (if it's 2D, otherwise reduce dimensions)
# For 2D visualization, if the data has more than 2 features, you can apply PCA or TSNE
# If X_scaled has more than 2 dimensions, reduce it using PCA
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X_scaled)

# Step 6: Plot the DBSCAN clusters
plt.figure(figsize=(10, 6))
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=dbscan_labels, cmap='viridis', s=50)
plt.title("DBSCAN Clustering")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(label='Cluster Label')
plt.show()
```

(a) DBSCAN input



(b) DBSCAN output

5.6 FP growth algorithm using Pyspark

This applies FP-Growth, a frequent pattern mining algorithm in PySpark, to identify frequent itemsets and generate association rules. It first initializes a SparkSession and loads the dataset, filtering for relevant columns while handling missing values. The dataset is grouped by Date, and collect_set is used to ensure unique categories within each transaction. The FP-Growth model is then trained with specified minimum support and confidence levels. Afterward, it extracts and displays the frequent itemsets and association rules, while also measuring the execution time. Finally, the Spark session is terminated, ensuring efficient handling of large-scale data.

```
# Step 1: Initialize SparkSession
spark = SparkSession.builder \
    .appName("FP-Growth Example") \
    .getOrCreate()

# Step 2: Load the dataset
dataset_path = "/content/ADS_DATASET - Sheet1.csv"  # Replace with your CSV file path
dataset = spark.read.csv(dataset_path, header=True, inferSchema=True)

# Step 3: Select and preprocess relevant columns
dataset = dataset.select("Date", "Category").dropna()

# Use collect_set to get unique items in each transaction
transaction_data = dataset.groupBy("Date").agg(
    collect_set("Category").alias("Frequency")  # Changed to collect_set
)

# Step 5: Fit the FP-Growth model
start_time = time.time()

fp_growth = FPGrowth(itemsCol="Frequency", minSupport=0.001, minConfidence=0.5)
model = fp_growth.fit(transaction_data)
# Step 6: Extract frequent itemsets
frequent_itemsets = model.freqItemsets
frequent_itemsets_count = frequent_itemsets.count()

end_time = time.time()

print("Length of Frequent Itemsets is:")
print(frequent_itemsets_count)

print("The frequent itemsets are:")
frequent_itemsets.show(20, truncate=False)

# Step 7: Extract association rules
association_rules = model.associationRules
association_rules_count = association_rules.count()

print("Length of Association Rules is:")
print(association_rules_count)

print("The association rules are:")
association_rules.show(10, truncate=False)

# Step 8: Print time taken
print("Time taken to find frequent itemsets by FpGrowth:", end_time - start_time, "seconds")

# Stop the Spark session
spark.stop()
```

(a) input

```
Length of Frequent Itemsets is:
291
The frequent itemsets are:
+--------------------------------------------+----+
|items                                       |freq|
+--------------------------------------------+----+
|[Dividend earned on Shares]                 |12  |
|[Dividend earned on Shares, Food]           |2   |
|[Petty cash]                                |6   |
|[Petty cash, Food]                          |3   |
|[Tax refund]                                |2   |
|[Rent]                                      |4   |
|[Rent, Money transfer]                      |3   |
|[Rent, Public Provident Fund]               |3   |
|[Rent, Public Provident Fund, Money transfer]|2   |
|[maid]                                      |17  |
|[maid, Recurring Deposit]                   |2   |
|[maid, Recurring Deposit, Food]             |2   |
|[maid, Household]                           |3   |
|[maid, Household, Food]                     |2   |
|[maid, Transportation]                      |2   |
|[maid, Transportation, Food]                |2   |
|[maid, Money transfer]                      |6   |
|[maid, Money transfer, Food]                |5   |
|[maid, Public Provident Fund]               |7   |
|[maid, Public Provident Fund, Recurring Deposit]|2   |
+--------------------------------------------+----+
only showing top 20 rows

Length of Association Rules is:
209
The association rules are:
+--------------------------------------------------------+--------------------+------------------+------------------+---------------------+
|antecedent                                              |consequent          |confidence        |lift              |support              |
+--------------------------------------------------------+--------------------+------------------+------------------+---------------------+
|[Saving Bank account 1, maid, Money transfer, Food]     |[Public Provident Fund]|1.0               |50.82758620689655 |0.0013568521031207597|
|[Equity Mutual Fund F, Recurring Deposit]               |[Food]              |0.6666666666666666|1.5141242937853108|0.0027137042062415195|
|[Family, Transportation]                                |[Health]            |0.5               |9.098765432098766 |0.0013568521031207597|
|[Family, Transportation]                                |[Food]              |0.75              |1.7033898305084747|0.0020352781546811396|
|[Public Provident Fund, Apparel, Food]                  |[Transportation]    |0.6666666666666666|4.328928046989721 |0.0013568521031207597|
|[Public Provident Fund, Apparel, Food]                  |[Money transfer]    |1.0               |34.27906976744186 |0.0020352781546811396|
|[Public Provident Fund, Money transfer, Transportation, Food]|[Apparel]           |1.0               |32.75555555555555 |0.0013568521031207597|
|[Public Provident Fund, Recurring Deposit]              |[Money transfer]    |0.8333333333333334|28.56589147286822 |0.0033921302578018998|
|[Family, Other]                                         |[Food]              |0.6               |1.3627118644067797|0.0020352781546811396|
|[Saving Bank account 1, Public Provident Fund]          |[Food]              |0.5555555555555556|1.2617702448210923|0.0033921302578018998|
+--------------------------------------------------------+--------------------+------------------+------------------+---------------------+
only showing top 10 rows

Time taken to find frequent itemsets by FpGrowth: 2.6218912601470947 seconds
```

(b) output

```
# Create and apply pipeline
pipeline = Pipeline(stages=stages)
data = pipeline.fit(data).transform(data)

# Step 4: Split the data into training and testing sets
train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)

# Step 5: Train the Naive Bayes model
nb = NaiveBayes(smoothing=1.0, modelType="multinomial")
nb_model = nb.fit(train_data)

# Step 6: Make predictions
predictions = nb_model.transform(test_data)

# Step 7: Evaluate the model
evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)

print(f"Model Accuracy: {accuracy:.2f}")

# Step 8: Confusion Matrix
confusion_matrix = predictions.groupBy("label", "prediction").count().toPandas().pivot(index="label", columns="prediction", values="count").fillna(0)

# Visualize Confusion Matrix
plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=True, yticklabels=True)
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

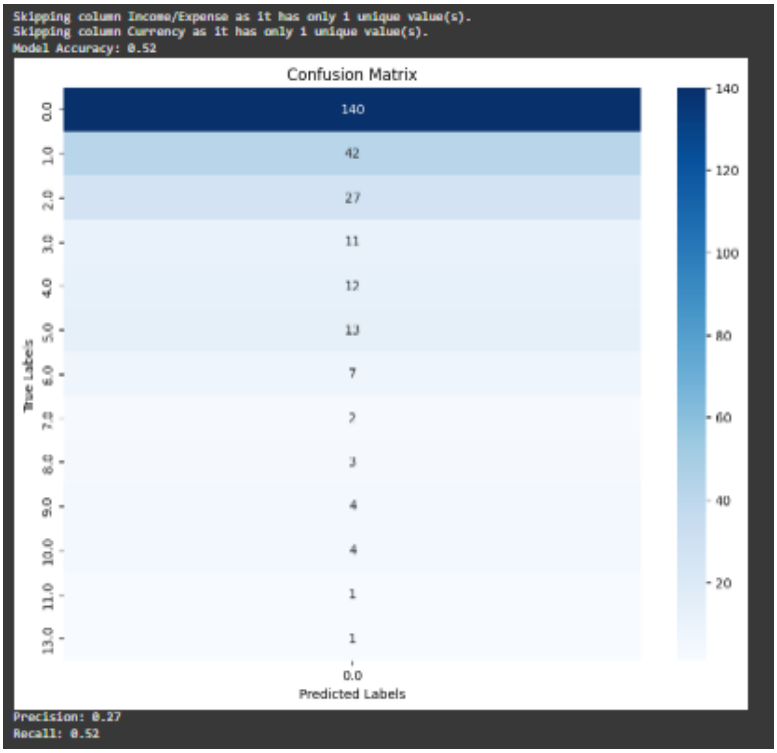
(b) input

5.7 Naive bayesian classification using Pyspark

This performs K-medoids clustering on a dataset using the KMedoids algorithm from sklearn_extra. First, the dataset is loaded, and any missing values are removed. Categorical variables are then encoded using LabelEncoder to convert them into numeric format, making the data suitable for clustering. Next, feature scaling is applied using StandardScaler to normalize the data, which improves the performance of the clustering algorithm. The K-medoids algorithm is then applied with three clusters, and the model is trained on the scaled data. The resulting clusters are assigned to each data point in the dataset. To visualize the clustering results, PCA (Principal Component Analysis) is used to reduce the data to two dimensions, allowing the clusters to be plotted in a scatter plot. The plot includes the actual data points representing the cluster centers (medoids), shown as red "X" markers. Finally, the original data points that serve as medoids are printed. This process provides insights into how the data is grouped and the representative points for each cluster.

```
# Get categorical columns
categorical_cols = [col for col, dtype in data.dtypes if dtype == "string"]

# Initialize stages for the pipeline
stages = []

# Process categorical columns
for col in categorical_cols:
    # Check distinct values
    distinct_count = data.select(col).distinct().count()
    if distinct_count > 1:
        # Index categorical column
        indexer = StringIndexer(inputCol=col, outputCol=f"{col}_indexed").fit(data)
        stages.append(indexer)

        # Encode indexed column
        encoder = OneHotEncoder(inputCol=f"{col}_indexed", outputCol=f"{col}_encoded")
        stages.append(encoder)
    else:
        print(f"Skipping column {col} as it has only {distinct_count} unique value(s).")

# Define feature columns
feature_columns = [f"{col}_encoded" for col in categorical_cols if f"{col}_encoded" in data.columns] + [
    col for col in data.columns if col not in categorical_cols + ["Category"]
]

# Assemble features
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
stages.append(assembler)
```

(a) input



(b) output

5.8 Clustering Data Using Bisecting K-Means in PySpark

This performs Bisecting K-Means clustering on a dataset using Apache Spark. It begins by initializing a Spark session and loading the dataset into a Spark DataFrame. The categorical columns (Mode, Category, Subcategory, Income/Expense) are converted to numeric values using StringIndexer, enabling them to be used in the clustering model. Then, the numerical and indexed columns are combined into a feature vector using VectorAssembler. Bisecting K-Means clustering is applied to these features, with the model set to create 3 clusters. Finally, the predicted cluster assignments are displayed for each data point, and the Spark session is stopped. This process effectively clusters the dataset based on the features provided, with categorical data appropriately handled through indexing.

```
+----------------+--------------------+-------------+--------------------+--------------------+------+--------------+--------+
|            Date|                Mode|     Category|         Subcategory|                Note|Amount|Income/Expense|Currency|
+----------------+--------------------+-------------+--------------------+--------------------+------+--------------+--------+
|20-09-2018 12:04|                Cash|Transportation|               Train|2 Place 5 to Place 0|  30.0|       Expense|     INR|
|20-09-2018 12:03|                Cash|         Food|              snacks|Idli medu Vada mi...|  60.0|       Expense|     INR|
|      19-09-2018|Saving Bank accou...| subscription|             Netflix|1 month subscription| 199.0|       Expense|     INR|
|17-09-2018 23:41|Saving Bank accou...| subscription|Mobile Service Pr...|   Data booster pack|  19.0|       Expense|     INR|
|16-09-2018 17:15|                Cash|    Festivals|        Ganesh Pujan|         Ganesh idol| 251.0|       Expense|     INR|
+----------------+--------------------+-------------+--------------------+--------------------+------+--------------+--------+
only showing top 5 rows

+----------------+--------------------+-------------+------+----------+
|            Date|                Mode|     Category|Amount|prediction|
+----------------+--------------------+-------------+------+----------+
|20-09-2018 12:04|                Cash|Transportation|  30.0|         0|
|20-09-2018 12:03|                Cash|         Food|  60.0|         0|
|      19-09-2018|Saving Bank accou...| subscription| 199.0|         0|
|17-09-2018 23:41|Saving Bank accou...| subscription|  19.0|         0|
|16-09-2018 17:15|                Cash|    Festivals| 251.0|         0|
|15-09-2018 06:34|         Credit Card| subscription| 200.0|         0|
|14-09-2018 05:39|                Cash|Transportation|  50.0|         0|
|13-09-2018 21:35|Saving Bank accou...|Transportation|  40.0|         0|
|13-09-2018 21:01|                Cash|         Food|  46.0|         0|
|      12-09-2018|         Credit Card| subscription| 667.0|         0|
|      11-09-2018|Saving Bank accou...|         Food| 650.0|         0|
|      11-09-2018|                Cash|         Food|  36.0|         0|
|      10-09-2018|                Cash|         Food|  36.0|         0|
|      08-09-2018|                Cash|       Family|  40.0|         0|
|      07-09-2018|                Cash|         Food|  37.0|         0|
|      06-09-2018|                Cash|         Food|  55.0|         0|
|      05-09-2018|                Cash|      Apparel|  77.0|         0|
|      05-09-2018|                Cash|         Food|  30.0|         0|
|      05-09-2018|                Cash|         Food|  40.0|         0|
|      05-09-2018|                Cash|         Food|  24.0|         0|
+----------------+--------------------+-------------+------+----------+
only showing top 20 rows
```

(b) output

```python
# Load CSV file into a Spark DataFrame
df = spark.read.csv("/content/ADS_DATASET - Sheet1.csv", header=True, inferSchema=True)

# Display the first few rows of the dataset to understand its structure
df.show(5)

# If you want to include categorical columns like "Mode" or "Category", convert them to numeric using StringIndexer
# Set handleInvalid to 'skip' to ignore rows with missing values
indexers = [
    StringIndexer(inputCol="Mode", outputCol="ModeIndex", handleInvalid="skip"),
    StringIndexer(inputCol="Category", outputCol="CategoryIndex", handleInvalid="skip"),
    StringIndexer(inputCol="Subcategory", outputCol="SubcategoryIndex", handleInvalid="skip"),
    StringIndexer(inputCol="Income/Expense", outputCol="IncomeExpenseIndex", handleInvalid="skip")
]

# Apply the indexers to the DataFrame
for indexer in indexers:
    df = indexer.fit(df).transform(df)

# Now, select the numerical columns and the newly created indexed columns for clustering
feature_columns = ["Amount", "ModeIndex", "CategoryIndex", "SubcategoryIndex", "IncomeExpenseIndex"]

# Assemble features into a single vector column
vec_assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
df = vec_assembler.transform(df)

# Apply Bisecting K-Means (Hierarchical clustering approximation)
bkm = BisectingKMeans(k=3, maxIter=10, seed=1, featuresCol="features", predictionCol="prediction")
model = bkm.fit(df)

# Make predictions
predictions = model.transform(df)
```

(a) input

5.9 AGNES Clustering

This uses Apache Spark and SciPy to perform hierarchical clustering (AGNES) on a dataset, followed by visualizing the results with a dendrogram. First, it initializes a Spark session and loads the dataset from a CSV file. Then, categorical columns like "Mode," "Category," "Subcategory," and "Income/Expense" are transformed into numeric values using StringIndexer. After that, a feature vector is created using VectorAssembler from the relevant columns. The data is collected as a NumPy array, and hierarchical clustering is applied using the linkage function from SciPy with the 'ward' method. Finally, a dendrogram is plotted to show how data points are grouped into clusters based on their distance. The process is wrapped up by stopping the Spark session.

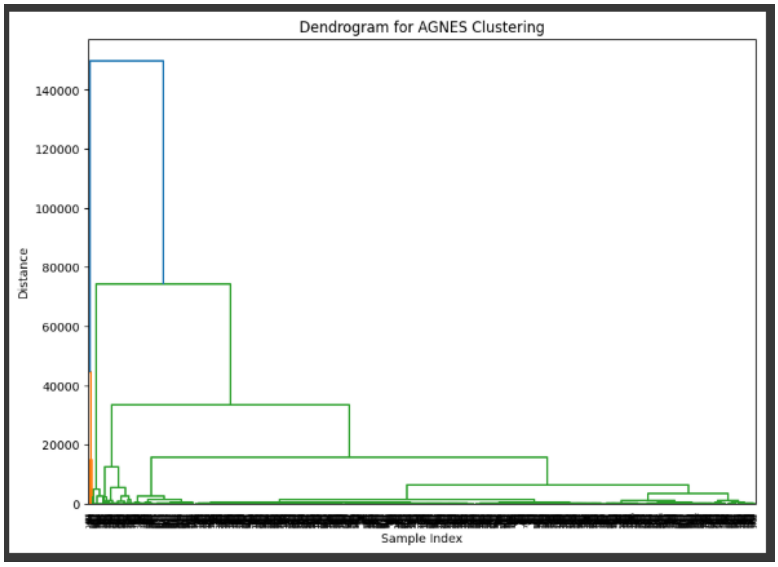
```
# Convert categorical columns to numerical values
# Set handleInvalid to 'skip' to ignore rows with missing values
indexers = [
    StringIndexer(inputCol="Mode", outputCol="ModeIndex", handleInvalid="skip"),
    StringIndexer(inputCol="Category", outputCol="CategoryIndex", handleInvalid="skip"),
    StringIndexer(inputCol="Subcategory", outputCol="SubcategoryIndex", handleInvalid="skip"),
    StringIndexer(inputCol="Income/Expense", outputCol="IncomeExpenseIndex", handleInvalid="skip")
]

# Apply indexers
for indexer in indexers:
    df = indexer.fit(df).transform(df)

# Select feature columns for clustering
feature_columns = ["Amount", "ModeIndex", "CategoryIndex", "SubcategoryIndex", "IncomeExpenseIndex"]

# Assemble features into a single vector
vec_assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
df = vec_assembler.transform(df)

# Collect the data as a NumPy array for dendrogram generation
data = np.array(df.select(feature_columns).collect())

# Perform hierarchical clustering using SciPy
linked = linkage(data, method='ward')  # 'ward' method minimizes variance within clusters

# Plot the dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linked, orientation='top', distance_sort='ascending', show_leaf_counts=True)
plt.title('Dendrogram for AGNES Clustering')
plt.xlabel('Sample Index')
plt.ylabel('Distance')
plt.show()
```

(a) input

Date	Mode	Category	Subcategory	Note	Amount	Income/Expense	Currency
20-09-2018 12:04	Cash	Transportation	Train	2 Place 5 to Place 0	30.0	Expense	INR
20-09-2018 12:03	Cash	Food	snacks	Idli medu Vada mi...	60.0	Expense	INR
19-09-2018	Saving Bank accou...	subscription	Netflix	1 month subscription	199.0	Expense	INR
17-09-2018 23:41	Saving Bank accou...	subscription	Mobile Service Pr...	Data booster pack	19.0	Expense	INR
16-09-2018 17:15	Cash	Festivals	Ganesh Pujan	Ganesh idol	251.0	Expense	INR

only showing top 5 rows

(b) output



(d) dendrogram

5.10 Classification Evaluation

- **Precision, Recall, and F1-Score:**
  - Precision: $\frac{TP}{TP+FP}$
  - Recall: $\frac{TP}{TP+FN}$
  - F1-Score: Harmonic mean of precision and recall.

Clustering Evaluation

- No explicit evaluation metrics in the code, but results can be interpreted using cluster centers (K-Means) and labels (DBSCAN).

Conclusion

This project offered insights into spending patterns through data preprocessing, normalization, feature selection, and visualizations.This analysis explores financial behaviors through various visualizations, highlighting spending patterns and preferred transaction modes. The **bar graph** shows that bank accounts and investments, particularly Saving Bank Account 1.In contrast, cash, debit cards, and credit cards account for smaller proportions of the total amounts spent. The graph highlights a preference for more secure and structured modes like savings and investments over immediate transaction options like cash and credit cards. The **pie chart** reveals that food is the largest expense category (43.7%), followed by other expenses and transportation. The **scatter plot** and **histogram** display the distribution of amounts across categories, showing some high-value transactions and frequent spending intervals.

The **box plot** indicates a skewed distribution with many outliers, while the **doughnut plot** and **area plot** illustrate the dominance of certain subcategories in total spending. The **violin plot** and **rug plot** show detailed variations in spending across categories. The **density plot** and **radar plot** provide insights into the overall distribution and average spending.

Frequent itemset mining using **Apriori** and **FP-Growth** reveals common transaction modes like "Cash" and "Credit Card," but no strong associations were identified between categories. Overall, the analysis highlights a preference for traditional banking and lifestyle expenses, with limited focus on investments and weak category associations.

The DIC algorithm identifies frequent itemsets for every 5 transactions using a support threshold of 0.01, uncovering itemsets such as 'Credit Card' with a support of 0.021 and 'Saving Bank account 1' with a support of 0.157, resulting in a total of 23 frequent itemsets. The Apriori algorithm applies hashing and candidate generation to identify 22 frequent itemsets, including 'Credit Card' with a support of 0.101 and 'Cash' with a support of 0.491. An enhanced Apriori with transaction reduction optimizes the process by removing irrelevant transactions during iterations, effectively identifying frequent itemsets like 'Credit Card' and 'Share Market Trading' while maintaining a total of 22 itemsets. Further refinement with Apriori and Closed Frequent Itemset Mining (CFI) discovers closed itemsets such as 'Credit Card', 'Cash' with a support of 0.029, ensuring that no supersets of these itemsets share the same support.

Additionally, the Pincer Search method focuses on maximal frequent itemsets, identifying 22 such itemsets, including 'Credit Card' with a support of 0.1011 and 'Recurring Deposit' with a support of 0.002. For supervised learning, a Decision Tree Classifier is trained on a preprocessed dataset, where missing values are handled, and categorical features are encoded. The model achieves high interpretability with performance metrics such as accuracy, precision, and recall, complemented by visual tools like confusion matrix heatmaps and tree plots.

In unsupervised learning, clustering techniques are employed. K-means clustering uses PCA for dimensionality reduction and groups data into three clusters, visualized in a scatter plot with cluster centroids marked. Similarly, K-medoids clustering identifies clusters using medoids as representatives, visualizing results in two dimensions. Agglomerative Hierarchical Clustering employs the 'ward' linkage method to build a dendrogram and determine the optimal number of clusters, while Divisive Hierarchical Clustering uses a top-down approach, splitting clusters iteratively with K-means and visualizing the results post-PCA. These methodologies provide comprehensive insights into the dataset, uncovering patterns, associations, and structures effectively.