Applied Data Science Project

December 7, 2024

Team Members:

- **SYED FARHAN** (CS23B2039)
- **MOHAMED AMJAD** (CS23B2013)
- **RAVITEJA B** (CS23B2011)

Abstract

In this "ADS Project," we performed data analysis and visualization using Python in a Google Colab environment. Our primary goal was to extract valuable insights from a grocery dataset by applying various data analytics techniques, including association rule mining, to uncover customer behaviors and purchasing patterns. These insights helped optimize business strategies, improve inventory management, and enhance customer satisfaction. We expanded our analysis to predictive analytics, using machine learning techniques like classification models and clustering to predict customer behavior, segment the customer base, and tailor marketing strategies accordingly. Additionally, we utilized PySpark for efficient large-scale data processing. Ultimately, our project combined both descriptive and predictive analytics to drive business growth, improve operational efficiency, and enhance the overall customer experience.

Descriptive Analytics

- **1. introduce data and explore the features**
- **2. Analysis Overview**
- **3. All kind of Plots and their inferences**

Predictive Analytics

1. Association Rules

- FP growth
- Apriori
- ARM
- Aclose algorithm
- Pincer search

2. Classification Models

- Decision tree models
- Naive Bayes models
- Multinomial models
- Support Vector Machine model

3. Clustering Models

- FP Growth
- DBSCAN Clustering
- HDBSCAN Clustering
- Agglomerative Clustering (AGNES)
- Hierarchical Clustering

4. PySpark

- FP Grwoth
- Naive Bayes model
- Agglomerative Clustering (AGNES)

1 Introduction

- **Member_number:** A unique ID for each member, likely a customer or account, identifying who made the transaction.

- **Date:** The specific date of the transaction, capturing when an item was purchased.

- **ItemDescription:** The product or item purchased, describing the type of goods (e.g., fruit, milk, etc.).

2 Analysis Overview

The analysis is divided into two main components:

- **Descriptive Analysis:** This component focuses on understanding sales trends through data visualization and data cleaning.
  - **Data Prepossessing:**
    - **Data Transformation:** To change the attributes to their appropriate datatype which is in a usable format for data mining.
  - **Plotting to enhance analysis:**
    - **Product Distribution:** Bar charts and Tree map to show the sales of various items.
    - **Time-Based Patterns:** Visualizations by year, month to observe seasonal trends.
    - **Customer type and their buying patterns:** Violin plot and box plot to show the distribution of purchase counts per customer and categorizing the customer based on their purchase counts.
- **Predictive Analysis:** This component aims to predict sales trends and discover associations within the dataset.
  - **Association Rule Mining:**
    - **Frequent Itemset Mining:** Identifying common item combinations and associations between different items.
    - **Association Rules:** Generating rules to understand relationships, such as which items are often bought together.
  - **Classification models:** Classification models in machine learning predict categorical labels based on input features. They are trained on labeled data and assign new data points to predefined classes. Common algorithms include decision trees, Naive Bayes, SVM, and logistic regression.
  - **Classification models:** Clustering models in machine learning group data points into clusters based on similarity. These unsupervised algorithms identify patterns and structures in data, with common techniques including K-means, DBSCAN, and hierarchical clustering.
- **Pyspark:** All the above models in PySpark for big data.

3 Descriptive Analysis

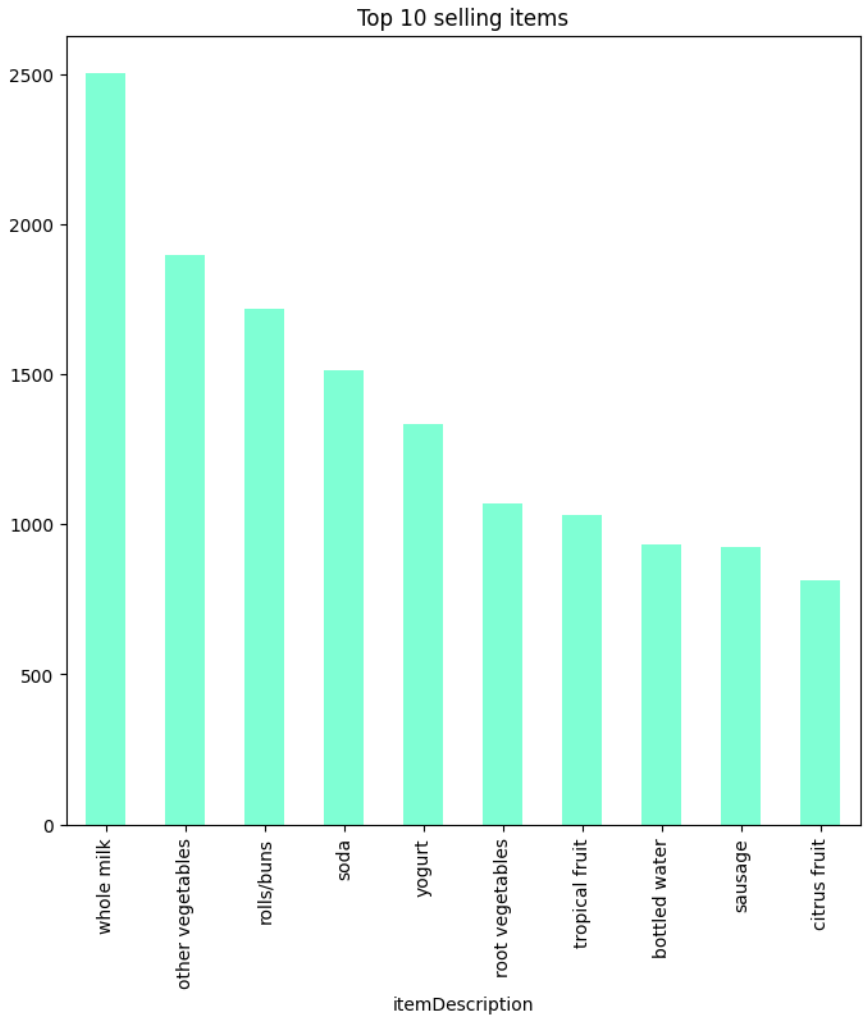
Plots and Inferences

- **Bar Chart for Top 10 Selling Items.**

Code:

```
# Bar Graph for the top 10 selling items
plt.figure(figsize=(8, 8))
df.itemDescription.value_counts().head(10).plot.bar(color='aquamarine')
plt.title("Top 10 Selling Items") # Proper spacing
plt.xlabel("Item Description")   # Proper spacing
plt.ylabel("Sales Count")        # Proper spacing
plt.show()
```

Plot:



- **Top Selling item:** Whole milk, other vegetables, rolls/buns are the top selling items.
- **Healthy Demand:** Most top selling items are healthy in nature, which indicates customers prefers healthy products.

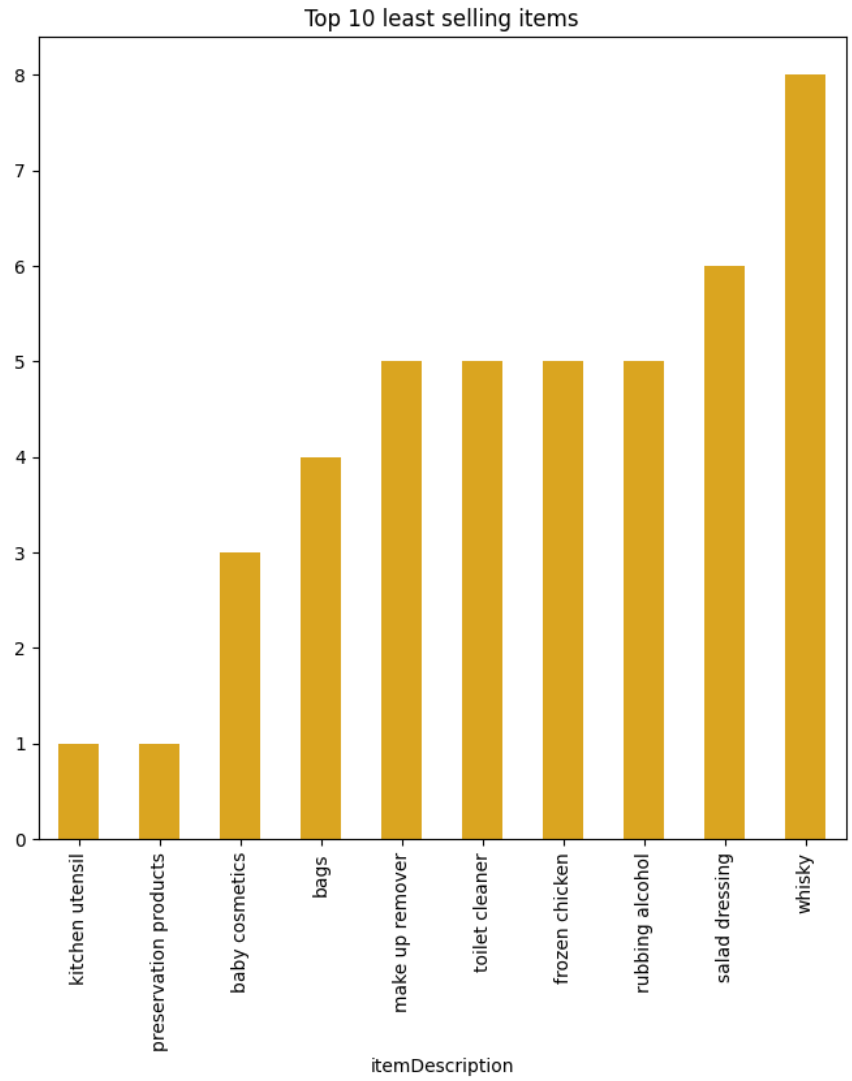
- **Balanced Basket:** Products like soda and rolls/buns show the demand for snacks and necessities.
- **Shelf Space:** Allocate more space for top-selling items to improve availability.
- **Diverse Needs:** Popular items include dairy, beverages, and produce, indicating varied customer preferences.

- **Bar Chart for Least 10 Selling Items**

Code:

```
# Bar Graph for the last 10 selling items
plt.figure(figsize = (8,8))
df.itemDescription.value_counts().tail(10).sort_values().plot.bar(color='
    goldenrod')
plt.title('Top 10 least selling items')
```

Plot:



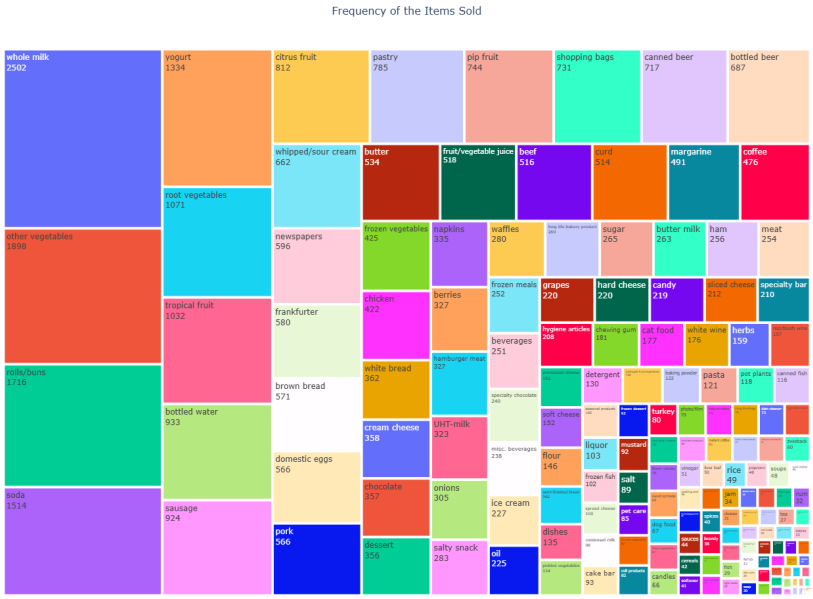
- **Whisky**: Low sales suggest limited demand or missed promotional opportunities.
- **Household Items**: Products like toilet cleaner and rubbing alcohol may have infrequent purchases or alternative sources.
- **Non-Food Products**: Low demand for kitchen utensils and preservation items might benefit from bundling with complementary products.
- **Niche Products**: Baby cosmetics and bags could perform better with targeted advertising.
- **Diverse Categories**: Various product types show low sales, indicating a potential misalignment with customer preferences.

• Treemap for the Frequency of Items Sold

Code:

```
frequency_of_items = df.groupby(pd.Grouper(key = 'itemDescription')).size().
    reset_index(name = 'count')
fig = px.treemap(frequency_of_items, path = ['itemDescription'], values = 'count
    ')
fig.update_layout(
    title_text = 'Frequency␣of␣the␣Items␣Sold',
    title_x = 0.5, title_font = dict(size = 16),
    height = 999
)
fig.update_traces(textinfo = "label+value")
fig.show()
```

Plot:

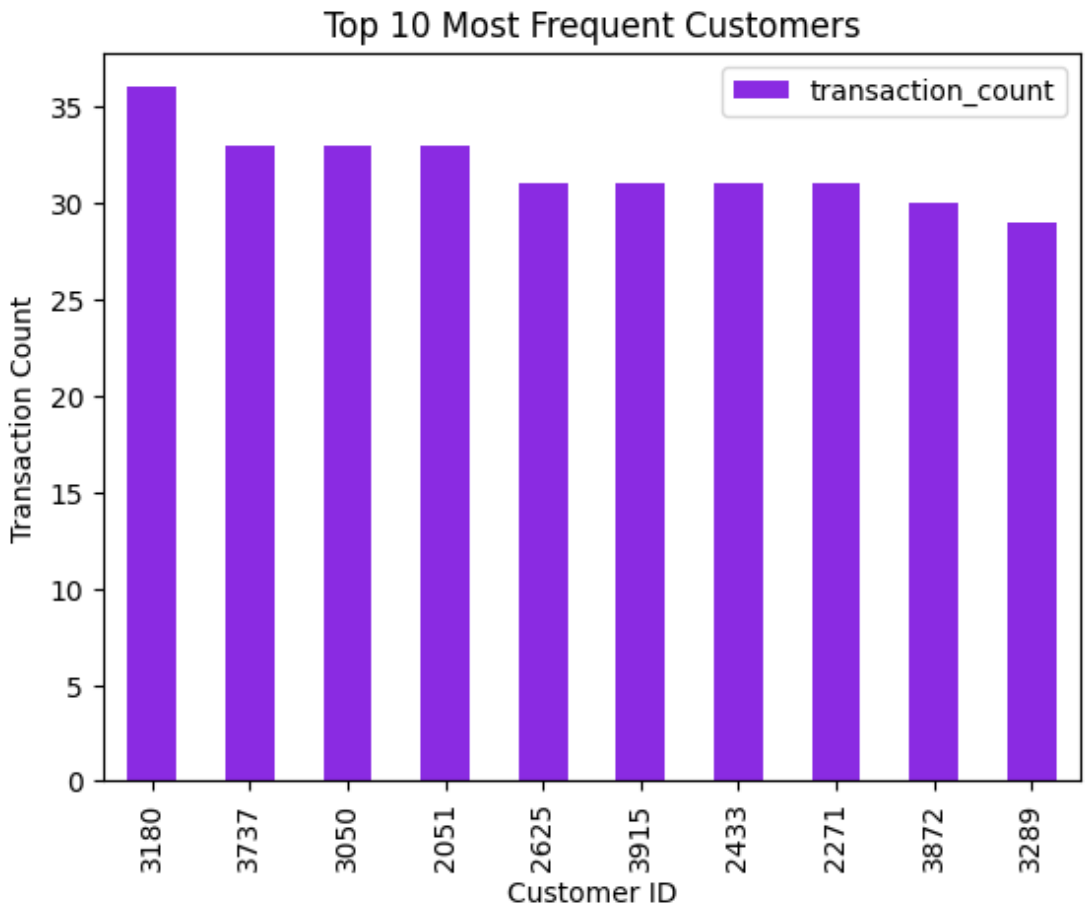


- **Top 10 Most Frequent Customers**

Code:

```
customer_frequency = df['Member_number'].value_counts()

# Converting to a DataFrame for easy handling and renaming the column
customer_frequency = customer_frequency.rename_axis('Member_number').reset_index
    (name='transaction_count')

# Displaying Top 10 most frequent customers
top_customers = customer_frequency.head(10)

top_customers.plot(kind='bar', x='Member_number', y='transaction_count', color='
    blueviolet')
plt.title("Top 10 Most Frequent Customers")
plt.xlabel("Customer ID")
plt.ylabel("Transaction Count")
plt.show()
```

Plot:



  - **Customer Frequency**: The store has a mix of loyal customers and occasional shoppers, with some making many more transactions than others.

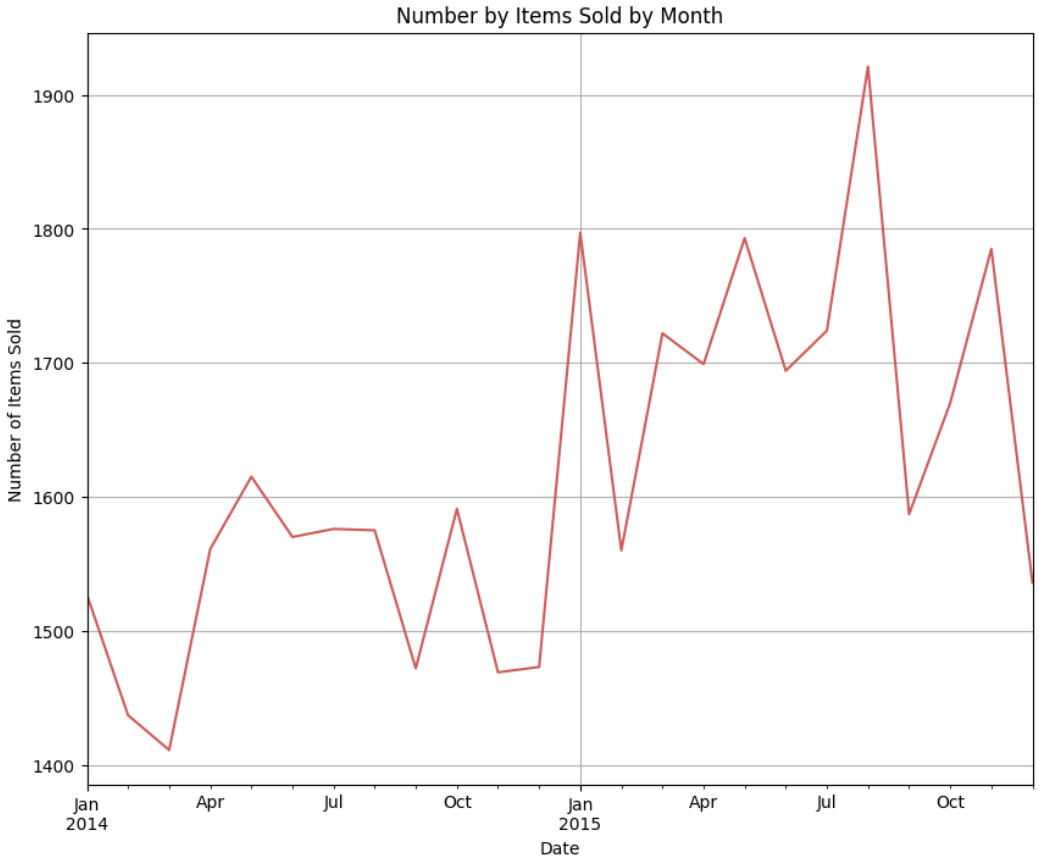
- **Top Customers**: The top 10 customers, with 28-36 transactions, show strong engagement, likely due to factors like satisfaction, loyalty programs, or personalized marketing.
- **Segmentation**: Customers can be segmented into "high-value" (frequent buyers) and "occasional" (low-frequency buyers) groups.
- **Marketing**: Targeted promotions for high-frequency customers could boost engagement even further.
- **Inventory**: Tracking frequent customers helps identify popular products, ensuring optimal stock levels.

- **Number of Items Sold by Month**

Code:

```
# Plotting the number of items sold in a month
df_date = df.set_index(['Date']) # Setting date as index for plotting purpose
df_date
df_date.resample("M")['itemDescription'].count().plot(figsize = (10,8),
       grid = True, color= 'indianred', title = "Number by Items Sold by Month")
           .set(xlabel = "Date",
       ylabel = "Number of Items Sold")
```

Plot:



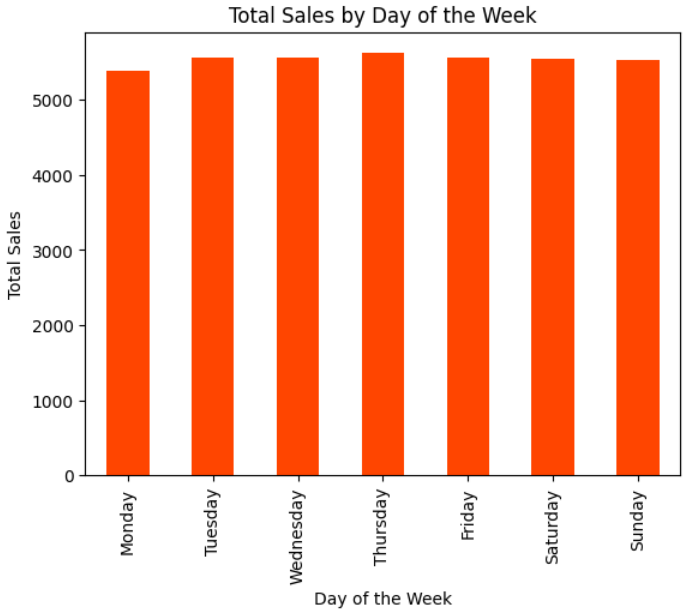
- **Seasonal Fluctuations**: Sales show a seasonal pattern, with peaks in the first quarter and dips in the third quarter.
- **Growth in 2015**: Sales in 2015 were higher than in 2014, indicating business growth.
- **Sharp Decreases**: Significant drops in sales occurred, particularly in the second quarter of 2015.
- **Unpredictable Peaks**: Sales were volatile, with unexpected spikes and drops throughout the year.
- **Potential for Improvement**: The seasonal and unpredictable sales suggest opportunities for better sales forecasting and inventory management.

• Total Sales by Days of the Week

Code:

```
df['day_of_week'] = df['Date'].dt.day_name() # Gets full day name, e.g., 'Monday
    '

# Group by the day of the week and sum quantities to get total sales
sales_by_day = df.groupby('day_of_week')['Member_number'].count().reindex(
    ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
        ]
)
sales_by_day.plot(kind='bar', color='orangered')
plt.title("Total␣Sales␣by␣Day␣of␣the␣Week")
plt.xlabel("Day␣of␣the␣Week")
plt.ylabel("Total␣Sales")
plt.show()
```

Plot:



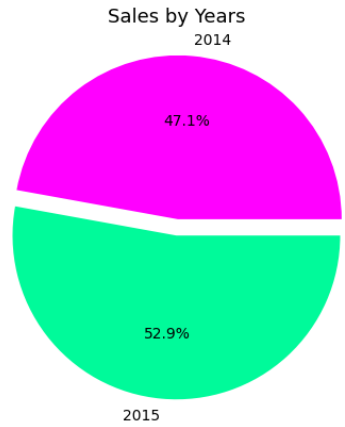
- **Total Sales on Thursdays**: Thursdays have slightly higher sales than other days, making them the day with the most sales.
- **Consistent Sales**: There is little variation in sales from day to day throughout the week.
- **Personnel Levels**: Equivalent sales levels imply that personnel needs are dispersed equally throughout the week.
- **Promotion Timing**: There is no significant loss in revenue if sales activities are planned on any day.
- **Customer Patterns**: The information points to regular purchasing behaviors devoid of day-specific preferences.

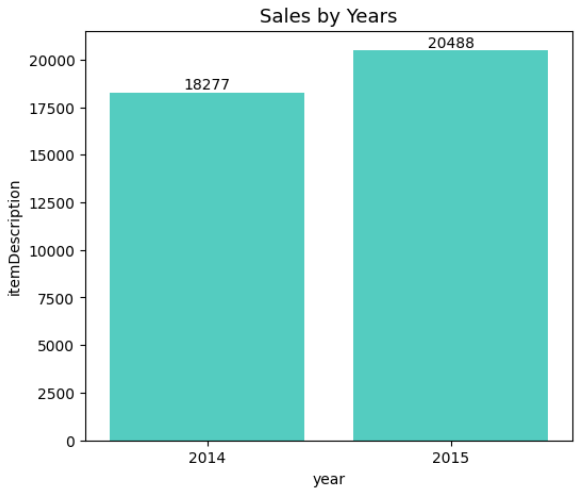
• Pie Chart and Bar Chart for Sales by Years

Code:

```
# Number of items sold in a year
datayears = df.groupby('year')['itemDescription'].count().reset_index()
datayearsy = datayears['year'].tolist()
dataitem = datayears['itemDescription'].tolist()

# Pie chart
plt.figure(figsize = (13, 5))
plt.subplot(1, 2, 1)
explode = (0.1, 0)
colors = sns.color_palette('Paired')
plt.pie(dataitem, labels = datayearsy, autopct = '%1.1f%%',colors=['fuchsia','
    mediumspringgreen'], explode = explode)
plt.title('Sales␣by␣Years', size = 13)

# Bar chart
plt.subplot(1, 2, 2)
ax = sns.barplot(x = 'year', y = 'itemDescription', data = datayears, color='
    turquoise')
for i in ax.containers:
    ax.bar_label(i)
plt.title('Sales␣by␣Years', size = 13)
plt.show()
```

Plot:





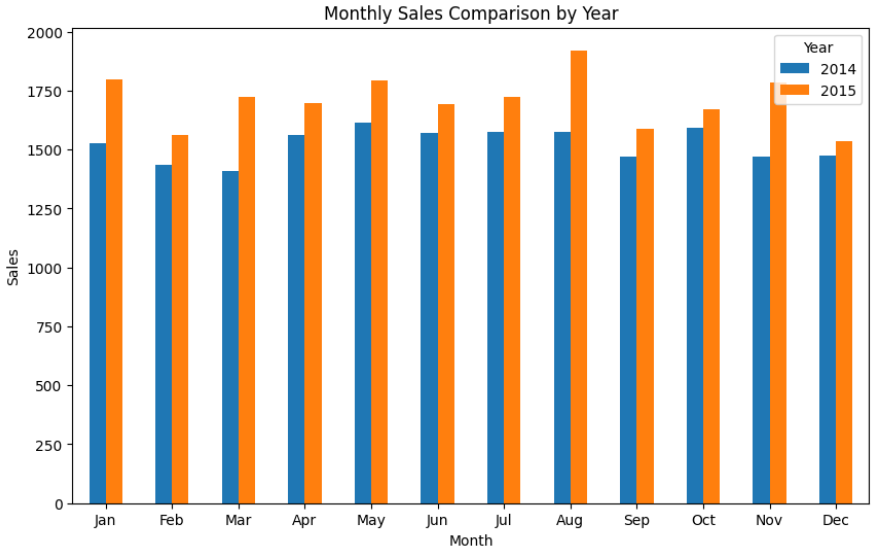
- **Total Sales:** In 2014, total sales reached 18,277 units, whereas in 2015, total sales reached 20,488 units.
- **Steady Growth:** Sales in 2015 exceeded 2014 by a moderate margin, showing growth.
- **Consistent Performance:** Sales figures between the two years are relatively close, indicating stable business.
- **Positive Trend:** The increase in sales suggests successful strategies or market expansion in 2015.
- **Strategic Analysis:** Understanding factors behind 2015's growth could optimize future sales strategies.

- **Double Bar Graph for Monthly Sales Comparison by Year**

Code:

```
monthly_sales = df.groupby(['year', 'month'])['itemDescription'].sum().
    reset_index()
pivot_df = monthly_sales.pivot(index='month', columns='year', values='
    itemDescription')
count_pivot_df = df.pivot_table(index='month', columns='year', values='
    itemDescription', aggfunc='count')
count_pivot_df.plot(kind='bar', figsize=(10, 6))
plt.title("Monthly Sales Comparison by Year")
plt.xlabel("Month")
plt.ylabel("Sales")

# Set the x-axis ticks to month abbrs
month_abbrs = [calendar.month_abbr[month] for month in pivot_df.index]
plt.xticks(ticks=range(len(month_abbrs)), labels=month_abbrs, rotation=0)

plt.legend(title="Year")
plt.show()
```

Plot:



- **Highest sales months:** The highest monthly sales ever recorded is in August 2015, followed by January 2015 and May 2015.
- **Least sales month:** The lowest monthly sales ever recorded is in March 2014.
- **Improved Sales :** All monthly sales in 2015 are higher than in 2014 when compared with their corresponding months.
- **Consistent Growth:** The consistent pattern of higher bars for 2015 indicates steady growth rather than fluctuations or season-specific spikes.
- **Consistency in Monthly Sales:** Each month shows a similar trend in both years, with no major disruptions or sudden drops in any month. This consistency implies a stable business environment, with predictable seasonal variations rather than unexpected volatility.

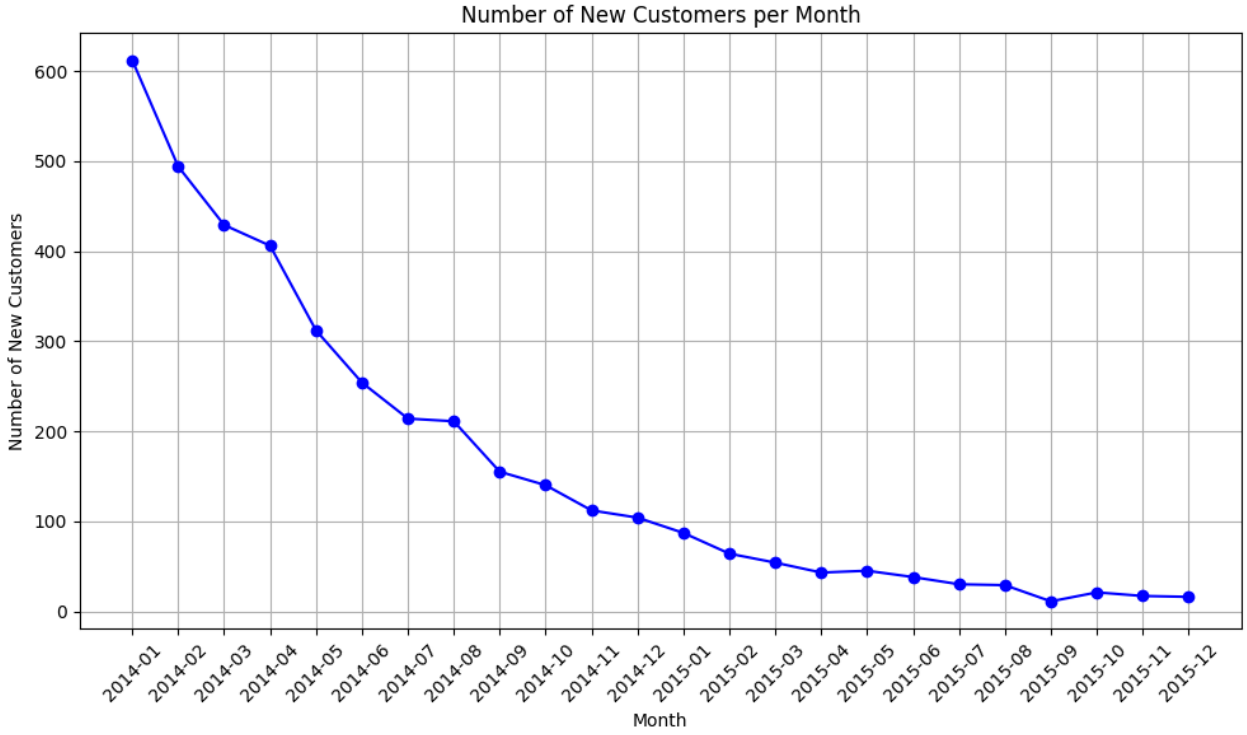
• Line Chart for Number for New Customers per Month

Code:

```
df['year_month'] = df['Date'].dt.to_period('M')

# Determining the first month each customer appeared
first_purchase = df.groupby('Member_number')['year_month'].min().reset_index()
first_purchase.columns = ['Member_number', 'first_purchase_month']

# Merge back to identify new customers per month
df = df.merge(first_purchase, on='Member_number')

# Filter for new customers and count them by month
new_customers_per_month = df[df['year_month'] == df['first_purchase_month']].
    groupby('year_month')['Member_number'].nunique()

# Convert to DataFrame for readability
```

```
new_customers_per_month = new_customers_per_month.reset_index()
new_customers_per_month.columns = ['Month', 'New_Customers']

# Plotting line chart
plt.figure(figsize=(10, 6))
plt.plot(new_customers_per_month['Month'].astype(str), new_customers_per_month['
    New_Customers'], marker='o', color='b')
plt.title("Number of New Customers per Month")
plt.xlabel("Month")
plt.ylabel("Number of New Customers")
plt.xticks(rotation=45) # Rotate x-axis labels for readability
plt.grid(True)
plt.tight_layout()
plt.show()
```

Plot:



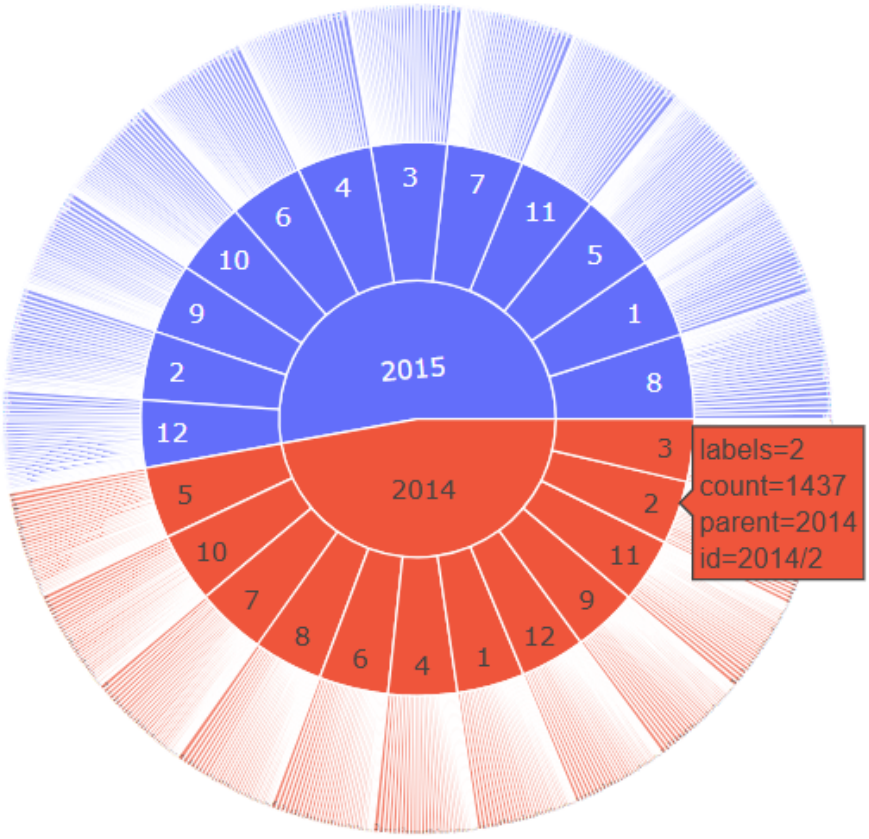
- **Steepest Decrease in New Customers:** The sharpest decrease in number of new customers is seen between the months January and February in 2014.
- **Stability after 2015-10 :** The number of new customers is the same after 2015-10.
- **Unusual Similarity:** The number of new customers is abnormally the same in months July and August in 2014 after which it decreases again.
- **Steady Decrease:** The number of new customers decreases steadily after September in 2014.
- **Overall Trend:** Overall the number of new customers shows a decreasing trend with first plummeting then a steady decrease over the span of two years.

- **Sunburst of Year, Month and Day**

Code:

```
fig = px.sunburst(df, path=["year",'month', 'day'])
fig.show()
```

Plot:



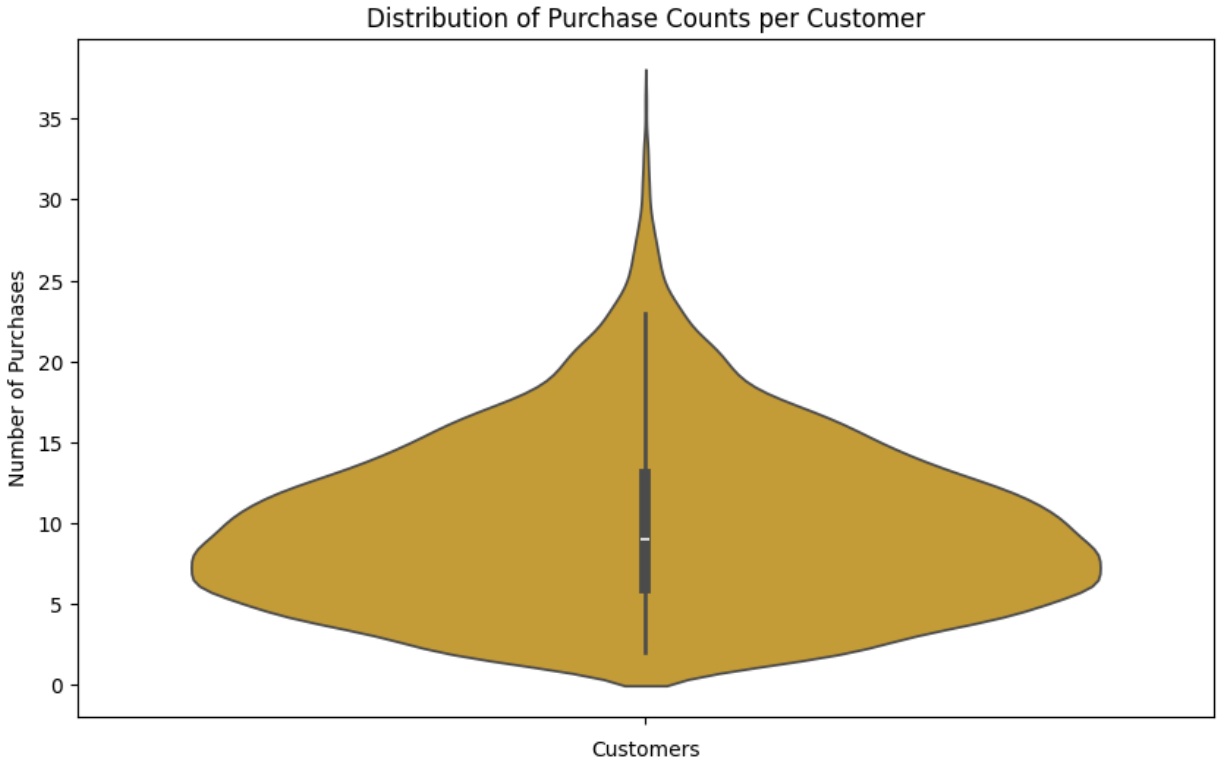
- **Least sales month in 2014:** March is the month which has got the least sales in 2014 but performed better in next year.
- **Least sales month in 2015 :** December is the month which has got the least sales in 2015, but performed better in previous year.
- **Least sales month in both years :** February is observed to be the second last in monthly sales in both the years.
- **Highest sales month in almost both years:** May month has the highest sales in 2014 and ranked the 3rd highest sales month in the next year.
- **Balanced sales in both years:** April month stands as the center point in sales in both the years.

- **Violin Plot for Distribution of Purchase Counts**

Code:

```
purchase_counts = df.groupby('Member_number').size().reset_index(name='
    purchase_count')
plt.figure(figsize=(10, 6))
sns.violinplot(data=purchase_counts, y='purchase_count', color='goldenrod')
plt.title('Distribution␣of␣Purchase␣Counts␣per␣Customer')
plt.xlabel('Customers')
plt.ylabel('Number␣of␣Purchases')
plt.show()
```

Plot:



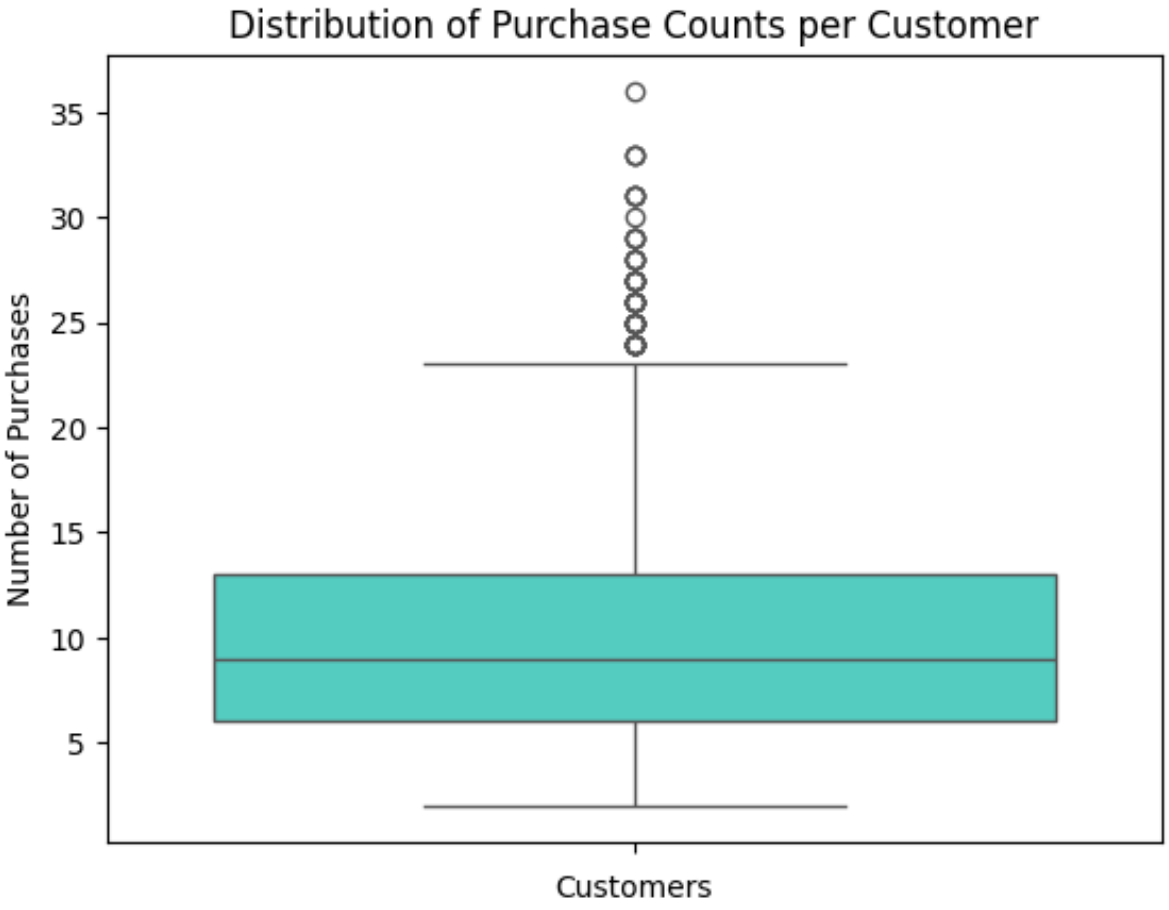
- **Distribution :** Most purchase counts are spread in between 5-10.
- **Min & Max Comparison :** The min purchase count which is 0 has more width than max purchase counts which is 36.
- **Expansion of Width:** The width of the violin plots rises steeply from the start.
- **Shrinking of Width:** The width of the violin plot decreases significantly after about 5-10 purchase counts.
- **Zero Width:** The width of the violin plot almost becomes zero, as it converges into a line after 33 purchase counts.

- **Box Plot for Distribution of Purchase Counts**

Code:

```
sns.boxplot(data=purchase_counts, y='purchase_count',color='turquoise')
plt.title('Distribution of Purchase Counts per Customer')
plt.xlabel('Customers')
plt.ylabel('Number of Purchases')
plt.show()
purchase_counts['purchase_count'].describe()
```

Plot:



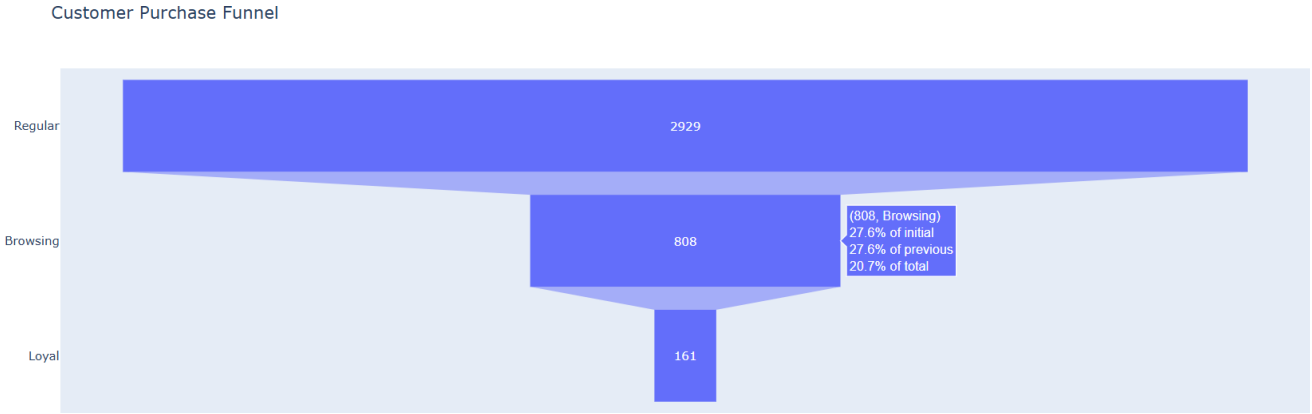
  - **First Quartile (Q1):** The 25th percentile value of the purchase counts is 6.
  - **Median :** The second quartile of the purchase counts per customer is 9.
  - **Third Quartile (Q3):** The 75th percentile value of the purchase counts is 13.
  - **Outliers:** There are 10 outliers, whose values are above 23.5 which is the upper limit of the box plot.
  - **Max Value :** The maximum value of the outlier is 36 purchase counts.

- **Funnel Chart of Customer Purchase Counts**

Code:

```
df['purchase_count'] = df.groupby('Member_number')['itemDescription'].transform(
    'count')
df['customer_stage'] = pd.cut(df['purchase_count'], bins=[0, 5, 20, float('inf')
    ], labels=['Browsing', 'Regular', 'Loyal'])

stage_counts = df.groupby('customer_stage')['Member_number'].nunique().
    sort_values(ascending=False)

# Plotting funnel chart
fig = go.Figure(go.Funnel(
    y = stage_counts.index,
    x = stage_counts.values
))
fig.update_layout(title='Customer Purchase Funnel')
fig.show()
```

Plot:



- **Customer type:** The customers are categorized based on the purchase counts,
  count: 0-5 = browsing customer,
  count: 5-20 = regular customer,
  count: above 20 = loyal customer.
- **Regular Customer Majority:** Majority of the customers are regular.
- **Browsing Customers:** Approximately 21% of customers are browsing.
- **Loyal Customers:** Only around 4% of the customers are loyal.
- **Overall Conversion Rate:** From 2,929 "Regular" customers, only 161 reach the "Loyal" stage, resulting in an overall conversion rate of approximately 5.5%. This shows that the majority of customers do not reach high levels of loyalty or repeated engagement.

4 Predictive Analysis

4.1 Frequent Itemset and Association Rule Mining

- Retail managers can use Association Rule Mining to find relevant shopping patterns, ultimately improving consumer satisfaction and retail efficiency.
- **Identifying association Between Products:** The technique finds associations between items that are frequently bought together. For example, it might uncover that bread and milk are often purchased in the same transaction.
- **Excluding Infrequent Itemsets:** By excluding less common itemsets, high minimum support focuses the algorithm on typical buying patterns, ignoring rare or unusual purchases to highlight frequently purchased items and shopping combinations.
- **Increased Efficiency:** A higher support threshold reduces the number of itemsets the algorithm must analyze, improving computational speed and efficiency, particularly in large transaction datasets where evaluating numerous item combinations can be computationally expensive.

4.1.1 Apriori Algorithm

Apriori relies on the principle that all subsets of frequent itemsets must also be frequent, which reduces the search space for item combinations in grocery data.

It uses the downward-closure property to discard infrequent itemsets early, ensuring only common product combinations are analyzed, making the algorithm efficient for large datasets.

Code:

```
import numpy as np
import pandas as pd
import time
from mlxtend.frequent_patterns import fpgrowth,apriori
from mlxtend.frequent_patterns import association_rules
df = pd.read_csv('Groceries_dataset.csv')
pivot_df = df.pivot_table(index='Member_number', columns='itemDescription',
    aggfunc='size', fill_value=0)

# Convert to one-hot encoding (binary)
one_hot_df = pivot_df.applymap(lambda x: 1 if x > 0 else 0).reset_index()
one_hot_df.to_csv('grocery_fp.csv', index=False)

df = pd.read_csv('grocery_fp.csv') # new updates dataset one-hot encoded

df = df.drop('Member_number', axis=1) # removing unique member numbers

frequent_itemsets = apriori(df, min_support=0.1, use_colnames=True)
print(frequent_itemsets)
```

Output:

Frequent itemsets generated by APRIORI

	support	itemsets
0	0.119548	(beef)
1	0.158799	(bottled beer)
2	0.213699	(bottled water)
3	0.135967	(brown bread)
4	0.126475	(butter)
5	0.165213	(canned beer)
6	0.100564	(chicken)
7	0.185480	(citrus fruit)
8	0.114931	(coffee)
9	0.120831	(curd)
10	0.133145	(domestic eggs)
11	0.137506	(frankfurter)
12	0.102617	(frozen vegetables)
13	0.124936	(fruit/vegetable juice)
14	0.116983	(margarine)
15	0.139815	(newspapers)
16	0.376603	(other vegetables)
17	0.177527	(pastry)
18	0.170600	(pip fruit)
19	0.132376	(pork)
20	0.349666	(rolls/buns)
21	0.230631	(root vegetables)
22	0.206003	(sausage)
23	0.168291	(shopping bags)
24	0.313494	(soda)
25	0.233710	(tropical fruit)
26	0.154695	(whipped/sour cream)
27	0.458184	(whole milk)
28	0.282966	(yogurt)
29	0.112365	(bottled water, whole milk)
30	0.146742	(rolls/buns, other vegetables)
31	0.124166	(soda, other vegetables)
32	0.191380	(whole milk, other vegetables)
33	0.120318	(yogurt, other vegetables)
34	0.119805	(soda, rolls/buns)
35	0.178553	(whole milk, rolls/buns)
36	0.111339	(yogurt, rolls/buns)
37	0.113135	(root vegetables, whole milk)
38	0.106978	(sausage, whole milk)
39	0.151103	(soda, whole milk)
40	0.116470	(tropical fruit, whole milk)
41	0.150590	(yogurt, whole milk)

4.1.2 FP-Growth Algorithm

FP-Growth uses an FP-tree to find frequent itemsets without generating candidate sets, making it faster and more memory-efficient than Apriori, especially in large grocery datasets.

By compacting data into an FP-tree, it efficiently identifies product associations, like milk and bread, while minimizing memory usage, even with high transaction volumes.

Efficient Mining of Frequent Itemsets: The FP-tree compresses frequent itemset data, reducing complexity and improving speed compared to Apriori, especially with large datasets.

Handling Large Datasets: The FP-tree avoids generating candidate itemsets, improving memory usage and processing speed, making it efficient for large datasets.

Code:

```
import numpy as np
import pandas as pd
from mlxtend.frequent_patterns import fpgrowth, apriori
from mlxtend.frequent_patterns import association_rules

df = pd.read_csv('Groceries_dataset.csv')
pivot_df = df.pivot_table(index='Member_number', columns='itemDescription',
    aggfunc='size', fill_value=0)

# Convert to one-hot encoding (binary)
one_hot_df = pivot_df.applymap(lambda x: 1 if x > 0 else 0).reset_index()
one_hot_df.to_csv('grocery_fp.csv', index=False)

df = pd.read_csv('grocery_fp.csv') # new updated dataset one-hot encoded

df = df.drop('Member_number', axis=1) # removing unique member numbers

frequent_itemsets = fpgrowth(df, min_support=0.1, use_colnames=True)
print(frequent_itemsets)
```

Output:

Frequent itemsets generated by FPGROWTH

	support	itemsets
0	0.458184	(whole milk)
1	0.313494	(soda)
2	0.282966	(yogurt)
3	0.206003	(sausage)
4	0.177527	(pastry)
5	0.165213	(canned beer)
6	0.349666	(rolls/buns)
7	0.154695	(whipped/sour cream)
8	0.137506	(frankfurter)
9	0.120831	(curd)
10	0.119548	(beef)
11	0.376603	(other vegetables)
12	0.233710	(tropical fruit)
13	0.126475	(butter)
14	0.102617	(frozen vegetables)
15	0.230631	(root vegetables)
16	0.170600	(pip fruit)
17	0.168291	(shopping bags)
18	0.116983	(margarine)
19	0.213699	(bottled water)
20	0.158799	(bottled beer)
21	0.100564	(chicken)
22	0.133145	(domestic eggs)
23	0.139815	(newspapers)
24	0.114931	(coffee)
25	0.185480	(citrus fruit)
26	0.135967	(brown bread)
27	0.124936	(fruit/vegetable juice)
28	0.132376	(pork)
29	0.151103	(soda, whole milk)
30	0.119805	(soda, rolls/buns)
31	0.124166	(soda, other vegetables)
32	0.150590	(yogurt, whole milk)
33	0.120318	(yogurt, other vegetables)
34	0.111339	(yogurt, rolls/buns)
35	0.106978	(sausage, whole milk)
36	0.178553	(whole milk, rolls/buns)
37	0.146742	(rolls/buns, other vegetables)
38	0.191380	(whole milk, other vegetables)
39	0.116470	(tropical fruit, whole milk)
40	0.113135	(root vegetables, whole milk)
41	0.112365	(bottled water, whole milk)

4.1.3 Association Rule Mining

Association Rule Mining uncovers relationships between products, such as items frequently bought together, helping retailers understand customer purchasing patterns in grocery transactions.

By analyzing frequent itemsets, it generates association rules (e.g., "if a customer buys bread, they often buy butter"), providing actionable insights for product placement and promotions.

Discovering Buying Patterns: Association Rule Mining identifies products frequently bought together, helping retailers understand customer preferences and optimize sales strategies.

Actionable Insights for Marketing: It generates rules like "if a customer buys bread, they often buy butter," enabling targeted promotions and personalized marketing.

Code:

```
import numpy as np
import pandas as pd
from mlxtend.frequent_patterns import fpgrowth, apriori
from mlxtend.frequent_patterns import association_rules

df = pd.read_csv('Groceries_dataset.csv')
pivot_df = df.pivot_table(index='Member_number', columns='itemDescription',
    aggfunc='size', fill_value=0)

# Convert to one-hot encoding (binary)
one_hot_df = pivot_df.applymap(lambda x: 1 if x > 0 else 0).reset_index()
one_hot_df.to_csv('grocery_fp.csv', index=False)

df = pd.read_csv('grocery_fp.csv') # new updates dataset one-hot encoded

df = df.drop('Member_number', axis=1) # removing unique member numbers

rules = association_rules(frequent_itemsets, metric="confidence", min_threshold
    =0.2)
print("Top␣20␣association␣rules")
rules_sorted = rules.sort_values(by=['confidence'], ascending=False)
rules_sorted = rules_sorted.reset_index(drop=True)
rules_sorted.index = rules_sorted.index + 1
rules_sorted[['antecedents', 'consequents', 'support', 'confidence']].head(20)
```

Output:

	antecedents	consequents	support	confidence
1	(yogurt)	(whole milk)	0.150590	0.532185
2	(bottled water)	(whole milk)	0.112365	0.525810
3	(sausage)	(whole milk)	0.106978	0.519303
4	(rolls/buns)	(whole milk)	0.178553	0.510638
5	(other vegetables)	(whole milk)	0.191380	0.508174
6	(tropical fruit)	(whole milk)	0.116470	0.498353
7	(root vegetables)	(whole milk)	0.113135	0.490545
8	(soda)	(whole milk)	0.151103	0.481997
9	(yogurt)	(other vegetables)	0.120318	0.425204
10	(rolls/buns)	(other vegetables)	0.146742	0.419663
11	(whole milk)	(other vegetables)	0.191380	0.417693
12	(soda)	(other vegetables)	0.124166	0.396072
13	(yogurt)	(rolls/buns)	0.111339	0.393472
14	(whole milk)	(rolls/buns)	0.178553	0.389698
15	(other vegetables)	(rolls/buns)	0.146742	0.389646
16	(soda)	(rolls/buns)	0.119805	0.382160
17	(rolls/buns)	(soda)	0.119805	0.342627
18	(whole milk)	(soda)	0.151103	0.329787
19	(other vegetables)	(soda)	0.124166	0.329700
20	(whole milk)	(yogurt)	0.150590	0.328667

4.1.4 AClose Algorithm

- The ACLOSE algorithm is a data mining algorithm designed for closed frequent itemset mining. It focuses on finding closed itemsets, which are maximal sets of items in a transaction database where no proper superset has the same frequency (support).
- Closed frequent itemsets, being lossless, are highly beneficial in market basket analysis, as they consolidate similar patterns by identifying unique itemset relationships. This approach captures all frequent itemsets without redundantly including their subsets, ensuring a more concise and efficient representation of item associations.

Code:

```
import pandas as pd
from mlxtend.frequent_patterns import apriori
from mlxtend.preprocessing import TransactionEncoder

def t_x(transactions, itemset):
    """
    This function returns the list of transactions where all items in the itemset
        occur.
    """
    transactions_with_itemset = [transaction for transaction in transactions if
        set(itemset).issubset(set(transaction))]
    return transactions_with_itemset

def i_t(transactions):
    """
    This function returns the maximum common subset of items across all
        transactions.
    """
    transactions_as_sets = [set(transaction) for transaction in transactions]
    common_items = set.intersection(*transactions_as_sets)
    return common_items

def closed_itemset(transactions, itemset):
    t_x_result = t_x(transactions, itemset)
    common_items = i_t(t_x_result)
    # Return if the intersection is equal to the itemset
    return common_items

def aclose(transactions, min_support):
    """
    This function performs the A-Close algorithm to find closed frequent itemsets
        .
    """
    # Convert transactions to a format suitable for apriori
    te = TransactionEncoder()
    te_ary = te.fit(transactions).transform(transactions)
    df = pd.DataFrame(te_ary, columns=te.columns_)

    # Generate frequent itemsets of length 2
    frequent_itemsets_size2 = apriori(df, min_support=min_support, use_colnames=
        True, max_len=2)
```

```
    # Total number of transactions
    total_transactions = len(transactions)

    # Add support count to the frequent itemsets
    frequent_itemsets_size2['support_count'] = frequent_itemsets_size2['support']
        * total_transactions

    # Create minimal generators (itemsets whose support is not the same as any of
        their subsets)
    minimal_generators = []
    for i, row in frequent_itemsets_size2.iterrows():
        itemset = row['itemsets']
        support = row['support']

        # Check if the itemset is a minimal generator
        is_minimal_generator = True
        for j, subset_row in frequent_itemsets_size2.iterrows():
            subset_itemset = subset_row['itemsets']
            subset_support = subset_row['support']

            # Check if the subset has the same support and is a proper subset
            if subset_itemset < itemset and subset_support == support:
                is_minimal_generator = False
                break

        # Add the itemset to minimal generators if it is minimal
        if is_minimal_generator:
            minimal_generators.append((itemset, support, row['support_count']))

    # Convert minimal_generators to DataFrame
    minimal_generators_df = pd.DataFrame(minimal_generators, columns=['itemsets',
        'support', 'support_count'])
    print("Mininmal generators")
    print(minimal_generators_df)
    # Initialize list for closed itemsets
    closed_item_list = []

    # Check each minimal generator for the closed itemset property
    for i, row in minimal_generators_df.iterrows():
        itemset = row['itemsets']
        closed_item_list.append(closed_itemset(transactions, itemset))
        #removing duplicates from the list
        closed_item_list = [list(item) for item in set(tuple(item) for item in
            closed_item_list)]

    return closed_item_list

groceries_data = pd.read_csv("Groceries_dataset.csv")

transactions = (
    groceries_data.groupby('Member_number')['itemDescription']
    .apply(lambda x: list(set(x))) # Get unique items for each member
    .tolist() # Convert the result to a list of lists
)
print(transactions)
min_support = 0.1 # Example minimum support threshold
```

```
closed_itemsets = aclose(transactions, min_support)
#converting closed_itemsets to set notation
closed_itemsets = [set(item) for item in closed_itemsets]
print("Closed Itemsets:", closed_itemsets)
```

Output:

	itemsets	support	support_count
0	(beef)	0.119548	466.0
1	(bottled beer)	0.158799	619.0
2	(bottled water)	0.213699	833.0
3	(brown bread)	0.135967	530.0
4	(butter)	0.126475	493.0
5	(canned beer)	0.165213	644.0
6	(chicken)	0.100564	392.0
7	(citrus fruit)	0.185480	723.0
8	(coffee)	0.114931	448.0
9	(curd)	0.120831	471.0
10	(domestic eggs)	0.133145	519.0
11	(frankfurter)	0.137506	536.0
12	(frozen vegetables)	0.102617	400.0
13	(fruit/vegetable juice)	0.124936	487.0
14	(margarine)	0.116983	456.0
15	(newspapers)	0.139815	545.0
16	(other vegetables)	0.376603	1468.0
17	(pastry)	0.177527	692.0
18	(pip fruit)	0.170600	665.0
19	(pork)	0.132376	516.0
20	(rolls/buns)	0.349666	1363.0
21	(root vegetables)	0.230631	899.0
22	(sausage)	0.206003	803.0
23	(shopping bags)	0.168291	656.0
24	(soda)	0.313494	1222.0
25	(tropical fruit)	0.233710	911.0
26	(whipped/sour cream)	0.154695	603.0
27	(whole milk)	0.458184	1786.0
28	(yogurt)	0.282966	1103.0
29	(whole milk, bottled water)	0.112365	438.0
30	(rolls/buns, other vegetables)	0.146742	572.0
31	(other vegetables, soda)	0.124166	484.0
32	(other vegetables, whole milk)	0.191380	746.0
33	(yogurt, other vegetables)	0.120318	469.0
34	(rolls/buns, soda)	0.119805	467.0
35	(rolls/buns, whole milk)	0.178553	696.0
36	(rolls/buns, yogurt)	0.111339	434.0

```
37   (whole milk, root vegetables)  0.113135          441.0
38           (whole milk, sausage)  0.106978          417.0
39              (whole milk, soda)  0.151103          589.0
40    (tropical fruit, whole milk)  0.116470          454.0
41            (yogurt, whole milk)  0.150590          587.0
{'rolls/buns', 'other vegetables'}
{'whole milk', 'soda'}
{'tropical fruit', 'whole milk'}
{'rolls/buns', 'soda'}
{'whole milk', 'sausage'}
{'margarine'}
{'yogurt', 'whole milk'}
{'tropical fruit'}
{'canned beer'}
{'brown bread'}
{'yogurt'}
{'rolls/buns'}
{'whole milk', 'bottled water'}
{'domestic eggs'}
{'whipped/sour cream'}
{'whole milk'}
{'butter'}
{'beef'}
{'pip fruit'}
{'whole milk', 'root vegetables'}
{'pork'}
{'other vegetables', 'soda'}
{'yogurt', 'other vegetables'}
{'fruit/vegetable juice'}
{'pastry'}
{'rolls/buns', 'yogurt'}
{'other vegetables'}
{'soda'}
{'root vegetables'}
{'chicken'}
{'frankfurter'}
{'sausage'}
{'rolls/buns', 'whole milk'}
{'frozen vegetables'}
{'citrus fruit'}
{'bottled beer'}
{'bottled water'}
{'newspapers'}
{'shopping bags'}
{'curd'}
{'coffee'}
{'other vegetables', 'whole milk'}
```

4.1.5 Pincer Search Algorithm

- The Pincer-Search algorithm is an advanced data mining algorithm designed for frequent itemset mining. It is particularly efficient for finding maximal frequent itemsets (MFIs), which are the largest itemsets that satisfy the minimum support threshold, meaning none of their supersets are frequent.
- Maximal Frequent Itemsets eliminate the need to report smaller subsets, making the results more compact. In Market Basket Analysis, MFIs helps in identifying the largest groups of items frequently purchased together.

Code:

```
from itertools import combinations
from collections import defaultdict

class PincerSearchOptimized:
    def __init__(self, dataset, min_support):
        self.dataset = dataset
        self.min_support = min_support
        self.transactions_count = len(dataset)
        self.itemsets_support = defaultdict(int) # Store support counts for
            faster lookup

    def calculate_support(self, itemset):
        """Calculate the support of an itemset"""
        # Cache the support values to avoid redundant calculations
        if frozenset(itemset) in self.itemsets_support:
            return self.itemsets_support[frozenset(itemset)] / self.
                transactions_count

        count = sum(1 for transaction in self.dataset if itemset.issubset(
            transaction))
        self.itemsets_support[frozenset(itemset)] = count
        return count / self.transactions_count

    def generate_candidates(self, size, prev_frequent_itemsets):
        """Generate candidate itemsets of a given size"""
        candidates = set()
        prev_frequent_itemsets = list(prev_frequent_itemsets) # Convert to list
            for indexing
        for i, itemset1 in enumerate(prev_frequent_itemsets):
            for itemset2 in prev_frequent_itemsets[i+1:]:
                # Join itemsets only if they share (size-2) elements
                candidate = itemset1 | itemset2
                if len(candidate) == size:
                    candidates.add(candidate)
        return candidates

    def mfs_prune(self, Lk, MFS):
        """Prune Lk based on the current MFS"""
        return {itemset for itemset in Lk if not any(itemset.issubset(m) for m in
             MFS)}

    def mfcs_prune(self, Ck, MFCS):
        """Prune Ck+1 based on the current MFCS"""
        return {itemset for itemset in Ck if any(itemset.issubset(m) for m in
            MFCS)}

    def mfcs_gen(self, Sk, MFCS):
        """Generate new candidates for MFCS"""
        new_mfcs = set(MFCS)
        for s in Sk:
            for m in MFCS:
                if s.issubset(m):
                    new_mfcs.remove(m) # Remove m if s is a subset
            for e in s:
                new_mfcs.add(m - {e} for m in MFCS if m - {e} not in MFCS)
        return new_mfcs
```

```
    def recovery(self, Lk, MFS):
        """Recover candidates for the next iteration (Ck+1)"""
        Ck_plus_1 = set()
        for l in Lk:
            for m in MFS:
                if all(item in m for item in list(l)[:len(l)-1]):
                    for item in m:
                        Ck_plus_1.add(frozenset(l | {item}))
        return Ck_plus_1

    def pincer_search(self):
        """Main Pincer-Search algorithm"""
        k = 1
        MFS = set()
        MFCS = {frozenset([item]) for transaction in self.dataset for item in
            transaction}

        # Initially populate MFS with frequent 1-itemsets
        frequent_1_itemsets = {itemset for itemset in MFCS if self.
            calculate_support(itemset) >= self.min_support}
        MFS.update(frequent_1_itemsets)

        max_frequent_size = 1 # Start by tracking 1-itemsets, then increase as
            necessary
        prev_frequent_itemsets = frequent_1_itemsets

        while True:
            Ck = self.generate_candidates(k + 1, prev_frequent_itemsets) #
                Generate candidate itemsets of size k+1
            frequent_itemsets = {itemset for itemset in Ck if self.
                calculate_support(itemset) >= self.min_support}

            MFS.update(frequent_itemsets)

            if frequent_itemsets:
                max_frequent_size = max(max_frequent_size, k + 1) # Track the
                    largest size found

            if not Ck or not frequent_itemsets:
                break

            # Prune Lk and MFCS
            Lk = self.mfs_prune(Ck, MFS) # Prune Lk based on MFS
            MFCS = self.mfcs_prune(Ck, MFCS) # Prune Ck+1 based on MFCS

            # Generate new candidates using the recovery procedure
            Ck_plus_1 = self.recovery(Lk, MFS)

            # Update prev_frequent_itemsets for the next iteration
            prev_frequent_itemsets = frequent_itemsets

            # Update k for the next iteration
            k += 1

        # Filter MFS to return only itemsets of the maximum size (greater than 1)
        max_frequent_itemsets = {itemset for itemset in MFS if len(itemset) ==
            max_frequent_size and len(itemset) > 1}
```

```
        return max_frequent_itemsets


def load_dataset(file_path):
    """Load the dataset from a file"""
    dataset = []
    with open(file_path, 'r') as file:
        for line in file:
            transaction = set(map(int, line.strip().split()))
            dataset.append(transaction)
    return dataset

groceries_data = pd.read_csv("Groceries_dataset.csv")

transactions = (
    groceries_data.groupby('Member_number')['itemDescription']
    .apply(lambda x: list(set(x))) # Get unique items for each member
    .tolist() # Convert the result to a list of lists
)
min_support = 0.1

# Initialize and run the Pincer-Search
pincer_search = PincerSearchOptimized(transactions, min_support)
max_frequent_itemset = pincer_search.pincer_search()
print("MAX␣FREQUENT␣ITEMSETS:")
for i in max_frequent_itemset:
    print(i)
```

Output:

```
MAX FREQUENT ITEMSETS:
frozenset({'rolls/buns', 'soda'})
frozenset({'other vegetables', 'soda'})
frozenset({'other vegetables', 'yogurt'})
frozenset({'tropical fruit', 'whole milk'})
frozenset({'rolls/buns', 'yogurt'})
frozenset({'rolls/buns', 'other vegetables'})
frozenset({'whole milk', 'soda'})
frozenset({'whole milk', 'bottled water'})
frozenset({'whole milk', 'root vegetables'})
frozenset({'whole milk', 'other vegetables'})
frozenset({'whole milk', 'rolls/buns'})
frozenset({'whole milk', 'yogurt'})
frozenset({'sausage', 'whole milk'})
```

4.1.6 Charm Algorithm

- CHARM (Closed Itemset Mining) is an efficient algorithm used to discover closed frequent itemsets from transaction data. A closed itemset is an itemset that has the same support as any of its supersets, meaning no superset of the itemset has a higher frequency.
- The CHARM algorithm is advantageous because it avoids generating redundant itemsets, thus significantly improving computational efficiency. It directly

identifies closed itemsets without needing to compare each frequent itemset to its supersets, reducing the size of the solution space.

– This makes CHARM particularly useful in applications like market basket analysis, where uncovering meaningful patterns (e.g., items frequently bought together) is crucial.

Code:

```
import pandas as pd
from mlxtend.frequent_patterns import apriori
from mlxtend.preprocessing import TransactionEncoder

# Load the dataset from CSV
df = pd.read_csv('grocery_fp.csv')

# Step 1: Process the dataset to create a list of transactions
# We assume each row represents an item purchased by a customer.
# Each column represents a product, and each cell is either 0 (not purchased) or
    1 (purchased).

# Drop the first column if it's not related to the items (for example, '
    Member_number' column)
df = df.drop(columns=["Member_number"]) # Modify if you have other irrelevant
    columns

# Convert the data to a list of transactions where each transaction is a list of
    items purchased
transactions = []
for index, row in df.iterrows():
    transaction = [df.columns[i] for i in range(len(row)) if row[i] == 1]
    transactions.append(transaction)

# Step 2: Transaction Encoder
encoder = TransactionEncoder()
encoded_data = encoder.fit_transform(transactions)
encoded_df = pd.DataFrame(encoded_data, columns=encoder.columns_)

# Step 3: Apply Apriori to find frequent itemsets
# Adjust the min_support as per your needs
frequent_itemsets = apriori(encoded_df, min_support=0.01, use_colnames=True)

# Step 4: Filter closed itemsets
# A closed itemset is one that is not included in any larger frequent itemset
def is_closed(itemset, frequent_itemsets):
    itemset_set = set(itemset)
    for other_itemset in frequent_itemsets['itemsets']:
        if itemset_set.issubset(other_itemset) and itemset_set != other_itemset:
            return False
    return True

# Filter the frequent itemsets to keep only closed ones
closed_itemsets = frequent_itemsets[frequent_itemsets['itemsets'].apply(lambda x
    : is_closed(x, frequent_itemsets))]

# Display the closed frequent itemsets
print("Closed␣Frequent␣Itemsets:")
print(closed_itemsets)
```

Output:

```
Closed Frequent Itemsets:
       support                                              itemsets
0     0.015393                          (Instant food products)
12    0.016932                                       (candles)
18    0.010775                                       (cereals)
22    0.015393                          (chocolate marshmallow)
24    0.018728                                (cling film/bags)
...        ...                                               ...
3011  0.011031     (whipped/sour cream, yogurt, whole milk, soda)
3012  0.010518  (other vegetables, yogurt, rolls/buns, bottled...
3013  0.013597  (other vegetables, sausage, yogurt, rolls/buns...
3014  0.010005  (other vegetables, shopping bags, yogurt, roll...
3015  0.013597  (other vegetables, yogurt, rolls/buns, whole m...
```

4.2 Classification models

- **Data Labeling and Training:** Classification models rely on labeled training data to learn how to predict the class or category of new data. Accurate labeling ensures that the model can correctly recognize patterns, such as predicting whether a customer will churn based on their past interactions.
- **Improved Decision-Making:** By predicting future outcomes based on historical data, classification models empower organizations to make informed decisions. For example, businesses can use classification to target high-value customers, predict product demand, or prevent fraud.
- **Scalability and Computational Efficiency:** Many classification algorithms are designed to handle large datasets, making them scalable for big data applications. Techniques such as ensemble learning (e.g., Random Forest) and optimized algorithms ensure that even with large volumes of data, the model remains efficient and accurate.
- **Evaluation Metrics:** Classification models are evaluated using various metrics, such as accuracy, precision, recall, and F1-score, which help assess the model's performance. These metrics are particularly useful in scenarios like imbalanced data, where accuracy alone might not be sufficient to gauge effectiveness.

4.2.1 Decision Tree Classification

- A supervised machine learning approach called Decision Tree Classification divides data into subsets according to the most important feature values in order to categorize the data. Each leaf node represents a class label, and each internal node indicates a decision based on a feature, forming a structure like a tree. To divide the data at each node, the tree is built using criteria like entropy or Gini impurity.
- In Market Basket Analysis, we use attributes like transaction ID and cost of the item to predict in which of the classes does a customer belong to (such as Low spender and high spender).

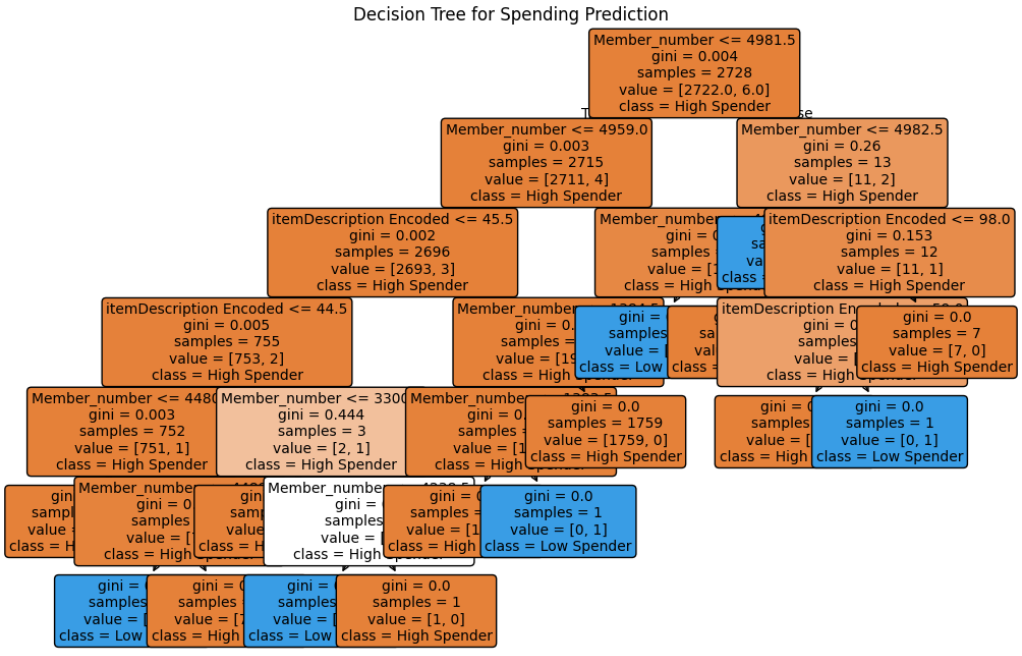
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, export_text
from sklearn.naive_bayes import MultinomialNB
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.metrics import classification_report, accuracy_score
# Load dataset
data = pd.read_excel('Groceries_dataset_with_costs.xlsx')

# View initial data
print(data.head())
# Derive Day of Week from 'Date'
data['Date'] = pd.to_datetime(data['Date'])
data['Day␣of␣Week'] = data['Date'].dt.day_name()
```

```
# Aggregate total spending per customer
customer_spending = data.groupby('Member_number')['Cost'].sum().reset_index()
customer_spending['Spender Category'] = np.where(customer_spending['Cost'] >
    200, 'High Spender', 'Low Spender')

# Merge back to original data
data = data.merge(customer_spending[['Member_number', 'Spender Category']], on='
    Member_number', how='left')


# Encode categorical features
le_item = LabelEncoder()
data['itemDescription Encoded'] = le_item.fit_transform(data['itemDescription'])



le_day = LabelEncoder()
data['Day of Week Encoded'] = le_day.fit_transform(data['Day of Week'])



# Create feature sets
features_1 = data.groupby('Member_number')['itemDescription Encoded'].first().
    reset_index() # For spending prediction
features_2 = data[['itemDescription Encoded', 'Day of Week Encoded']].
    drop_duplicates() # For day prediction



# Targets
target_1 = data.drop_duplicates('Member_number')['Spender Category'] # High/Low
    spender
target_2 = data.drop_duplicates(['itemDescription', 'Day of Week'])['Day of Week
    Encoded'] # Day of week



# Split data
X_train1, X_test1, y_train1, y_test1 = train_test_split(features_1, target_1,
    test_size=0.3, random_state=42)
X_train2, X_test2, y_train2, y_test2 = train_test_split(features_2, target_2,
    test_size=0.3, random_state=42)


from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.tree import plot_tree

# Initialize the Decision Tree Classifier
clf_1 = DecisionTreeClassifier(random_state=42)

# Train the model
clf_1.fit(X_train1, y_train1)

# Make predictions
y_pred = clf_1.predict(X_test1)
```

```
# Evaluate the model
accuracy = accuracy_score(y_test1, y_pred)
print(f"Accuracy: {accuracy*100:.2f}%")

# Print the classification report
print("\nClassification Report:")
print(classification_report(y_test1, y_pred))

# Visualize the decision tree
plt.figure(figsize=(12, 8))
plot_tree(clf_1, filled=True, feature_names=features_1.columns, class_names=
    target_1.unique(), rounded=True, fontsize=10)
plt.title("Decision Tree for Spending Prediction")
plt.show()
```

Output:

```
Accuracy: 99.57%

Classification Report:
              precision    recall  f1-score   support

 High Spender       1.00      1.00      1.00      1166
  Low Spender       0.00      0.00      0.00         4

     accuracy                           1.00      1170
    macro avg       0.50      0.50      0.50      1170
 weighted avg       0.99      1.00      0.99      1170
```



4.2.2 Naive Bayes Classification

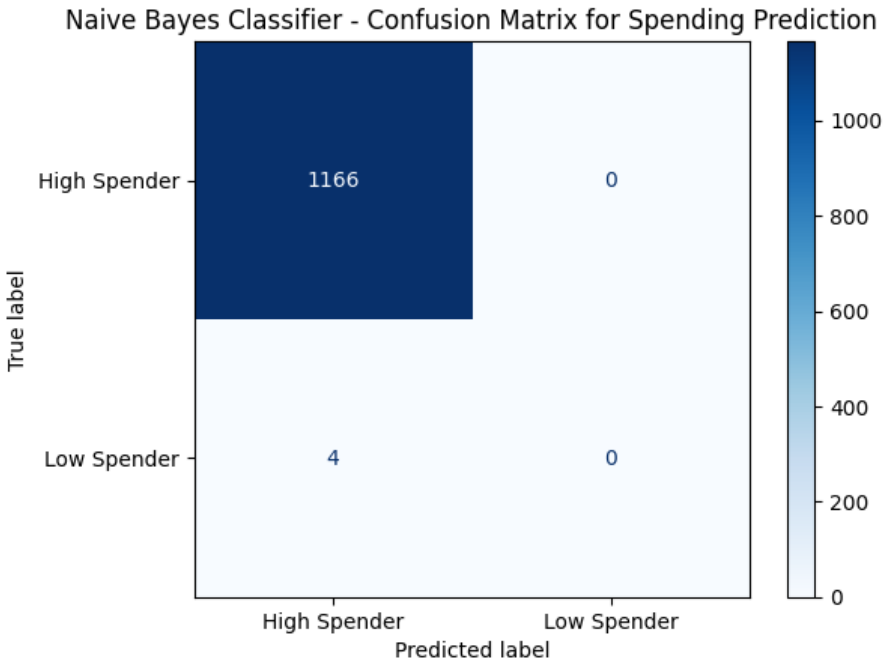
- Based on Bayes' Theorem, Naive Bayes Classification is a probabilistic supervised machine learning technique. The computation of class probability is made simpler by the assumption that the features used to predict the class label are independent. It is especially useful for handling categorical features and big datasets. Each data point is assigned to the class with the highest probability after the algorithm calculates the posterior probability for each class.
- Using variables like location, time, and other contextual information, Naive Bayes can be used to predict the sorts of crimes in crime datasets. Naive Bayes assists in recognizing probable crime trends and guiding focused measures for crime prevention by categorizing criminal events according to the likelihood of various outcomes.
- Naive Bayes is robust to missing data. Since it computes probabilities based on individual features independently, missing values in some features can be easily handled by ignoring them during probability calculations, making the algorithm useful for datasets with incomplete information.

```
# Create feature sets
features_1 = data.groupby('Member_number')['itemDescription␣Encoded'].first().
    reset_index() # For spending prediction
features_2 = data[['itemDescription␣Encoded', 'Day␣of␣Week␣Encoded']].
    drop_duplicates() # For day prediction



# Targets
target_1 = data.drop_duplicates('Member_number')['Spender␣Category'] # High/Low
    spender
target_2 = data.drop_duplicates(['itemDescription', 'Day␣of␣Week'])['Day␣of␣Week
    ␣Encoded'] # Day of week



# Split data
X_train1, X_test1, y_train1, y_test1 = train_test_split(features_1, target_1,
    test_size=0.3, random_state=42)
X_train2, X_test2, y_train2, y_test2 = train_test_split(features_2, target_2,
    test_size=0.3, random_state=42)

from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Initialize the Naive Bayes Classifier
nb_clf_1 = GaussianNB()

# Train the model for spending prediction (target_1)
nb_clf_1.fit(X_train1, y_train1)
```

```
# Make predictions
y_pred_nb = nb_clf_1.predict(X_test1)

# Evaluate the model
accuracy_nb = accuracy_score(y_test1, y_pred_nb)
print(f"Accuracy: {accuracy_nb*100:.2f}%")

# Print the classification report
print("\nClassification Report:")
print(classification_report(y_test1, y_pred_nb))

# Display confusion matrix using confusion_matrix function
cm = confusion_matrix(y_test1, y_pred_nb) # Calculate confusion matrix first
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=nb_clf_1.
    classes_)
disp.plot(cmap=plt.cm.Blues)
plt.title("Naive Bayes Classifier - Confusion Matrix for Spending Prediction")
plt.show()
```

Output:

```
Accuracy: 99.66%

Classification Report:
              precision    recall  f1-score   support

 High Spender       1.00      1.00      1.00      1166
  Low Spender       0.00      0.00      0.00         4

    accuracy                           1.00      1170
   macro avg       0.50      0.50      0.50      1170
weighted avg       0.99      1.00      0.99      1170
```



4.2.3 Multinomial Naive Bayes Classification

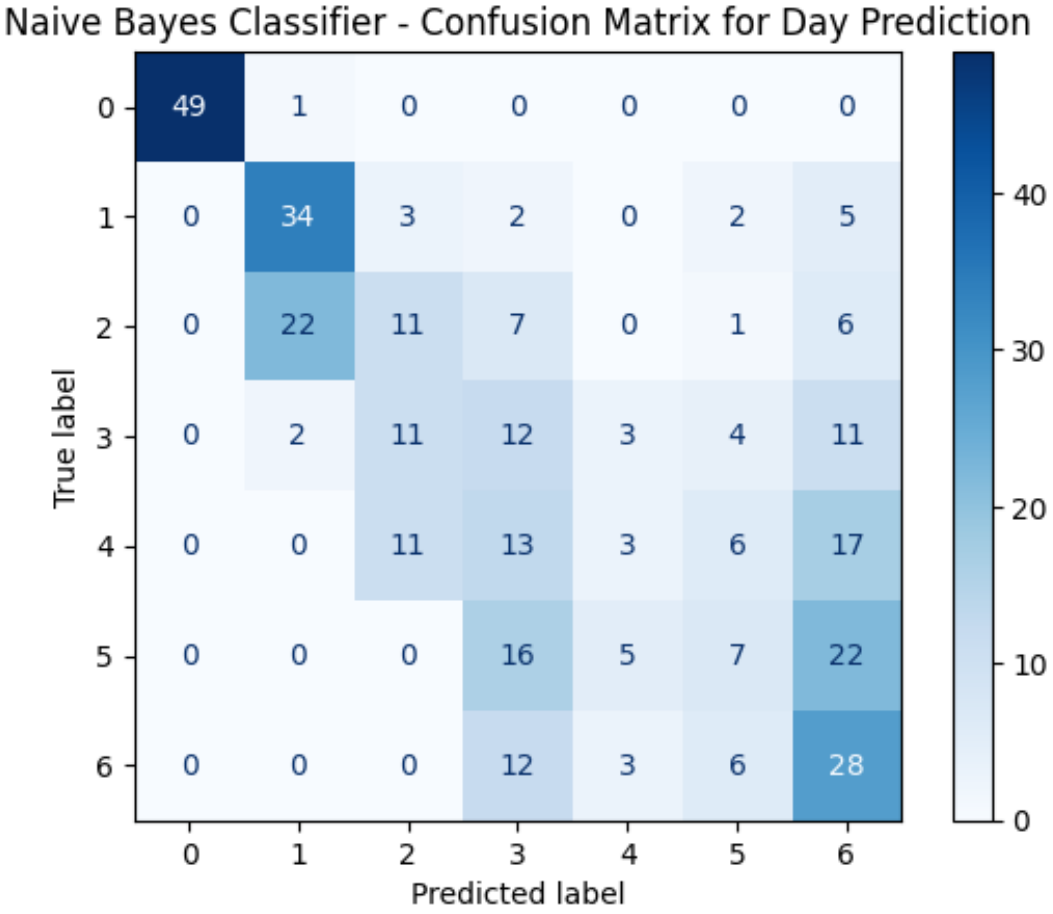
- **Works Well for Text Classification:** Multinomial Naive Bayes is particularly effective for text classification tasks such as spam detection and sentiment analysis. It assumes that the features (words) are multinomially distributed and calculates the probability of a class given the frequency of each word in the document, making it ideal for analyzing word counts or frequency-based features.
- **Handles Discrete Features:** Unlike Gaussian Naive Bayes, which is used for continuous features, Multinomial Naive Bayes is designed for discrete data, such as word counts or event frequencies. This makes it suitable for problems where the features represent counts, such as in document classification or event prediction.
- **Assumes Feature Independence:** Like other Naive Bayes variants, the Multinomial Naive Bayes classifier assumes that features are conditionally independent given the class label. This simplifies the computation of the posterior probabilities and makes the model computationally efficient, especially in large datasets.

```
# Create feature sets
features_1 = data.groupby('Member_number')['itemDescription␣Encoded'].first().
    reset_index() # For spending prediction
features_2 = data[['itemDescription␣Encoded', 'Day␣of␣Week␣Encoded']].
    drop_duplicates() # For day prediction



# Targets
target_1 = data.drop_duplicates('Member_number')['Spender␣Category'] # High/Low
    spender
```

```
target_2 = data.drop_duplicates(['itemDescription', 'Day of Week'])['Day of Week
    Encoded'] # Day of week



# Split data
X_train1, X_test1, y_train1, y_test1 = train_test_split(features_1, target_1,
    test_size=0.3, random_state=42)
X_train2, X_test2, y_train2, y_test2 = train_test_split(features_2, target_2,
    test_size=0.3, random_state=42)

from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split

# Assuming you have the features_2 and target_2 already created from your
    dataset

# Initialize the Naive Bayes Classifier
nb_clf_2 = MultinomialNB()

# Train the model for day prediction (target_2)
nb_clf_2.fit(X_train2, y_train2)

# Make predictions
y_pred_nb_2 = nb_clf_2.predict(X_test2)

# Evaluate the model
accuracy_nb_2 = accuracy_score(y_test2, y_pred_nb_2)
print(f"Accuracy: {accuracy_nb_2*100:.2f}%")

# Print the classification report
print("\nClassification Report:")
print(classification_report(y_test2, y_pred_nb_2))

# Display confusion matrix using confusion_matrix function
cm_2 = confusion_matrix(y_test2, y_pred_nb_2) # Calculate confusion matrix first
disp_2 = ConfusionMatrixDisplay(confusion_matrix=cm_2, display_labels=nb_clf_2.
    classes_)
disp_2.plot(cmap=plt.cm.Blues)
plt.title("Naive Bayes Classifier - Confusion Matrix for Day Prediction")
plt.show()
```

Output:

```
Accuracy: 42.99%

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99        50
           1       0.58      0.74      0.65        46
           2       0.31      0.23      0.27        47
           3       0.19      0.28      0.23        43
           4       0.21      0.06      0.09        50
           5       0.27      0.14      0.18        50
           6       0.31      0.57      0.41        49

    accuracy                           0.43       335
   macro avg       0.41      0.43      0.40       335
weighted avg       0.41      0.43      0.40       335
```



4.2.4 Support Vector Classification

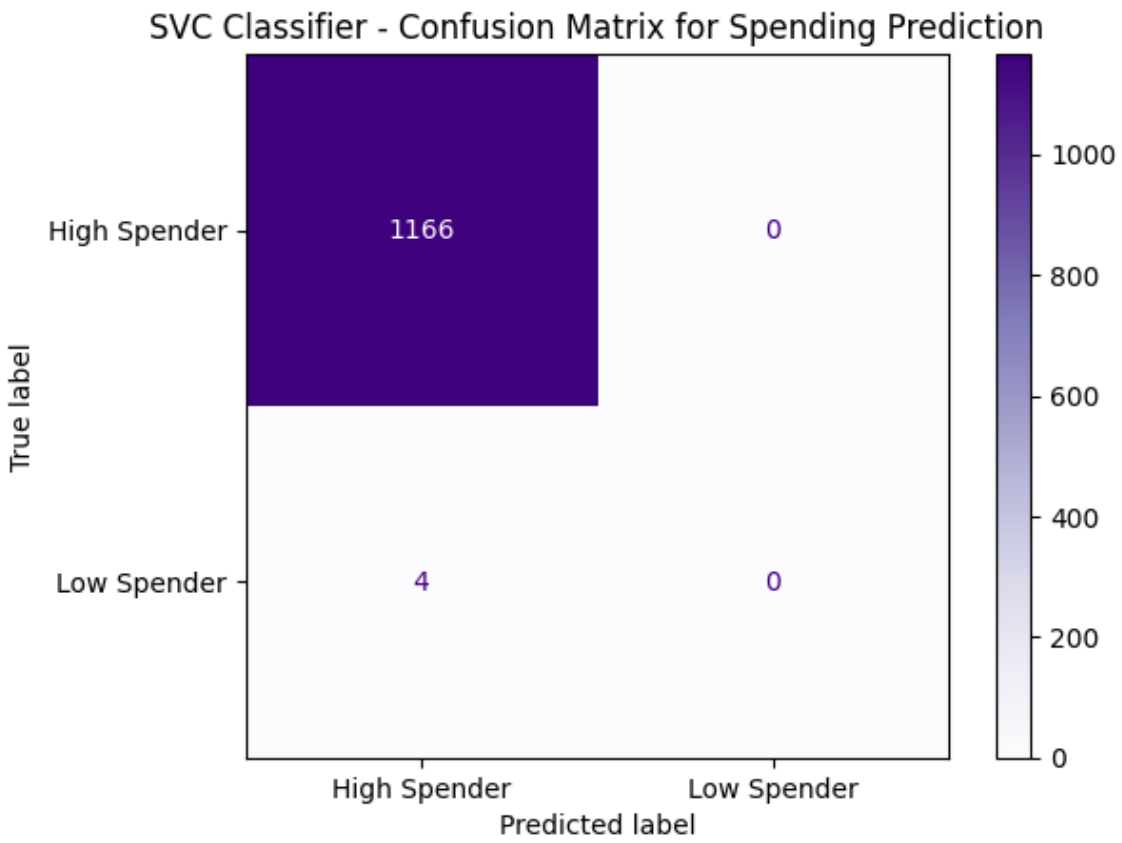
- **Effective in High-Dimensional Spaces:** Support Vector Classification (SVC) is particularly powerful in high-dimensional spaces, making it effective for tasks such as text classification and image recognition. It performs well even when the number of features exceeds the number of data points, which is a common scenario in real-world datasets.
- **Works Well with Non-linear Boundaries:** SVC uses the kernel trick to map data into higher dimensions where a linear separator can be used. This makes it highly effective for classification problems where the data is not linearly separable in the original input space, as it can find non-linear decision boundaries.
- **Robust to Overfitting:** SVC is less prone to overfitting, especially in high-dimensional spaces, due to the use of the regularization parameter (C). The parameter helps to control the margin size and balance between maximizing the margin and minimizing classification error, making SVC robust to noisy data.

```
# Create feature sets
features_1 = data.groupby('Member_number')['itemDescription␣Encoded'].first().
    reset_index() # For spending prediction
features_2 = data[['itemDescription␣Encoded', 'Day␣of␣Week␣Encoded']].
    drop_duplicates() # For day prediction



# Targets
target_1 = data.drop_duplicates('Member_number')['Spender␣Category'] # High/Low
    spender
target_2 = data.drop_duplicates(['itemDescription', 'Day␣of␣Week'])['Day␣of␣Week
    ␣Encoded'] # Day of week



# Split data
X_train1, X_test1, y_train1, y_test1 = train_test_split(features_1, target_1,
    test_size=0.3, random_state=42)
X_train2, X_test2, y_train2, y_test2 = train_test_split(features_2, target_2,
    test_size=0.3, random_state=42)

from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Initialize the Support Vector Classifier
svc_clf_1 = SVC(random_state=42)

# Train the model for spending prediction (target_1)
svc_clf_1.fit(X_train1, y_train1)

# Make predictions
y_pred_svc = svc_clf_1.predict(X_test1)
```

```
# Evaluate the model
accuracy_svc = accuracy_score(y_test1, y_pred_svc)
print(f"Accuracy: {accuracy_svc*100:.2f}%")

# Print the classification report
print("\nClassification Report:")
print(classification_report(y_test1, y_pred_svc))

# Display confusion matrix using confusion_matrix function
cm_svc = confusion_matrix(y_test1, y_pred_svc) # Calculate confusion matrix
    first
disp_svc = ConfusionMatrixDisplay(confusion_matrix=cm_svc, display_labels=
    svc_clf_1.classes_)
disp_svc.plot(cmap=plt.cm.Purples, values_format='d', colorbar=True)
plt.title("SVC Classifier - Confusion Matrix for Spending Prediction")
plt.show()
```

Output:

```
Accuracy: 99.66%

Classification Report:
              precision    recall  f1-score   support

 High Spender       1.00      1.00      1.00      1166
  Low Spender       0.00      0.00      0.00         4

     accuracy                           1.00      1170
    macro avg       0.50      0.50      0.50      1170
weighted avg       0.99      1.00      0.99      1170
```



Summary: Spending prediction as high or low spender the best model is **Naive bayesian SVC** (it's obtained by keeping all **best hyper parameters** for the respective models)

4.3 Clustering

Clustering is an unsupervised machine learning technique used to group data points into clusters based on their similarity. Unlike classification, clustering does not rely on labeled data but instead discovers patterns or structures in the dataset.

4.3.1 K-Means

K-Means is one of the most popular clustering algorithms. It works by partitioning the dataset into k clusters, where k is a user-defined parameter. The algorithm minimizes the variance within each cluster by iteratively updating cluster centroids and assigning data points to the nearest centroid. K-Means is effective for partitioning data when the clusters are spherical and well-separated.

4.3.1.1 Elbow Method for Optimal Number of Clusters

The following code demonstrates the elbow method to determine the optimal number of clusters:

Listing 1: Elbow Method for Optimal Clusters

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = pd.read_excel('Groceries_dataset_with_costs.xlsx')

# Convert Date to a datetime object
data['Date'] = pd.to_datetime(data['Date'])

scaler = StandardScaler()
memberid_scaled = scaler.fit_transform(data[['Member_number']])

# Elbow Method
wcss = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=300, n_init=10,
        random_state=42)
    kmeans.fit(memberid_scaled)
    wcss.append(kmeans.inertia_)
```

```
plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), wcss, marker='o')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS (Within-Cluster Sum of Squares)')
plt.show()
```

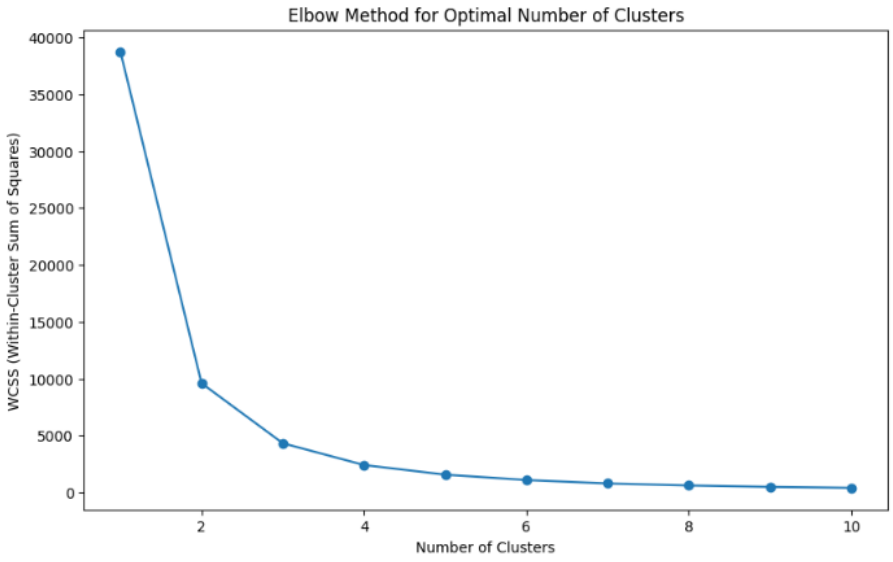


Figure 1: Elbow Method for Optimal Clusters

4.3.1.2 K-Means Clustering

Listing 2: K-Means Clustering

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
df = pd.read_excel('Groceries_dataset_with_costs.xlsx')

# Convert Date to a datetime object
df['Date'] = pd.to_datetime(df['Date'])

# Feature Engineering
customer_features = df.groupby('Member_number').agg(
    Total_Spending=('Cost', 'sum'),
    Total_Purchases=('itemDescription', 'count'),
    Unique_Items=('itemDescription', 'nunique'),
    Avg_Spending=('Cost', 'mean')
).reset_index()

scaler = StandardScaler()
scaled_features = scaler.fit_transform(customer_features[['Total_Spending', '
    Total_Purchases', 'Unique_Items', 'Avg_Spending']])
```

```
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(scaled_features)

plt.figure(figsize=(8, 6))
sns.scatterplot(
    x=scaled_features[:, 0],
    y=scaled_features[:, 1],
    hue=kmeans_labels,
    palette='viridis',
    s=50
)
plt.title('K-Means Clustering')
plt.xlabel('Total Spending (scaled)')
plt.ylabel('Total Purchases (scaled)')
plt.show()
```

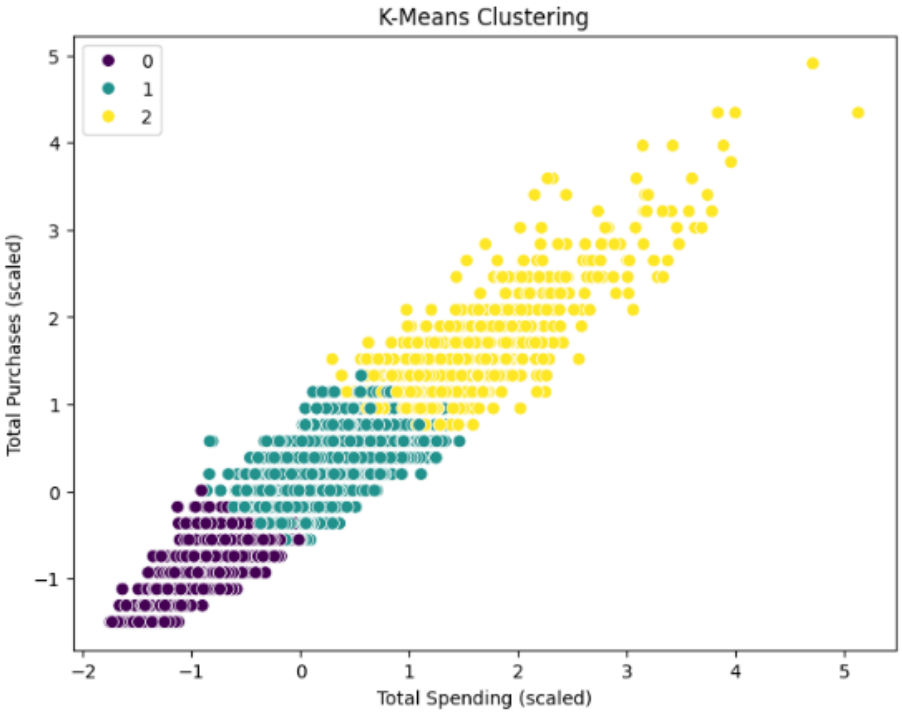


Figure 2: K-Means Clustering Results

4.3.2 DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm. It groups together closely packed points and marks points that are in low-density regions as outliers. Unlike K-Means, DBSCAN does not require specifying the number of clusters in advance and can handle clusters of arbitrary shape. The key parameters in DBSCAN are **eps** (the

maximum distance between two points to be considered neighbors) and **min_samples** (the minimum number of points required to form a dense region).

Listing 3: DBSCAN Clustering

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
df = pd.read_excel('Groceries_dataset_with_costs.xlsx')

# Convert Date to a datetime object
df['Date'] = pd.to_datetime(df['Date'])

customer_features = df.groupby('Member_number').agg(
    Total_Spending=('Cost', 'sum'),
    Total_Purchases=('itemDescription', 'count'),
    Unique_Items=('itemDescription', 'nunique'),
    Avg_Spending=('Cost', 'mean')
).reset_index()

scaler = StandardScaler()
scaled_features = scaler.fit_transform(customer_features[['Total_Spending', '
    Total_Purchases', 'Unique_Items', 'Avg_Spending']])

dbscan = DBSCAN(eps=1, min_samples=150)
dbscan_labels = dbscan.fit_predict(scaled_features)

plt.figure(figsize=(8, 6))
sns.scatterplot(
    x=scaled_features[:, 0],
    y=scaled_features[:, 1],
    hue=dbscan_labels,
    palette='Set2',
    s=50
)
plt.title('DBSCAN␣Clustering')
plt.xlabel('Total␣Spending␣(scaled)')
plt.ylabel('Total␣Purchases␣(scaled)')
plt.show()
```

Figure 3: DBSCAN Clustering Results

4.3.3 HDBSCAN

HDBSCAN (Hierarchical DBSCAN) is an extension of DBSCAN that applies hierarchical clustering to improve upon DBSCAN's ability to find clusters with varying densities. It does not require setting the `eps` parameter, as it automatically finds the optimal distance between points. HDBSCAN is often more robust when dealing with datasets that contain clusters of varying shapes and sizes.

Listing 4: HDBSCAN Clustering

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
import hdbscan
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_excel('Groceries_dataset_with_costs.xlsx')
df['Date'] = pd.to_datetime(df['Date'])

customer_features = df.groupby('Member_number').agg(
    Total_Spending=('Cost', 'sum'),
    Total_Purchases=('itemDescription', 'count'),
```

```
    Unique_Items=('itemDescription', 'nunique'),
    Avg_Spending=('Cost', 'mean')
).reset_index()

scaler = StandardScaler()
scaled_features = scaler.fit_transform(customer_features[['Total_Spending', '
    Total_Purchases', 'Unique_Items', 'Avg_Spending']])

hdbscan_model = hdbscan.HDBSCAN(min_samples=150, cluster_selection_method='eom')
hdbscan_labels = hdbscan_model.fit_predict(scaled_features)

plt.figure(figsize=(8, 6))
sns.scatterplot(
    x=scaled_features[:, 0],
    y=scaled_features[:, 1],
    hue=hdbscan_labels,
    palette='Set2',
    s=50
)
plt.title('HDBSCAN Clustering')
plt.xlabel('Total Spending (scaled)')
plt.ylabel('Total Purchases (scaled)')
plt.show()
```

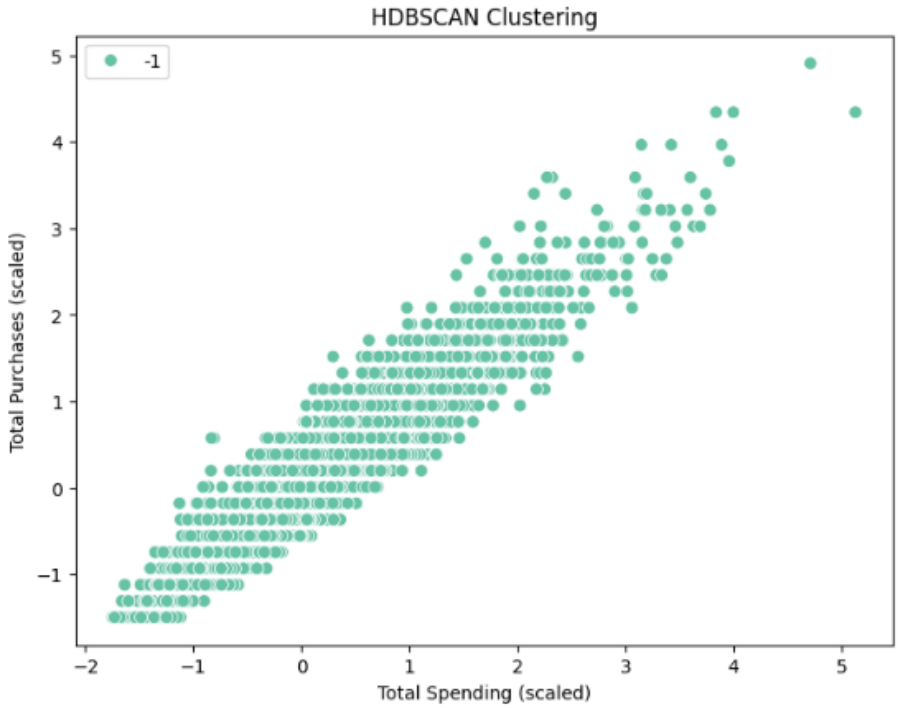


Figure 4: HDBSCAN Clustering Results

4.3.4 Agglomerative Clustering (AGNES)

Agglomerative Clustering (AGNES) is a hierarchical clustering algorithm that starts by treating each data point as its own cluster and then iteratively merges the closest pairs

of clusters based on a distance metric, typically Euclidean distance. The key parameter is the `n_clusters`, which determines the number of clusters the algorithm should create. AGNES can be visualized using both a scatter plot of the clustered data and a dendrogram.

4.3.4.1 AGNES Clustering Scatter Plot

Listing 5: AGNES Clustering Scatter Plot

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
df = pd.read_excel('Groceries_dataset_with_costs.xlsx')

# Convert Date to a datetime object
df['Date'] = pd.to_datetime(df['Date'])

customer_features = df.groupby('Member_number').agg(
    Total_Spending=('Cost', 'sum'),
    Total_Purchases=('itemDescription', 'count'),
    Unique_Items=('itemDescription', 'nunique'),
    Avg_Spending=('Cost', 'mean')
).reset_index()

scaler = StandardScaler()
scaled_features = scaler.fit_transform(customer_features[['Total_Spending', '
    Total_Purchases', 'Unique_Items', 'Avg_Spending']])

agg_clustering = AgglomerativeClustering(n_clusters=3, affinity='euclidean',
    linkage='ward')
agg_labels = agg_clustering.fit_predict(scaled_features)

plt.figure(figsize=(8, 6))
sns.scatterplot(
    x=scaled_features[:, 0],
    y=scaled_features[:, 1],
    hue=agg_labels,
    palette='Set1',
    s=50
)
plt.title('Agglomerative␣Clustering␣(AGNES)')
plt.xlabel('Total␣Spending␣(scaled)')
plt.ylabel('Total␣Purchases␣(scaled)')
plt.show()
```

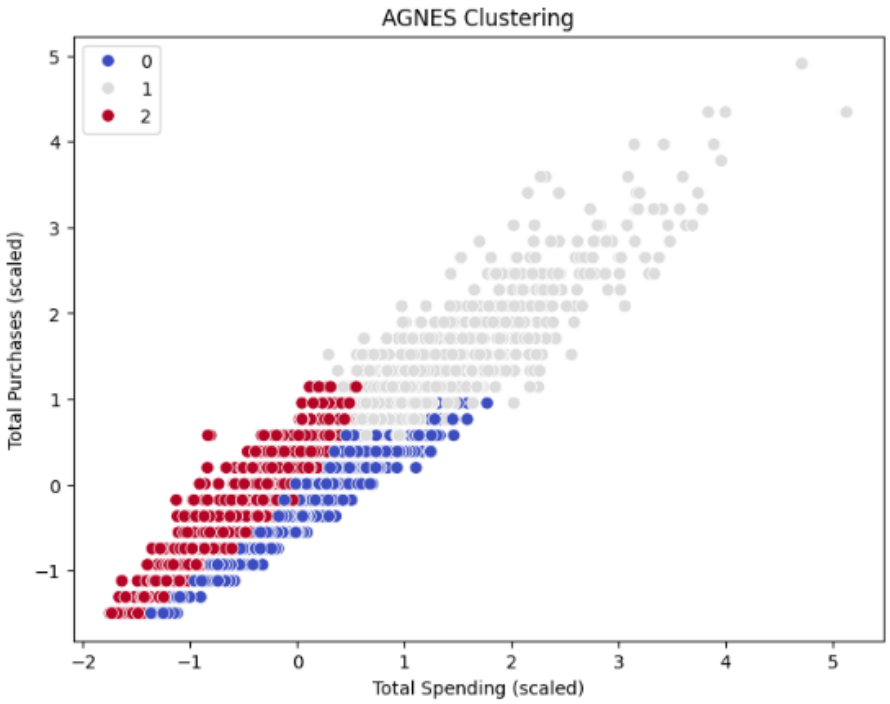


Figure 5: Agglomerative Clustering Results (AGNES)

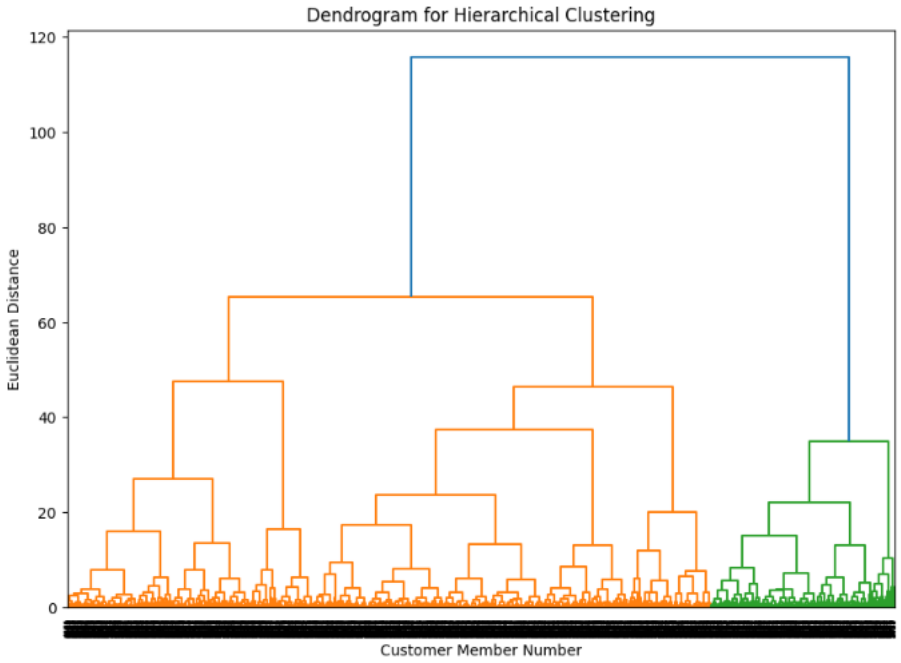


Figure 6: Dendogram for AGNES

5 PySpark

- PySpark helps process massive datasets at scale, making it easier for businesses and organizations to analyze large volumes of data and extract meaningful insights.
- It uses distributed computing across multiple machines to handle big data, ensuring faster computations and the ability to process data that exceed the memory capacity of a single machine.
- PySpark's MLlib library provides tools for scalable machine learning, including classification, regression, clustering, and recommendation algorithms, enabling seamless integration into data workflows.
- PySpark integrates easily with various data sources like HDFS, Hive, Cassandra, and JDBC-compliant databases. It also supports modern formats like Parquet and JSON, simplifying data ingestion and export.

5.1 FPGrowth Algorithm using Pyspark

Code:

```
import pandas as pd
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_date, col
from pyspark.sql.functions import collect_list
from pyspark.ml.fpm import FPGrowth
from pyspark.sql.functions import collect_set


# Initialize Spark session
spark = SparkSession.builder.appName("ExcelDataProcessing").getOrCreate()

# Load the CSV file into a PySpark DataFrame
csv_file_path = "Groceries_dataset.csv" # Convert the xlsx to csv first
spark_df = spark.read.option("header", "true").csv(csv_file_path)


# Convert Date to DateType
spark_df = spark_df.withColumn("Date", to_date(col("Date"), "yyyy-MM-dd"))

# Convert Member_number to Integer
spark_df = spark_df.withColumn("Member_number", col("Member_number").cast("int")
    )
```

```
# Group items by Member_number and Date
transactions_df = spark_df.groupBy("Member_number", "Date") \
    .agg(collect_list("itemDescription").alias("items"))


# Group items by Member_number and Date, ensuring unique items per transaction
transactions_df = spark_df.groupBy("Member_number", "Date") \
    .agg(collect_set("itemDescription").alias("items")) # Use collect_set for
        unique items

fpGrowth = FPGrowth(itemsCol="items", minSupport=0.01, minConfidence=0.1)
model = fpGrowth.fit(transactions_df)

# Frequent Itemsets
model.freqItemsets.show(truncate=False)

# Association Rules
model.associationRules.show(truncate=False)
```

Output:

```
+------------------------------------------------+----+
|items                                           |freq|
+------------------------------------------------+----+
|[specialty cheese]                              |71  |
|[chocolate marshmallow]                         |60  |
|[pet care]                                      |85  |
|[pet care, rolls/buns]                          |40  |
|[pet care, other vegetables]                    |40  |
|[house keeping products]                        |45  |
|[flower (seeds)]                                |67  |
|[curd]                                          |471 |
|[curd, sausage]                                 |125 |
|[curd, sausage, rolls/buns]                     |59  |
|[curd, sausage, rolls/buns, whole milk]         |39  |
|[curd, sausage, yogurt]                         |58  |
|[curd, sausage, yogurt, whole milk]             |39  |
|[curd, sausage, other vegetables]               |58  |
|[curd, sausage, other vegetables, whole milk]|40  |
|[curd, sausage, soda]                           |52  |
|[curd, sausage, whole milk]                     |74  |
|[curd, frankfurter]                             |77  |
|[curd, frankfurter, rolls/buns]                 |45  |
|[curd, frankfurter, whole milk]                 |48  |
+------------------------------------------------+----+
only showing top 20 rows
```

antecedent	consequent	confidence	lift	support
[bottled beer, rolls/buns, whole milk]	[sausage]	0.3221476510067114	1.5638001788594782	0.012314007183170857
[bottled beer, rolls/buns, whole milk]	[other vegetables]	0.5033557046979866	1.3365671232375693	0.019240636223704463
[bottled beer, rolls/buns, whole milk]	[tropical fruit]	0.3288590604026846	1.4071269126780073	0.012570548999486916
[bottled beer, rolls/buns, whole milk]	[yogurt]	0.3624161073825503	1.2807778663437726	0.013853258081067214
[bottled beer, rolls/buns, whole milk]	[bottled water]	0.2751677852348993	1.287639888170033	0.01051821446895844
[bottled beer, rolls/buns, whole milk]	[soda]	0.3288590604026846	1.0490119619064358	0.012570548999486916
[bottled beer, rolls/buns, whole milk]	[root vegetables]	0.28187919463087246	1.222208120880023	0.0107747562852745
[frankfurter, rolls/buns, whole milk]	[other vegetables]	0.525	1.3940395095367848	0.01616213442791175
[frankfurter, rolls/buns, whole milk]	[yogurt]	0.35	1.2368993653671805	0.0107747562852745
[frankfurter, rolls/buns, whole milk]	[bottled water]	0.325	1.520828331332533	0.010005130836326322
[frankfurter, rolls/buns, whole milk]	[soda]	0.425	1.3556873977086743	0.013083632632119035
[frankfurter, rolls/buns, whole milk]	[root vegetables]	0.3416666666666667	1.4814423433444568	0.01051821446895844
[bottled beer, yogurt, other vegetables]	[whole milk]	0.6385542168674698	1.3936642426368406	0.013596716264751155
[margarine, soda]	[sausage]	0.2468354430379747	1.198212399703643	0.010005130836326322
[margarine, soda]	[yogurt]	0.3227848101265823	1.1407209337021014	0.013083632632119035
[margarine, soda]	[bottled water]	0.25316455696202533	1.1846764022064522	0.01026167265264238
[margarine, soda]	[rolls/buns]	0.43670886075949367	1.2489296692887062	0.017701385325808106
[margarine, soda]	[other vegetables]	0.44936708860759494	1.1932104301038182	0.018214468958440224
[margarine, soda]	[whole milk]	0.5379746835443038	1.174146313804981	0.02180605438686506
[margarine, soda]	[shopping bags]	0.25949367088607594	1.5419303797468356	0.01051821446895844

Figure 7: Frequent Itemsets and rules