DATA ANALYTICS-1

ASSIGNMENT-2

PART-2

Team members:

Saloni Goyal (2023115001) - Part 2 (From BUC)

Sachi Thonse Rao (2023101120) (Part1+Part2 AOI)

PART-2

Preprocessing:

1. **Data types: Proper**

```
#check for the datatype of each column
print(df.dtypes)
# print(df.info())
```

✓ 0.0s

```
Campaign_ID                          object
Product_ID                           object
Budget                              float64
Clicks                                int64
Conversions                           int64
Revenue_Generated                   float64
ROI                                 float64
Customer_ID                          object
Subscription_Tier                    object
Subscription_Length                   int64
Flash_Sale_ID                        object
Discount_Level                        int64
Units_Sold                            int64
Bundle_ID                            object
Bundle_Price                        float64
Customer_Satisfaction_Post_Refund     int64
Common_Keywords                      object
dtype: object
```

2. **Null values: No**

```python
#check for null values
print(df.isnull().sum())
# print(df.head(5))
```

✓ 0.0s

```
Campaign_ID                          0
Product_ID                           0
Budget                               0
Clicks                               0
Conversions                          0
Revenue_Generated                    0
ROI                                  0
Customer_ID                          0
Subscription_Tier                    0
Subscription_Length                  0
Flash_Sale_ID                        0
Discount_Level                       0
Units_Sold                           0
Bundle_ID                            0
Bundle_Price                         0
Customer_Satisfaction_Post_Refund    0
Common_Keywords                      0
dtype: int64
```

3. Unique values per column:

```python
# Count unique values per column
unique_counts = df.nunique().sort_values(ascending=False)

# Display the result
print("Unique values per column:\n")
print(unique_counts)
```

✓ 0.0s

```
Unique values per column:

Campaign_ID                          10000
Customer_ID                          10000
Product_ID                           10000
Flash_Sale_ID                        10000
Bundle_ID                            10000
Revenue_Generated                     9997
Budget                                9995
Bundle_Price                          8977
Clicks                                4298
Conversions                            999
ROI                                    451
Units_Sold                             199
Discount_Level                          60
Subscription_Length                     35
Customer_Satisfaction_Post_Refund        4
Common_Keywords                          4
Subscription_Tier                        3
dtype: int64
```

Since there are no null values and all columns have the correct data types, no additional preprocessing is needed.

We will also drop the five columns that contain 10,000 unique values each, as they would not contribute to aggregation. Including them would still result in 10,000 rows, which goes against the purpose of aggregation.

Hence, dropping the columns Campaign_ID, Product_ID, Customer_ID, Flash_Sale_ID, Bundle_ID from the dataset.

Attribute Oriented Induction works on a dataset by extracting generalized knowledge like patterns, rules, etc. For generalization we use different methods like binning, etc.

Column Name	After Binning
Subscription_Length_Group	Short, Medium, Long
Revenue_Level	Low, Medium, High
Budget-Level	Low, Medium, High
ROI_Level	Low, Medium, High
Bundle_Price_Level	Low, Medium, High
Discount_Level_Group	Low, Medium, High
Units_Sold_Group	Low, Medium, High
Satisfaction_Level	Low, Medium, High

But the output file “AOI_1.csv” has 8536 rows, which is not a very good generalization. Hence, let’s generalize more:

Column Name	After Binning
Subscription_Length_Group	Short, Long
Revenue_Level	Low, High
Budget-Level	Low, High
ROI_Level	Low, High
Bundle_Price_Level	Low, High
Discount_Level_Group	Low, Medium, High
Units_Sold_Group	Low, Medium, High
Satisfaction_Level	Low, Medium, High

Not doing for the remaining three, as their frequency of unique values are already low, hence binning more won’t impact that much.

This leads to an output file “AOI_2.csv” which has around 5000 rows.

Generalizing even further:

Column Name	After Binning
Subscription_Length_Group	Short, Long
Revenue_Level	Low, High
Budget-Level	Low, High
ROI_Level	Low, High
Discount_Level_Group	Low, High
Satisfaction_Level	Low, High

This generalization leads to around 50 rows, the data is stored in AOI_3.csv.

Excessive generalization will also lead to loss of valuable information. But AOI helped in condensing the large dataset into manageable summaries. It now helps in knowledge extraction i.e. we can extract rules and insights from the raw data after analysis.

To analyze the data, we can take the top 10 characteristic rules which are stored in Top_Characteristic_Rules.csv.

1. Subscription Tiers

- Basic and Premium users dominate the top rules.
- Standard users appear only in Rule 1, but with the highest frequency (7).
- This suggests that Premium and Basic tiers drive the majority of recurring patterns.

2. Subscription Length

- Long-term subscribers (Rules 1, 2, 4, 5, 7, 8, 10) dominate the rules.
- Indicates that long-term commitment is strongly correlated with high activity and recurring patterns.
- Short-term Premium users (Rule 3) are linked to higher revenue and medium satisfaction.

3. Revenue & ROI

- High ROI appears in all 10 rules → campaigns are generally profitable in these clusters.
- Revenue is mostly Medium, except Rule 3 (High revenue for Premium Short subscribers) and Rule 7/8 (Low revenue clusters).
- Suggests ROI is more stable than absolute revenue, meaning even low-revenue groups achieve high ROI when budgets are low.

4. Budgets and Discounts

- Low budget clusters dominate (6 out of 10 rules).
- Even with low spending, high ROI is achieved, likely due to high discounts.

- Most frequent rules (1, 2, 4, 5) show High discounts, emphasizing their role in driving sales.

5. Customer Satisfaction

- Low satisfaction dominates (9 out of 10 rules).
- Even in high-revenue or high-ROI groups, satisfaction remains low → indicating a post-purchase experience gap.
- Only Rule 10 shows High satisfaction, linked to Basic subscribers, long-term, medium revenue, high discounts, innovative products.

6. Common Keywords (Product Perceptions)

- Affordable (Rules 1, 2, 6, 9) is the most frequent keyword → value-for-money perception drives customer engagement.
- Innovative (Rules 5, 8, 10) → tied with both long-term Basic and Premium clusters.
- Stylish/Durable appear in niche segments (Premium Short, Basic Long).

Some implications on business strategies:

1. Retention Strategy: Long-term subscribers are the most active, but low satisfaction could threaten retention. Post-purchase support and quality improvements are critical.
2. Discount Effectiveness: High discounts strongly drive ROI and sales, but may be unsustainable long-term. Businesses should evaluate if profit margins are being compromised.
3. Premium Segment Opportunity: Premium subscribers (Rules 3, 5, 6, 7, 9) frequently appear in the top rules, suggesting targeted premium bundles and loyalty benefits could further enhance ROI.
4. Customer Experience Gap: Across tiers, satisfaction is low despite good ROI/revenue → indicates that while marketing campaigns succeed, product/service quality improvements are needed.
5. Product Positioning: Affordable products dominate, but Innovative and Stylish products also show strong patterns. Balancing affordability with innovation may broaden appeal.

BUC Algorithm Implementation:

From a marketing perspective, for different subscription tiers (such as premium, basic, etc.), customers with subscription lengths of around 3-4 months often use common keywords like "affordable." Marketing strategies target these customers by allocating specific budgets, tracking the number of clicks generated on ads, and measuring how many customers convert to subscriptions. The overall impact is then evaluated by analyzing the revenue generated, the number of units sold, and the customer satisfaction levels after the campaign. This helps marketers understand the effectiveness of their campaigns and optimize future efforts accordingly.

Attributes not getting added because of all 10k unique values:

1. Campaign_ID
2. Product_Id
3. Customer_ID
4. Flash_Sale_ID
5. Bundle_ID

We are also not using:

1. Common_Keywords: Because it's related to the Discount_Level column. If the discount is huge, common_keyword is most places seen to be affordable. Hence, not need to include this attribute as some rows which will appear will give no new information.
2. Bundle_Price: Because I am focusing on marketing effectiveness, bundle price is not a required attribute.
3. ROI: Revenue per investment can be generated by (Budget/revenue_generated)*100, hence this attribute will be redundant as well.

My dimensions: (Categorical arguments)

1. Subscription_tier
2. Subscription_length
3. Discount_Level

My measures: (Numerical arguments which help explain the dimensions by different mathematical measures like sum, avg, count, etc.)

1. Budgets
2. Clicks
3. Conversions
4. Revenue_generated
5. Units_sold
6. Customer_satisfaction_post_refund

1. **In-Memory implementation**

The Bottom-Up Cube (BUC) In-Memory algorithm recursively groups the dataset by each dimension and computes aggregates for the measure columns. It also adds "ALL" rows to represent roll-ups across dimension values.

I used the method of recursion to recursively form an aggregate combination with my base value being an empty array.

We added an attribute frequency as well which tells us the frequency of each combination.

Output: Dataset of size 7472*10.

2. **Out-Memory implementation**

All data and intermediate cube that can't be stored in RAM are solved here using paging. Its slow, and complex compared to In-memory, but efficient how huge datasets.

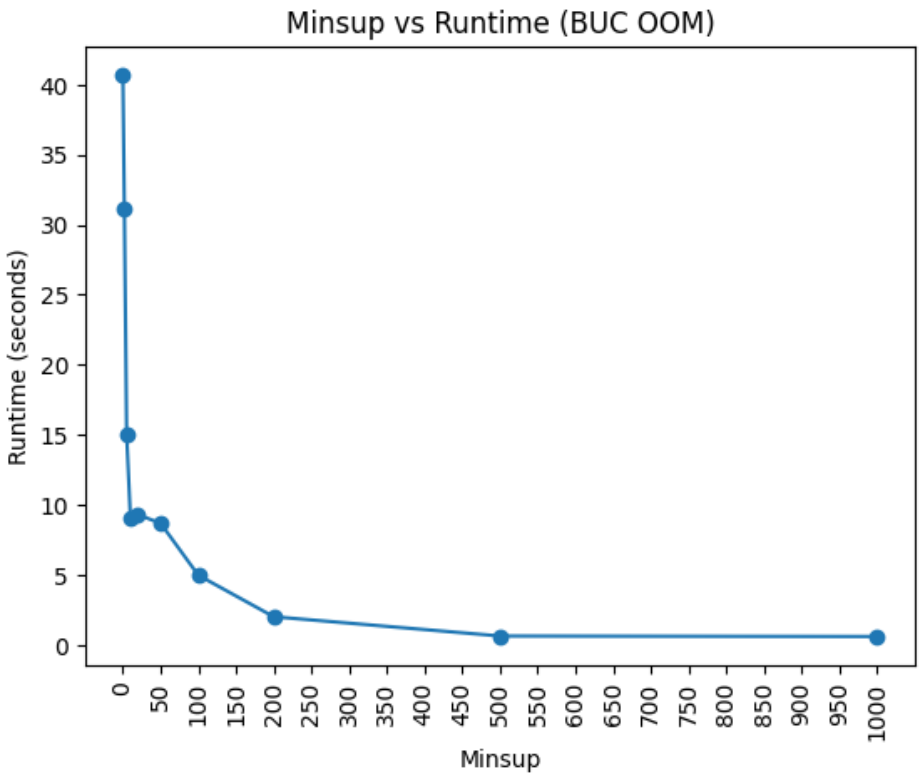
I divided the dataset into 10 chunks of 1000, and then applied the Out_BUC algorithm, it's saved in BUC_Out.csv with the total rows 7472*9 as we are not including the frequency attribute here.

Performance Analysis:

Minsup: Threshold for frequency

1. **A plot of minsup vs runtime, keeping allotted memory fixed.**
   If the memory is fixed, we can use the Out-memory BUC Algorithm by Paging and then pruning the aggregations which are below certain minimum support.
   Let's assume different supports e.g. 0,2,5,10.., etc. And check for the performance.

```
minsup=0, runtime=41.43s, result_shape=(7472, 10)
minsup=2, runtime=31.34s, result_shape=(5350, 10)
minsup=5, runtime=13.42s, result_shape=(1621, 10)
minsup=10, runtime=8.41s, result_shape=(438, 10)
minsup=20, runtime=8.20s, result_shape=(384, 10)
minsup=50, runtime=7.79s, result_shape=(346, 10)
minsup=100, runtime=4.41s, result_shape=(132, 10)
minsup=200, runtime=2.01s, result_shape=(39, 10)
minsup=500, runtime=0.54s, result_shape=(4, 10)
minsup=1000, runtime=0.53s, result_shape=(4, 10)
```

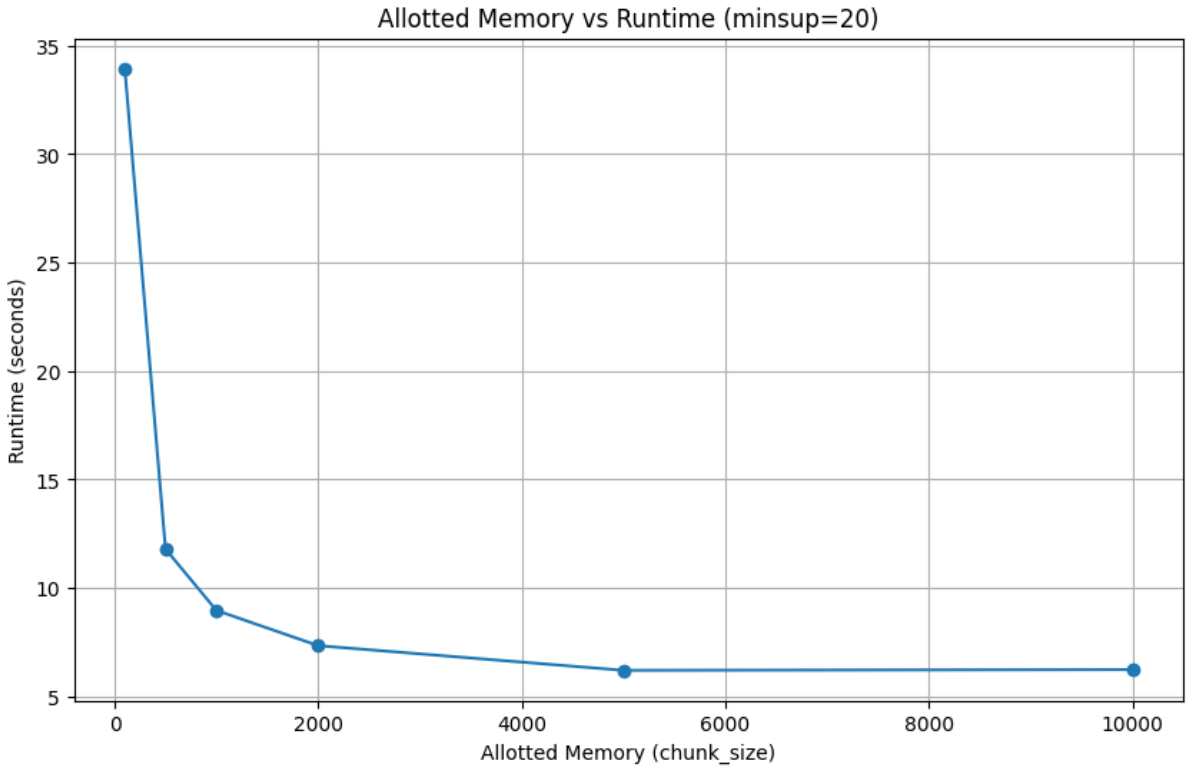


- At minsup = 0 → Runtime ~ 40s
  Since there's no pruning, the algorithm computes all possible combinations. Runtime is highest.
- At minsup = 50 → Runtime drops sharply to ~32s
  Now, groups with support < 50 are removed. A significant portion of the cube is pruned.
- At minsup = 100–500 → Runtime keeps decreasing (14s → 2s → 1s)
  Each increase in minsup removes more low-frequency groups. The recursive calls in BUC become shallower.
- At minsup ≥ 500 → Runtime almost flat (~1s)
  Very few combinations survive. The cube is small, so runtime is minimal.

We can see if minsup increases, more rows are getting pruned and hence runtime is decreasing.

2. **<u>A plot of allotted memory vs runtime, keeping minsup fixed.</u>**
   Let's assume minsup to be 20 and apply the same Buc algorithm as applied for previous part.

```
chunk_size=100, runtime=33.95s, result_shape=(384, 10)
chunk_size=500, runtime=11.79s, result_shape=(384, 10)
chunk_size=1000, runtime=8.96s, result_shape=(384, 10)
chunk_size=2000, runtime=7.33s, result_shape=(384, 10)
chunk_size=5000, runtime=6.19s, result_shape=(384, 10)
chunk_size=10000, runtime=6.23s, result_shape=(384, 10)
```



- For smaller chunk sizes (100), the runtime was the highest at approximately 34 seconds.
- As the chunk size increased, the runtime consistently decreased:
- 500: 11.79 seconds
- 1000: 8.96 seconds
- 2000: 7.33 seconds
- 5000: 6.19 seconds
- 10000: 6.23 seconds
- The runtime improvement is significant from chunk size 100 to 5000, with diminishing returns beyond 5000.

Increasing chunk size are reducing the processing time due to decreased overhead and more efficient batch processing. However, after a certain point (around chunk size 5000), further increase in chunk size is providing minimal runtime improvements. Therefore, an optimal chunk size balances processing efficiency and system resource constraints, with chunk sizes around 5000 to 10000 showing the best performance in this case.

Optimization technique:

Let's choose Iceberg Cubing. It is an optimization technique for data cube computation where only aggregates that meet a minimum support threshold are computed and stored. Aggregates with counts below this threshold are pruned early in the analysis.

For each dimension (or combination of dimensions), only groups with at least minsup rows are kept. Any combination that occurs fewer than minsup=20 times is ignored completely.

ınd	Frequency	
.59	60	
.61	65	
.57	64	
.45	64	
.42	57	
.46	63	
.49	59	
:14	89	
.63	63	
.46	57	
.69	71	
.59	63	
.29	51	
.54	61	
.43	57	
.57	64	
.62	64	
.38	54	
.59	62	
.66	66	
.43	52	
.28	56	
.68	61	
.28	49	
.51	63	
.41	54	
.89	79	
.61	58	
.52	62	
.74	69	
.18	49	
.44	55	
.63	63	
.30	54	
.24	59	ALL
;46	3416	
.63	59	
.70	62	
.24	56	
.44	58	
.67	68	
.35	53	
.40	59	
.42	57	

We can see that the frequency column of the resultant dataset has all the combinations whose frequency>minsup i.e. >20 here.

And hence we are getting 171 rows after the pruning in the resultant dataset "buc_iceberg.csv" for this case.

This reduces the number of aggregates and give us only the most frequent aggregates hence helping us generalize and come to a conclusion easily.

Comparison of BUC and AOI:

	AOI	**BUC**
Primary purpose	To generalize and summarize data by replacing specific attributes with higher level generalisations of by removing the attributes which are irrelevant.	To compute data cubes efficiently by aggregating over multiple dimensions.
Use cases	Identifying generalized trends e.g. Most customers from urban areas prefer product X	Multidimensional analysis e.g. Finding frequent item sets in large datasets.
Insights or patterns it discovers	Gives insights about which attributes are significant for summarization. Eg, in our implementation we aggregated on revenue level, budget level, etc.	Gives detailed frequency counts for each combination of attributes, helping identify high support patterns.
Computational efficiency and Scalability	Works well on medium sized datasets. Relatively efficient as well.	Works well on large datasets where paging and pruning are required. Hence because of increased functionalities it can be computationally expensive.
Interpretability of the results produced	Results like bundle_price are summarised as low, medium, high. So, we converted numerical data to categorical data, which makes it easy for generalised interpretation.	Results are highly detailed and quantitative. The number of rows in the resultant dataset are more than the actual dataset (for the attributes we are choosing for analysis). Hence, we need to have diff visualization techniques like performance analysis's to visualise and understand better.
Scenario where one is preferred over the other	For generalised insights, understanding which attributes are more relevant.	For detailed Multidimensional

		aggregation, data cube analysis, etc.