Project 2

Transactional data

Team members

Menna Abdo Saeed Ahmed	**23102385**
Menna Allah Osama Khalil	**23102157**
Maryam Ali Mohammed	**23102387**
Catherine Adel Zaki Gaballah	**23102369**
Mohammed Ayman Mohammed	**23101455**

-We chose this **transactional** dataset from Kaggle to work on

https://www.kaggle.com/code/nandinibagga/apriori-algorithm/input

```
df = pd.read_csv("bread basket.csv")
df
```

✓ 0.1s

	Transaction	Item	date_time	period_day	weekday_weekend
0	1	Bread	30-10-2016 09:58	morning	weekend
1	2	Scandinavian	30-10-2016 10:05	morning	weekend
2	2	Scandinavian	30-10-2016 10:05	morning	weekend
3	3	Hot chocolate	30-10-2016 10:07	morning	weekend
4	3	Jam	30-10-2016 10:07	morning	weekend
...	...	...	...	...	...
20502	9682	Coffee	09-04-2017 14:32	afternoon	weekend
20503	9682	Tea	09-04-2017 14:32	afternoon	weekend
20504	9683	Coffee	09-04-2017 14:57	afternoon	weekend
20505	9683	Pastry	09-04-2017 14:57	afternoon	weekend
20506	9684	Smoothies	09-04-2017 15:04	afternoon	weekend

20507 rows × 5 columns

I. Data Cleaning:

- It is an unclean dataset that has(1620) **duplicated** rows, so we decided to remove these duplicates

```
print(df.duplicated().sum())
```
✓ 0.0s

```
1620
```

Generate | + Code | + Markdown

```
df = df.drop_duplicates(subset=['Transaction', 'Item', 'date'], keep='first')
df
```
✓ 0.0s

I.I The **Item** column contained inconsistent formatting, including different separators, unwanted characters, and placeholder values.

- Handling inconsistent separators:
Some records used semicolons (;) instead of commas (,). All semicolons were replaced with commas to maintain consistency.

```
#If Item column sometimes contains separators like ';' replace with ',' to handle inconsistent separators
df['Item'] = df['Item'].astype(str).str.replace(';', ',')
```

- Removing non-breaking spaces:
Unicode non-breaking spaces (\u00A0) were replaced with regular spaces to avoid hidden formatting issues.

```
#Remove stray unicode non-breaking spaces
df['Item'] = df['Item'].str.replace('\u00A0', ' ')
```

- Normalizing text format:
All text was converted to lowercase and leading/trailing whitespace was removed. This ensures that similar items are treated as identical **(e.g., "Milk" and " milk ").**

```
#Normalize whitespace, lower, strip
df['Item'] = df['Item'].astype(str).str.strip().str.lower()
```

- Removing placeholder and invalid values
Rows containing meaningless or placeholder values such as "none", "nan", or empty strings were removed from the dataset.

- Converting Date-Time Data:
The **date_time** column was converted into a proper datetime format, enabling time-based feature extraction.

```
df['date_time'] = pd.to_datetime(df['date_time'])
df.head()
```
✓ 0.0s

- Extracting Date and Time Features:

To support time-based analysis, several features were derived from the **date_time** column:

- **Date extraction**

```
# Extracting date
df['date'] = df['date_time'].dt.date
```

This creates a separate column containing only the calendar date.

- **Time extraction**

```
#Extracting time
df['time'] = df['date_time'].dt.time
```

This captures the exact time of the transaction.

- Extracting and Labeling Month:

The numeric month was extracted and then replaced with its corresponding month name for better interpretability.

```
# Extracting month and replacing it with text
df['month'] = df['date_time'].dt.month
df['month'] = df['month'].replace((1,2,3,4,5,6,7,8,9,10,11,12),
                                  ('January','February','March','April','May','June','July','August',
                                   'September','October','November','December'))
```

- Extracting and Categorizing Hour:

The hour was extracted from the datetime field.
Hours were grouped into time intervals **(e.g., 9–10, 10–11)** to make temporal patterns easier to analyze.

```
# Extracting hour
df['hour'] = df['date_time'].dt.hour
# Replacing hours with text
hour_in_num = (1,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23)
hour_in_obj = ('1-2','7-8','8-9','9-10','10-11','11-12','12-13','13-14','14-15',
               '15-16','16-17','17-18','18-19','19-20','20-21','21-22','22-23','23-24')
df['hour'] = df['hour'].replace(hour_in_num, hour_in_obj)
```

- Extracting and Labeling Weekday:

The numeric weekday (0–6) was extracted and converted into readable weekday names.

```
# Extracting weekday and replacing it with text
df['weekday'] = df['date_time'].dt.weekday
df['weekday'] = df['weekday'].replace((0,1,2,3,4,5,6),
                                      ('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'))
```

- Dropping Redundant Column

After extracting all required features, the original date_time column was removed to avoid redundancy.

```
# dropping date_time column
df.drop('date_time', axis = 1, inplace = True)
```

Data after cleaning:

	Transaction	Item	period_day	weekday_weekend	date	time	month	hour	weekday
0	1	bread	morning	weekend	2016-10-30	09:58:00	October	9-10	Sunday
1	2	scandinavian	morning	weekend	2016-10-30	10:05:00	October	10-11	Sunday
3	3	hot chocolate	morning	weekend	2016-10-30	10:07:00	October	10-11	Sunday
4	3	jam	morning	weekend	2016-10-30	10:07:00	October	10-11	Sunday
5	3	cookies	morning	weekend	2016-10-30	10:07:00	October	10-11	Sunday
...	...	...	...	...	...	...	...	...	...
20502	9682	coffee	afternoon	weekend	2017-04-09	14:32:00	April	14-15	Sunday
20503	9682	tea	afternoon	weekend	2017-04-09	14:32:00	April	14-15	Sunday
20504	9683	coffee	afternoon	weekend	2017-04-09	14:57:00	April	14-15	Sunday
20505	9683	pastry	afternoon	weekend	2017-04-09	14:57:00	April	14-15	Sunday
20506	9684	smoothies	afternoon	weekend	2017-04-09	15:04:00	April	15-16	Sunday

18887 rows × 9 columns

II. Transaction Construction and Market Basket Preparation

- Creating Transaction-Level Item Lists

To prepare the data for market basket analysis, individual item records were grouped by transaction:

```
transactions = df.groupby('Transaction')['Item'].apply(list).reset_index(name='items_list')
transactions.to_csv("transactions.csv", index=False)
transactions
```

✓ 0.2s

	Transaction	items_list
0	1	[bread]
1	2	[scandinavian]
2	3	[hot chocolate, jam, cookies]
3	4	[muffin]
4	5	[coffee, pastry, bread]
...	...	...
9460	9680	[bread]
9461	9681	[truffles, tea, spanish brunch, christmas common]
9462	9682	[muffin, tacos/fajita, coffee, tea]
9463	9683	[coffee, pastry]
9464	9684	[smoothies]

9465 rows × 2 columns

- Each **Transaction ID** is converted into a list of items purchased together.

- This transformation produces a transactional dataset suitable for association rule mining.

III. One-Hot Encoding (Basket Matrix)

- Transaction Encoding

To apply the Apriori algorithm, transactions were converted into a binary (one-hot encoded) basket format:

```
te = TransactionEncoder()
te_array = te.fit(transactions['items_list']).transform(transactions['items_list'])
basket = pd.DataFrame(te_array, columns=te.columns_)
#Save basket matrix
basket.to_csv("basket.csv", index=False)
print("Basket (one-hot) shape:", basket.shape)
basket.head()
```

✓ 0.4s Python

```
Basket (one-hot) shape: (9465, 94)
```

	adjustment	afternoon with the baker	alfajores	argentina night	art tray	bacon	baguette	bakewell	bare popcorn	basket	...	the bart	the nomad	tiffin	toast	truffles	tshirt	valentine's card	vegan feast
0	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False

- Each column represents a unique item.

- Each row represents a transaction.

- A value of True indicates the presence of an item in a transaction.

IV. Custom Apriori Algorithm

- Frequent Itemset Mining:

A custom implementation of the Apriori algorithm was used to allow full control over the process and thresholds.

Key Characteristics:
Converts the basket matrix into a Boolean NumPy array for efficient computation.

Identifies frequent:
1) 1-itemsets
2) 2-itemsets
3) Higher-order itemsets (until no further frequent itemsets are found).
4) Applies Apriori pruning (all subsets must be frequent).
5) Calculates support at each iteration.

```
def custom_apriori(df, min_support, use_colnames=False, max_len=None, verbose=0):

    X = df.values.astype(bool)
    n_rows = X.shape[0]
    n_cols = X.shape[1]

    # Calculate support for single items
    support_dict = {}
    itemset_dict = {}

    # Level 1: Single items
    support_series = np.sum(X, axis=0) / n_rows
    col_indices = np.arange(n_cols, dtype=np.int32)

    # Filter by minimum support
    mask = support_series >= min_support
    itemset_dict[1] = col_indices[mask].reshape(-1, 1)
    support_dict[1] = support_series[mask]

    if verbose >= 1:
        print(f"Level 1: Found {len(support_dict[1])} frequent 1-itemsets")

    max_itemset = 1

    # Generate larger itemsets
    while max_itemset < (max_len or float('inf')) and len(itemset_dict[max_itemset]) > 0:
        # Generate candidate itemsets of size max_itemset + 1
        candidates = []
        itemsets = itemset_dict[max_itemset]
```

```
                # Check if this candidate is already in the list
                if valid:
                    is_duplicate = False
                    for existing_candidate in candidates:
                        if np.array_equal(union, existing_candidate):
                            is_duplicate = True
                            break

                    if not is_duplicate:
                        candidates.append(union)

        if len(candidates) == 0:
            break

        # Calculate support for candidates
        next_itemset = max_itemset + 1
        support_list = []
        valid_itemsets = []

        for candidate in candidates:
            # Convert candidate to int32 to use as index
            candidate_indices = np.asarray(candidate, dtype=np.int32)

            # Count transactions containing all items in candidate
            count = np.sum(np.all(X[:, candidate_indices], axis=1))
            sup = count / n_rows

            if sup >= min_support:
                support_list.append(sup)
                valid_itemsets.append(candidate_indices)

        if len(valid_itemsets) == 0:
            break
```

```
        itemset_dict[next_itemset] = np.array(valid_itemsets, dtype=object)
        support_dict[next_itemset] = np.array(support_list)

        if verbose >= 1:
            print(f"Level {next_itemset}: Found {len(support_dict[next_itemset])} frequent {next_itemset}-itemsets")

        max_itemset = next_itemset

    # Convert to DataFrame
    all_res = []
    for k in sorted(itemset_dict.keys()):
        support_series_k = pd.Series(support_dict[k])

        # Convert to column names if requested
        if use_colnames:
            itemsets_list = []
            for itemset in itemset_dict[k]:
                itemset_names = frozenset([df.columns[int(i)] for i in itemset])
                itemsets_list.append(itemset_names)
        else:
            itemsets_list = [frozenset(itemset) for itemset in itemset_dict[k]]

        itemsets_series = pd.Series(itemsets_list, dtype='object')

        res = pd.concat([support_series_k, itemsets_series], axis=1)
        all_res.append(res)

    res_df = pd.concat(all_res, ignore_index=True)
    res_df.columns = ['support', 'itemsets']
    res_df = res_df.reset_index(drop=True)

    return res_df
```

```
# Generate combinations
for combo in combinations(range(len(itemsets)), 2):
    i, j = combo
    itemset_i = itemsets[i].astype(np.int32)
    itemset_j = itemsets[j].astype(np.int32)

    # Merge itemsets that differ in only one item
    union = np.union1d(itemset_i, itemset_j).astype(np.int32)

    if len(union) == max_itemset + 1:
        # Check if all subsets are frequent
        valid = True
        if max_itemset >= 2:
            for sub_combo in combinations(union, max_itemset):
                sub_itemset = np.array(list(sub_combo), dtype=np.int32)
                found = False
                for item in itemsets:
                    if np.array_equal(sub_itemset, item.astype(np.int32)):
                        found = True
                        break
                if not found:
                    valid = False
                    break

        # Check if this candidate is already in the list
        if valid:
            is_duplicate = False
            for existing_candidate in candidates:
                if np.array_equal(union, existing_candidate):
                    is_duplicate = True
                    break
```

Minimum support threshold.
Maximum itemset length.
Optional verbose output.
Conversion of item indices to column names.

A **minimum support threshold of 2%** was used to filter out infrequent patterns.

```
# Set minimum support threshold
min_support = 0.02

# Use custom Apriori implementation instead of built-in
freq_itemsets = custom_apriori(basket, min_support=min_support, use_colnames=True, verbose=1)

# Sort by support in descending order
freq_itemsets = freq_itemsets.sort_values(by='support', ascending=False).reset_index(drop=True)
```

V. Association Rule Generation

A custom function was implemented to generate association rules from the frequent itemsets.

```
def custom_association_rules(freq_itemsets, metric="confidence", min_threshold=0.01):

    # Check required columns
    if not all(col in freq_itemsets.columns for col in ['support', 'itemsets']):
        raise ValueError("DataFrame must contain 'support' and 'itemsets' columns")

    # Build support dictionary
    support_dict = {}
    for _, row in freq_itemsets.iterrows():
        itemset = frozenset(row['itemsets'])
        support_dict[itemset] = row['support']

    # Lists to store results
    antecedents_list = []
    consequents_list = []
    antecedent_support_list = []
    consequent_support_list = []
    support_list = []
    confidence_list = []
    lift_list = []

    # Loop through each itemset
    for itemset in support_dict.keys():
        itemset_support = support_dict[itemset]

        # Skip single items
        if len(itemset) < 2:
            continue

        # Generate all possible antecedent-consequent combinations
        for i in range(1, len(itemset)):
            for antecedent_tuple in combinations(itemset, i):
                antecedent = frozenset(antecedent_tuple)
                consequent = itemset - antecedent
```

```
# Check if support exists
if antecedent not in support_dict or consequent not in support_dict:
    continue

antecedent_sup = support_dict[antecedent]
consequent_sup = support_dict[consequent]

# Calculate metrics
confidence = itemset_support / antecedent_sup if antecedent_sup > 0 else 0
lift = confidence / consequent_sup if consequent_sup > 0 else 0




# Select metric for filtering
if metric == "confidence":
    metric_value = confidence
elif metric == "lift":
    metric_value = lift
elif metric == "support":
    metric_value = itemset_support
else:
    metric_value = confidence

# Add rule if meets threshold
if metric_value >= min_threshold:
    antecedents_list.append(antecedent)
    consequents_list.append(consequent)
    antecedent_support_list.append(antecedent_sup)
    consequent_support_list.append(consequent_sup)
    support_list.append(itemset_support)
    confidence_list.append(confidence)
    lift_list.append(lift)
```

```
    # Create results DataFrame
    rules_df = pd.DataFrame({
        'antecedents': antecedents_list,
        'consequents': consequents_list,
        'antecedent support': antecedent_support_list,
        'consequent support': consequent_support_list,
        'support': support_list,
        'confidence': confidence_list,
        'lift': lift_list
    })

    return rules_df
```

Metrics Calculated:
Support: Frequency of the itemset in all transactions.
Confidence: Likelihood of the consequent given the antecedent.
Lift: Strength of association compared to random co-occurrence.

The function:
Iterates through all frequent itemsets with two or more items.
Generates all valid antecedent–consequent combinations.
Filters rules based on the chosen metric and threshold.

To make it more readable:

```
rules = custom_association_rules(freq_itemsets, metric="confidence", min_threshold=0.01)
#Add readable strings for antecedents/consequents
rules['antecedents_str'] = rules['antecedents'].apply(lambda x: ', '.join(list(x)))
rules['consequents_str'] = rules['consequents'].apply(lambda x: ', '.join(list(x)))

#Sort rules for inspection
rules = rules.sort_values(['confidence','lift','support'], ascending=[False, False, False]).reset_index(drop=True)
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	antecedents_str	consequents_str
0	(toast)	(coffee)	0.033597	0.478394	0.023666	0.704403	1.472431	toast	coffee
1	(medialuna)	(coffee)	0.061807	0.478394	0.035182	0.569231	1.189878	medialuna	coffee
2	(pastry)	(coffee)	0.086107	0.478394	0.047544	0.552147	1.154168	pastry	coffee
3	(juice)	(coffee)	0.038563	0.478394	0.020602	0.534247	1.116750	juice	coffee
4	(sandwich)	(coffee)	0.071844	0.478394	0.038246	0.532353	1.112792	sandwich	coffee
5	(cake)	(coffee)	0.103856	0.478394	0.054728	0.526958	1.101515	cake	coffee
6	(cookies)	(coffee)	0.054411	0.478394	0.028209	0.518447	1.083723	cookies	coffee
7	(hot chocolate)	(coffee)	0.058320	0.478394	0.029583	0.507246	1.060311	hot chocolate	coffee
8	(tea)	(coffee)	0.142631	0.478394	0.049868	0.349630	0.730840	tea	coffee
9	(pastry)	(bread)	0.086107	0.327205	0.029160	0.338650	1.034977	pastry	bread
10	(bread)	(coffee)	0.327205	0.478394	0.090016	0.275105	0.575059	bread	coffee
11	(cake)	(tea)	0.103856	0.142631	0.023772	0.228891	1.604781	cake	tea
12	(cake)	(bread)	0.103856	0.327205	0.023349	0.224822	0.687097	cake	bread
13	(tea)	(bread)	0.142631	0.327205	0.028104	0.197037	0.602181	tea	bread
14	(coffee)	(bread)	0.478394	0.327205	0.090016	0.188163	0.575059	coffee	bread
15	(tea)	(cake)	0.142631	0.103856	0.023772	0.166667	1.604781	tea	cake
16	(coffee)	(cake)	0.478394	0.103856	0.054728	0.114399	1.101515	coffee	cake
17	(coffee)	(tea)	0.478394	0.142631	0.049868	0.104240	0.730840	coffee	tea
18	(coffee)	(pastry)	0.478394	0.086107	0.047544	0.099382	1.154168	coffee	pastry
19	(bread)	(pastry)	0.327205	0.086107	0.029160	0.089119	1.034977	bread	pastry

VI. Identify Strong Association Rules

To focus on the most meaningful relationships, stronger rules were filtered using stricter thresholds:

```
#display the strongest rules
strong_rules = rules[(rules['confidence'] >= 0.3) & (rules['lift'] >= 1.2)].sort_values('lift', ascending=False)
print("\nStrong rules (confidence>=0.3 and lift>=1.2):")
if strong_rules.empty:
    print("No strong rules found with these thresholds. Try lowering thresholds.")
else:
    display(strong_rules[['antecedents_str','consequents_str','support','confidence','lift']])
```

Strong rule is between **Toast** and **Coffee**

```
Strong rules (confidence>=0.3 and lift>=1.2):
```

	antecedents_str	consequents_str	support	confidence	lift
0	toast	coffee	0.023666	0.704403	1.472431

Interpretation Criteria:

Confidence ≥ 0.3: The consequent occurs in at least 30% of cases where the antecedent is present.
Lift ≥ 1.20: The association is at least 20% stronger than random chance.

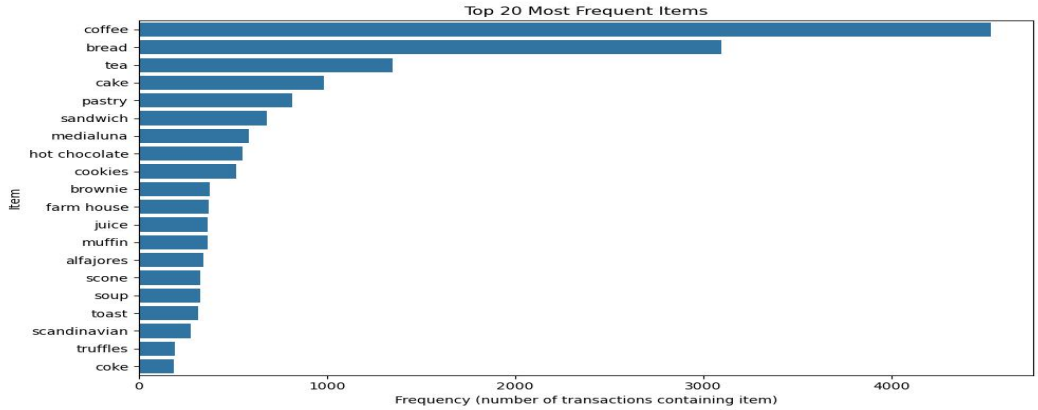
If no rules met these criteria, **the thresholds can be relaxed to explore weaker patterns**.

VII. Visualization:

To better understand item frequencies and the strength of discovered association rules, several visualizations were generated.

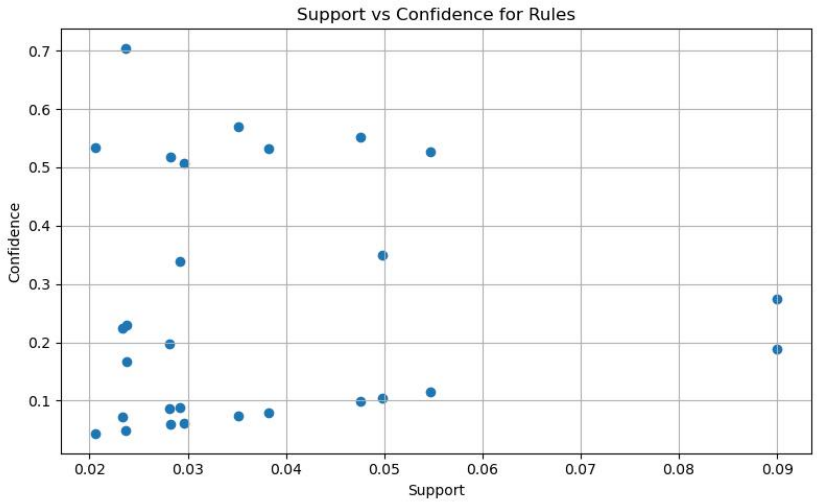
1. Top Frequent Items

This analysis identifies the most frequently purchased items based on the number of transactions in which each item appears. The frequency was computed by summing the one-hot encoded basket matrix across all transactions. Visualizing the top items provides insight into overall customer demand and highlights high-volume products that may warrant priority in inventory management, promotions, or shelf placement.



2. Support vs Confidence

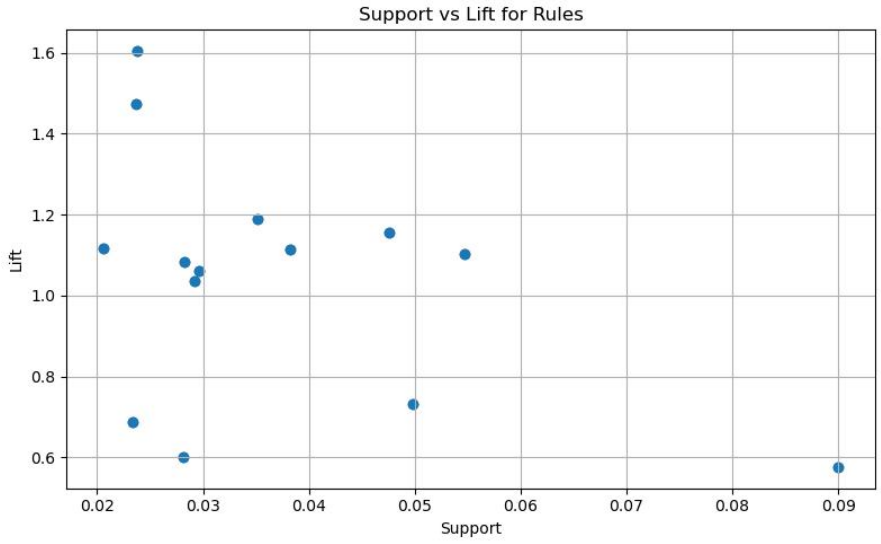
The relationship between support and confidence was examined to understand how frequently rules occur and how reliable they are. Support measures how often a rule appears in the dataset, while confidence reflects the conditional probability of the consequent given the antecedent.



This scatter plot reveals that rules with very high confidence often have lower support, indicating strong but less frequent associations. Conversely, higher-support rules tend to exhibit moderate confidence, reflecting common but less deterministic purchasing patterns.

3.Support vs Lift

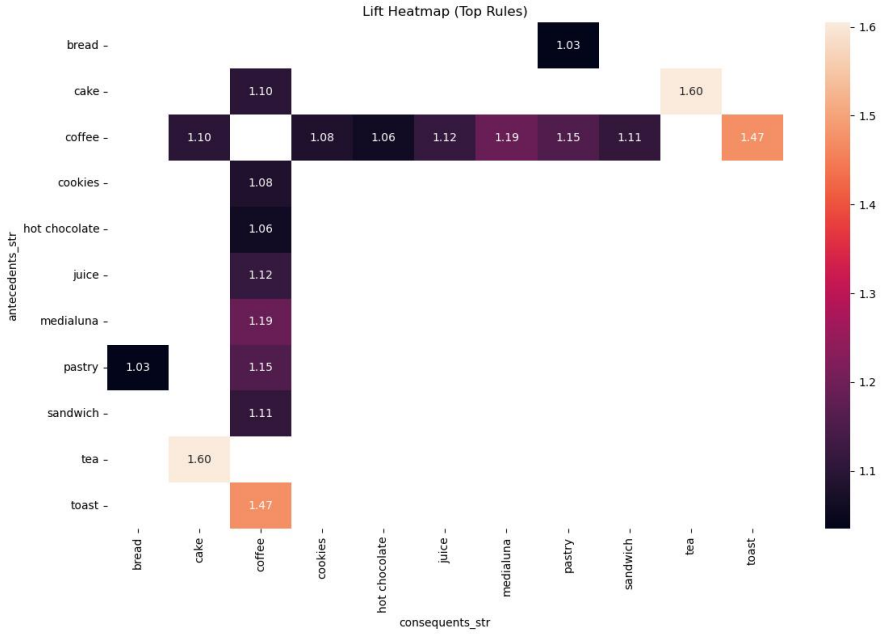
Support and lift were compared to assess the strength and usefulness of discovered rules. Lift measures how much more likely the consequent is to occur with the antecedent than would be expected by chance.



Rules with high lift values and reasonable support are of particular interest, as they represent meaningful and non-random associations that can be leveraged for recommendation or bundling strategies.

4. Lift Heatmap of Top Rules

To further examine the strongest associations, a heatmap was constructed using the top rules ranked by lift. Antecedents and consequents were arranged along the axes, with color intensity representing lift magnitude.



5. Network Graph of Association Rules

A directed network graph was used to visualize the structure of the strongest association rules. In this graph, nodes represent itemsets, and directed edges represent association rules from antecedent to consequent. Edge weights correspond to lift values.