Isabella Correa, Gabriel Colonna

Introduction and system design

Intro:

- This program works as a supermarket transaction explorer. It creates shopping transactions and then runs tests on the data sets. The algorithms test for frequent itemsets and association rules. It also takes it a step further and recommends best products following trends.

System Design:

- Front End: Using Streamlit to create and render HTML templates, upload files, product selection and the entire UI.
- Back End: Separate python code load the data, algorithms, and run preprocessing/data cleaner.
- Data: We used sample data and csv files that are then loaded into the program for algorithm use.

Data preprocessing approach

- Data is input through either manual selection, csv upload, or using the sample data. Csv uploads require that the information already be organized for processing. Unknown items are automatically removed and duplicates are erased. After the automatic process, we run the preprocessing to filter out any errors that were previously missed to ensure the cleanest data set for algorithms to run.

Before Preprocess Cleaning:
"Total_transactions":100

"Empty_transactions":5

"Single_item_transactions":5

"Duplicate_items_found":9

"Invalid_items_found":2

After Preprocess Cleaning:
"Valid_transactions":89

"Total_items":276

"unique_products":30

"Transactions_removed":11

"items_removed":11

Isabella Correa, Gabriel Colonna

Algorithm implementation details (include pseudocode)

Apriori:

Apriori is a level-wise breadth-first search using horizontal transactions. It merges frequent itemsets while pruning those with infrequent subsets. Support counts are tracked and any qualifying itemsets are used for rule generation.

Pseudocode:

```
// Level 1: Count single items
item_counter ← Counter()
FOR each transaction IN transaction_list:
   item_counter.update(transaction)

frequent_itemsets ← {frozenset([item]): count
                  FOR item, count IN item_counter.items()
                  IF count ≥ min_support_count}

current_level ← keys(frequent_itemsets)
k ← 2

// Level-wise generation
WHILE current_level is not empty:
   candidates ← GENERATE_CANDIDATES(current_level, k)
   IF candidates is empty:
      BREAK

   candidate_counts ← COUNT_CANDIDATES(transaction_list, candidates)
   current_level ← {itemset FOR itemset, count IN candidate_counts.items()
                 IF count ≥ min_support_count}

   frequent_itemsets.update({itemset: candidate_counts[itemset]
                       FOR itemset IN current_level})
   k ← k + 1

RETURN FrequentItemsetResult(frequent_itemsets, transaction_count, min_support)
```

Isabella Correa, Gabriel Colonna

Eclat:

Eclat is depth first search with vertical transaction representation. Each item has a set of which transaction ID's contain that item. Support count is tracked between sets. Recursion continues until the transaction support count falls between the minimum support threshold.

Pseudocode:

```
// Build TID-sets (vertical representation)
tid_sets ← empty dictionary
FOR tid, transaction IN enumerate(transaction_list):
    FOR each item IN transaction:
        tid_sets[item].add(tid)

// Filter to frequent items only
initial_items ← sorted([(item, tids)
                        FOR item, tids IN tid_sets.items()
                        IF length(tids) ≥ min_support_count])

frequent_itemsets ← empty dictionary

RECURSIVE(prefix, items):
    FOR idx, (item, tid_set) IN enumerate(items):
        new_itemset ← frozenset(prefix + (item,))
        support_count ← length(tid_set)

        IF support_count < min_support_count:
            CONTINUE

        frequent_itemsets[new_itemset] ← support_count

        // Generate suffix candidates via intersection
        suffix_candidates ← empty list
        FOR next_item, next_tid_set IN items[idx+1:]:
            intersection ← tid_set ∩ next_tid_set
            IF length(intersection) ≥ min_support_count:
                suffix_candidates.append((next_item, intersection))
        IF suffix_candidates is not empty:
            RECURSIVE(tuple(sorted(new_itemset)), suffix_candidates)

RECURSIVE(empty tuple, initial_items)
RETURN FrequentItemsetResult(frequent_itemsets, transaction_count, min_support)
```

Closet:

Closet algorithm finds frequent itemset by first finding all itemsets using depth-first id intersections similar to the Eclat algorithm. It uses recursive search to find all frequent itemsets, then it filters those itemsets by checking that no superset has the same support count.

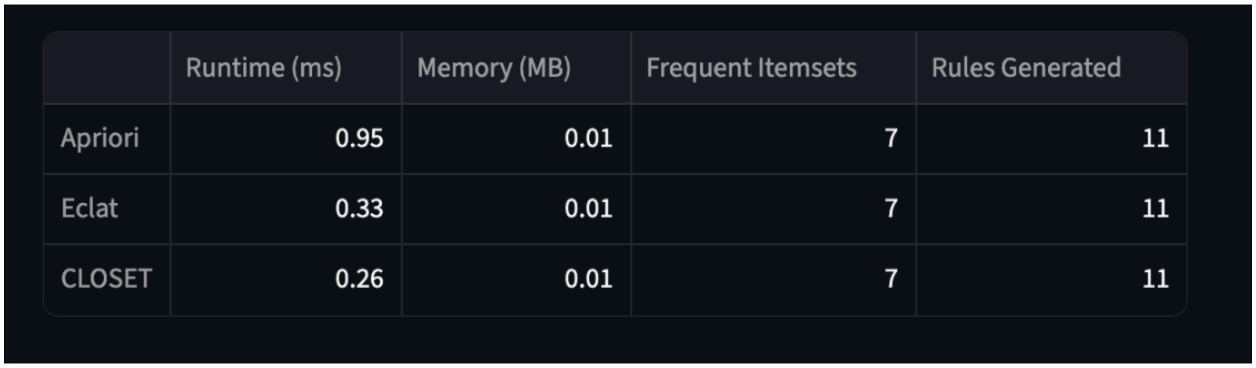
Pseudocode:

```
// Build TID-sets (vertical representation)
tid_sets ← empty dictionary
FOR tid, transaction IN enumerate(transaction_list):
    FOR each item IN transaction:
        tid_sets[item].add(tid)

// Filter to frequent items only
initial_items ← sorted([(item, tids)
                        FOR item, tids IN tid_sets.items()
                        IF length(tids) ≥ min_support_count])

// Phase 1: Mine all frequent itemsets (like Eclat)
all_frequent_itemsets ← empty dictionary

MINE_FREQUENT(prefix, items):
    FOR idx, (item, tid_set) IN enumerate(items):
        new_itemset ← frozenset(prefix + (item,))
        support_count ← length(tid_set)

        IF support_count < min_support_count:
            CONTINUE
        all_frequent_itemsets[new_itemset] ← support_count

        // Generate suffix candidates via intersection
        suffix_candidates ← empty list
        FOR next_item, next_tid_set IN items[idx+1:]:
            intersection ← tid_set ∩ next_tid_set
            IF length(intersection) ≥ min_support_count:
                suffix_candidates.append((next_item, intersection))

        IF suffix_candidates is not empty:
            MINE_FREQUENT(tuple(sorted(new_itemset)), suffix_candidates)

MINE_FREQUENT(empty tuple, initial_items)

// Phase 2: Filter to closed itemsets only
closed_itemsets ← empty dictionary
FOR itemset, support_count IN all_frequent_itemsets.items():
    is_closed ← True
    FOR other_itemset, other_support IN all_frequent_itemsets.items():
        IF itemset ≠ other_itemset AND
           itemset ⊆ other_itemset AND
           other_support = support_count:
            is_closed ← False
            BREAK
    IF is_closed:
        closed_itemsets[itemset] ← support_count
RETURN FrequentItemsetResult(closed_itemsets, transaction_count, min_support)
```

Isabella Correa, Gabriel Colonna

Performance analysis and comparison

- Performance is measured by runtime in milliseconds. On our web application you'll see the runtime of each. We also keep a count of the amount of rules that are generated by each algorithm.
- To compare the algorithms, this is the process:
  - Run three algorithms on the same cleaned_transactions function and then compare the runtime_ms and counts reported directly into the UI.
  - When we have larger baskets, Eclat usually outperforms the other algorithms in the runtime. However, it also uses much more memory in comparison.
  - I have added an example of our comparison table after running the program.

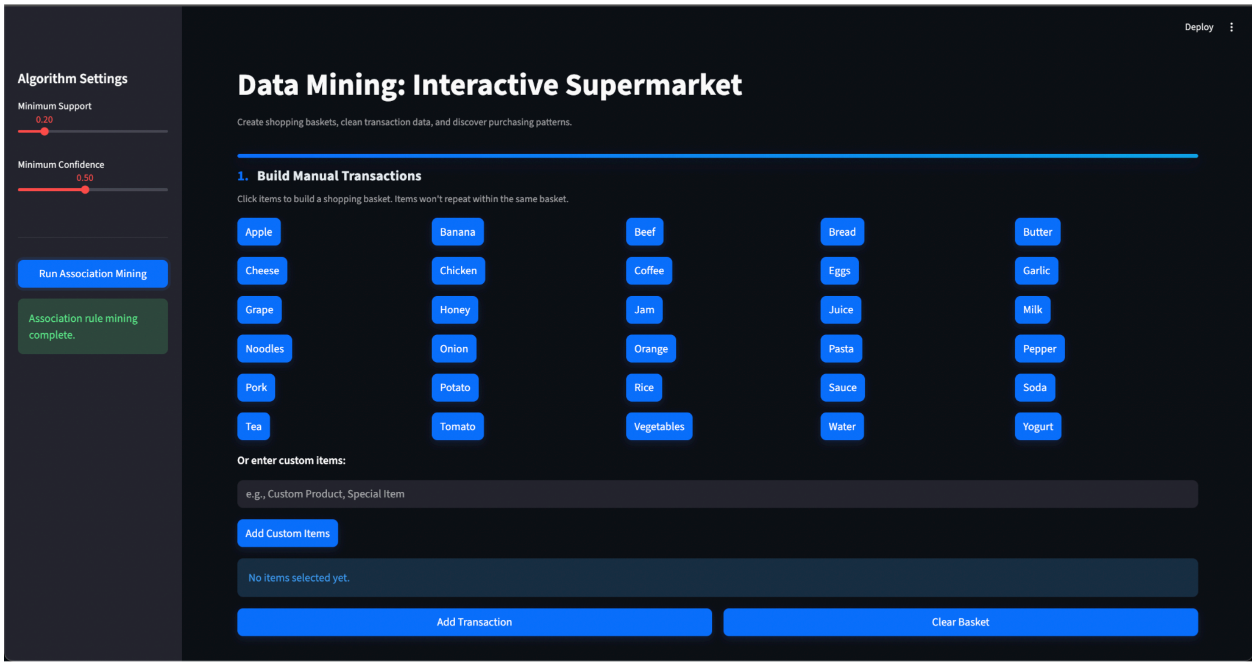


	Runtime (ms)	Memory (MB)	Frequent Itemsets	Rules Generated
Apriori	0.95	0.01	7	11
Eclat	0.33	0.01	7	11
CLOSET	0.26	0.01	7	11

Isabella Correa, Gabriel Colonna

User interface design

We went with a dark mode theme for the UI. The program is designed for the user to go through each step in order. First the user can build a manual transaction list, or upload with a csv.Then the data is analyzed, showing total transactions, total items, unique items, empty transactions, single item transactions, duplicate items, and invalid items. Next the user will preprocess the transaction list, cleaning the data by removing any invalid or empty items/ transactions. Then the user can run association mining using the 3 different algorithms. It will give a comparison table that records the runtime (ms), and memory usage. Then the user can get product recommendations for items found to have an association.



Algorithm Settings

Minimum Support 0.20

Minimum Confidence 0.50

Association rule mining complete.

Data Mining: Interactive Supermarket

Create shopping baskets, clean transaction data, and discover purchasing patterns.

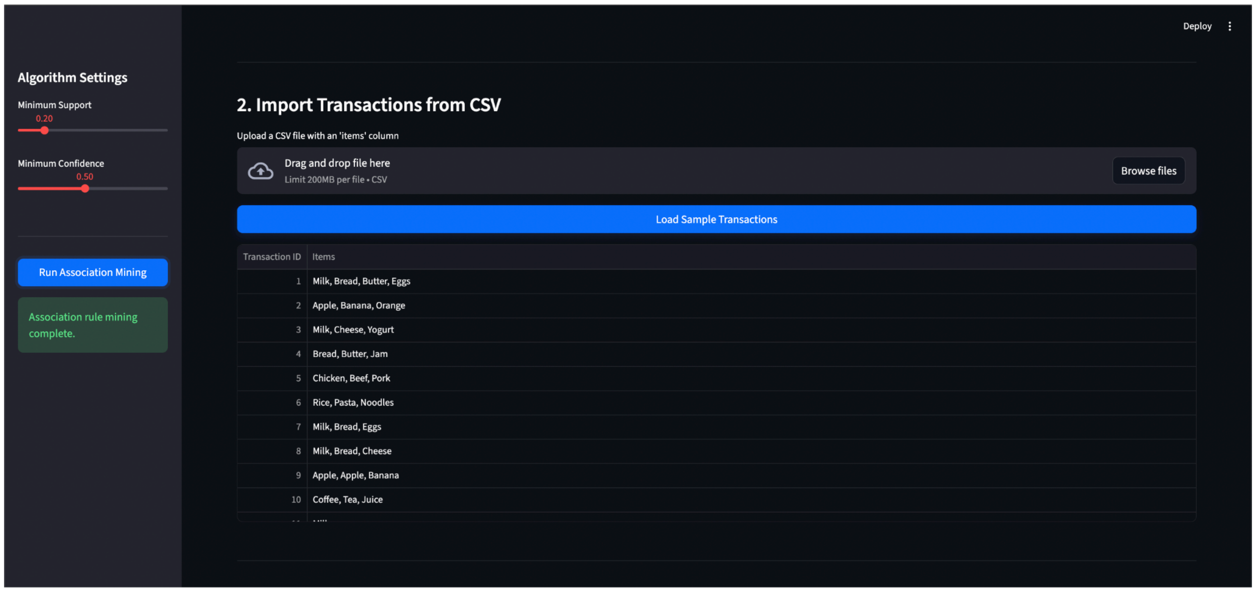
1. Build Manual Transactions

Click items to build a shopping basket. Items won't repeat within the same basket.

Or enter custom items:

e.g., Custom Product, Special Item

No items selected yet.



Algorithm Settings

Minimum Support 0.20

Minimum Confidence 0.50

Association rule mining complete.

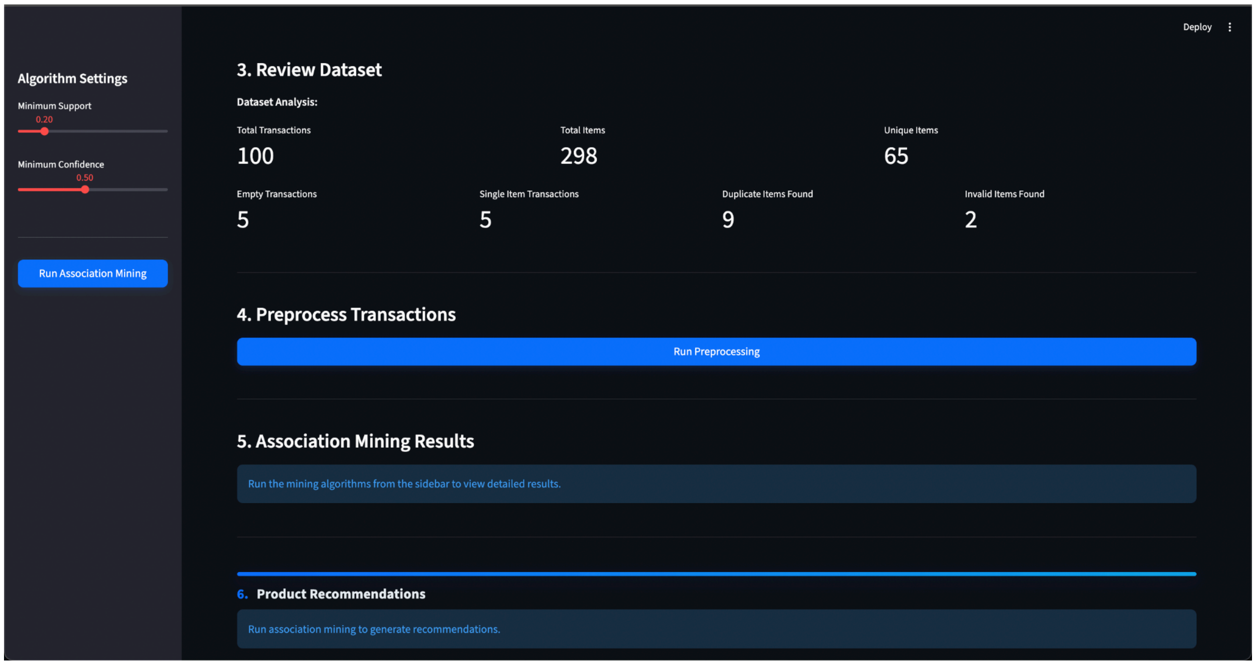
2. Import Transactions from CSV

Upload a CSV file with an 'items' column

Drag and drop file here
Limit 200MB per file • CSV

Transaction ID	Items
1	Milk, Bread, Butter, Eggs
2	Apple, Banana, Orange
3	Milk, Cheese, Yogurt
4	Bread, Butter, Jam
5	Chicken, Beef, Pork
6	Rice, Pasta, Noodles
7	Milk, Bread, Eggs
8	Milk, Bread, Cheese
9	Apple, Apple, Banana
10	Coffee, Tea, Juice

Isabella Correa, Gabriel Colonna



3. Review Dataset

Dataset Analysis:

Total Transactions 100
Total Items 298
Unique Items 65
Empty Transactions 5
Single Item Transactions 5
Duplicate Items Found 9
Invalid Items Found 2

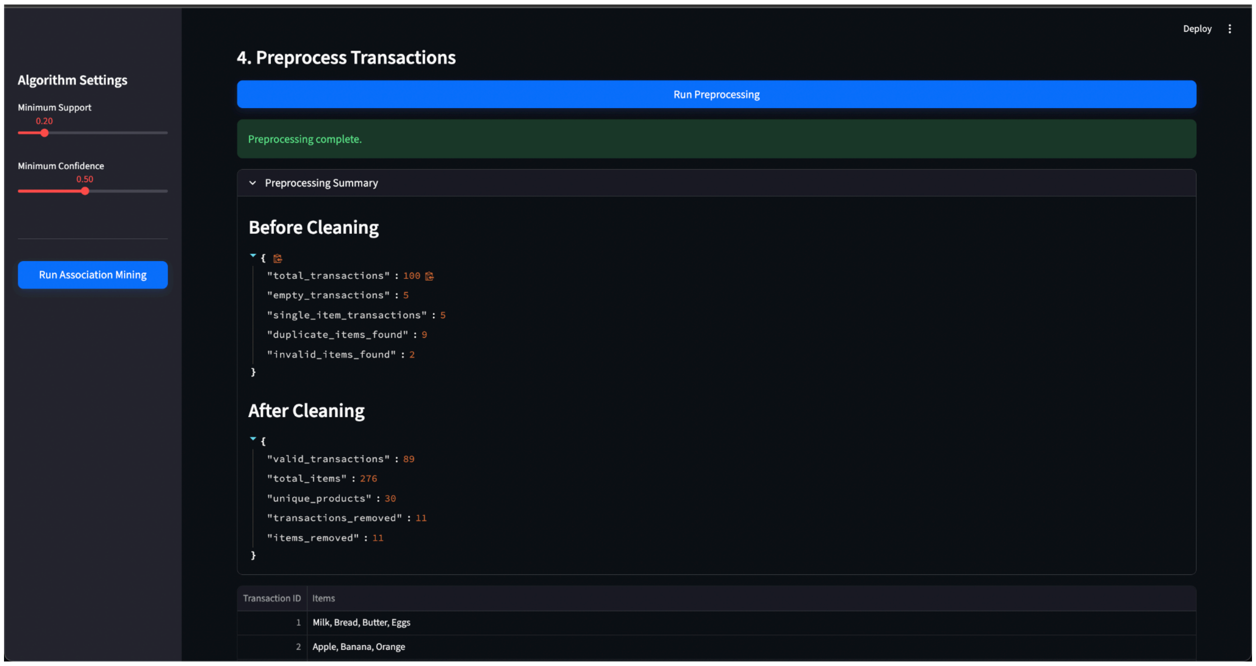
4. Preprocess Transactions

5. Association Mining Results

Run the mining algorithms from the sidebar to view detailed results.

6. Product Recommendations

Run association mining to generate recommendations.



4. Preprocess Transactions

Preprocessing complete.

Preprocessing Summary

Before Cleaning

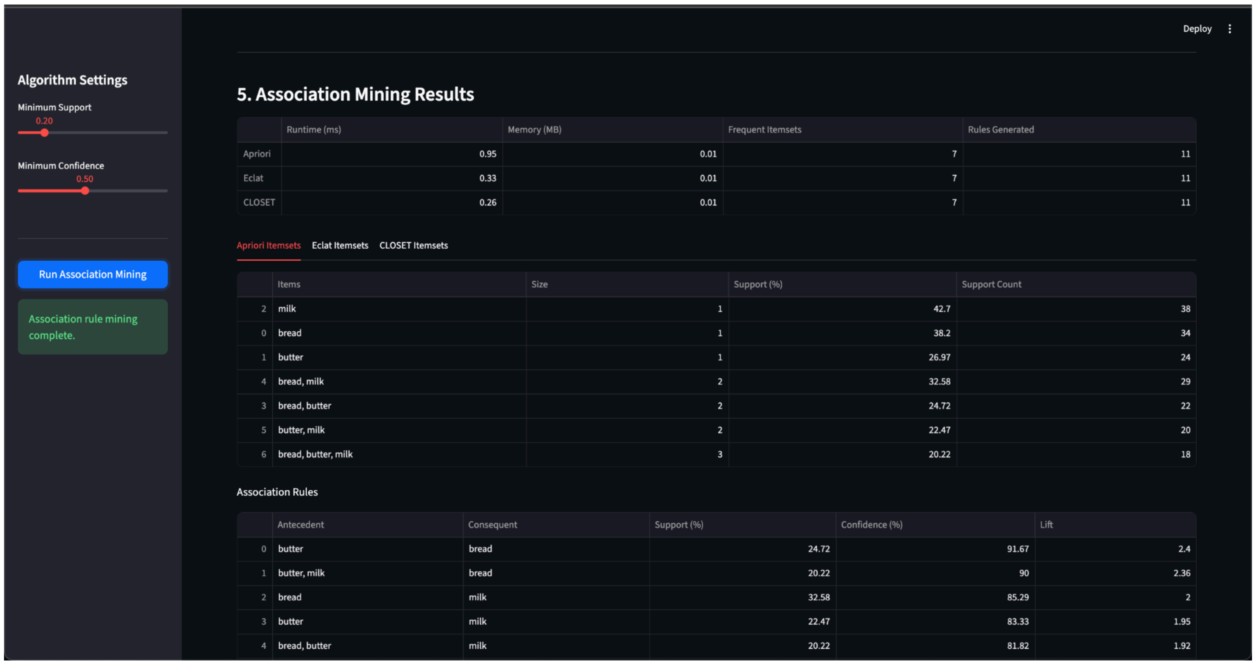
```
{
  "total_transactions" : 100
  "empty_transactions" : 5
  "single_item_transactions" : 5
  "duplicate_items_found" : 9
  "invalid_items_found" : 2
}
```

After Cleaning

```
{
  "valid_transactions" : 89
  "total_items" : 276
  "unique_products" : 30
  "transactions_removed" : 11
  "items_removed" : 11
}
```

Transaction ID	Items
1	Milk, Bread, Butter, Eggs
2	Apple, Banana, Orange

Isabella Correa, Gabriel Colonna



Algorithm Settings

Minimum Support 0.20

Minimum Confidence 0.50

Association rule mining complete.

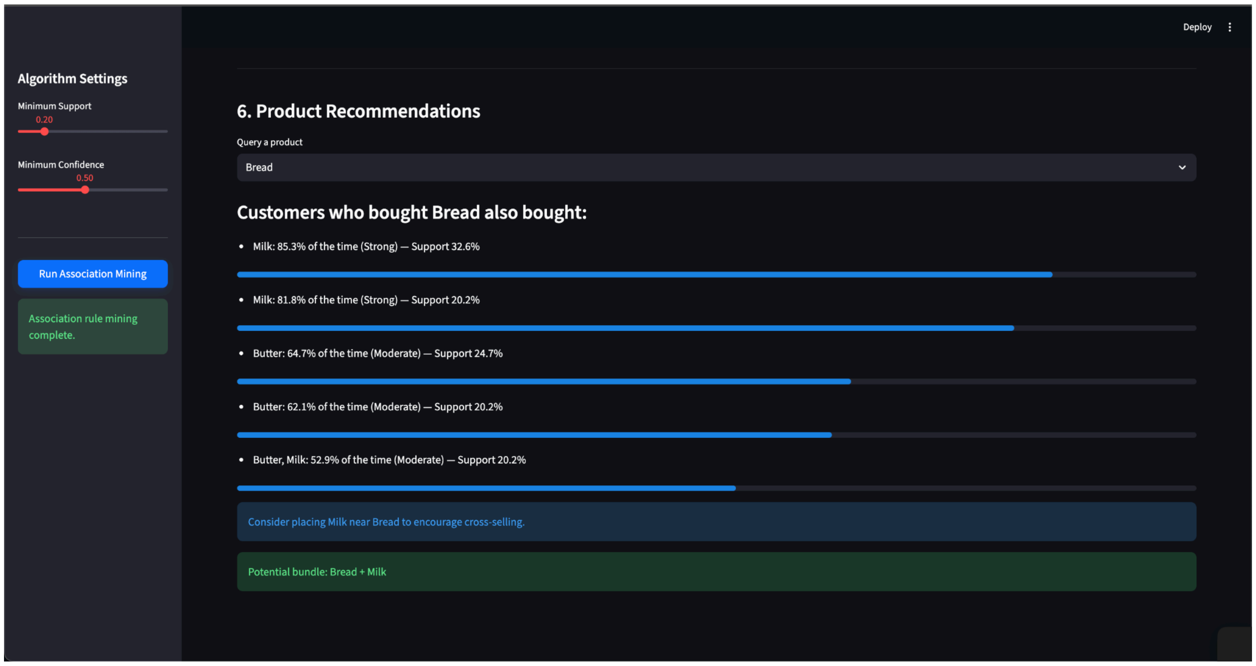
5. Association Mining Results

	Runtime (ms)	Memory (MB)	Frequent Itemsets	Rules Generated
Apriori	0.95	0.01	7	11
Eclat	0.33	0.01	7	11
CLOSET	0.26	0.01	7	11

	Items	Size	Support (%)	Support Count
2	milk	1	42.7	38
0	bread	1	38.2	34
1	butter	1	26.97	24
4	bread, milk	2	32.58	29
3	bread, butter	2	24.72	22
5	butter, milk	2	22.47	20
6	bread, butter, milk	3	20.22	18

Association Rules

	Antecedent	Consequent	Support (%)	Confidence (%)	Lift
0	butter	bread	24.72	91.67	2.4
1	butter, milk	bread	20.22	90	2.36
2	bread	milk	32.58	85.29	2
3	butter	milk	22.47	83.33	1.95
4	bread, butter	milk	20.22	81.82	1.92



Algorithm Settings

Minimum Support 0.20

Minimum Confidence 0.50

Association rule mining complete.

6. Product Recommendations

Query a product: Bread

Customers who bought Bread also bought:

- Milk: 85.3% of the time (Strong) — Support 32.6%
- Milk: 81.8% of the time (Strong) — Support 20.2%
- Butter: 64.7% of the time (Moderate) — Support 24.7%
- Butter: 62.1% of the time (Moderate) — Support 20.2%
- Butter, Milk: 52.9% of the time (Moderate) — Support 20.2%

Consider placing Milk near Bread to encourage cross-selling.

Potential bundle: Bread + Milk

Isabella Correa, Gabriel Colonna

Testing and results

We tested using the sample_transaction.csv.

The data analysis shows:
Total Transactions:100

Total Items: 298

Unique Items: 65

Empty Transactions: 5

Single Item Transactions: 5

Duplicate Items Found: 9

Invalid Items Found: 2

Then we run Association Mining:

	Runtime (ms)	Memory (MB)	Frequent Itemsets	Rules Generated
Apriori	0.95	0.01	7	11
Eclat	0.33	0.01	7	11
CLOSET	0.26	0.01	7	11

We are using a Minimum support of 0.2 (20%) and Minimum Confident of 0.5 (50%).
Closet algorithm takes the least runtime (ms) with 0.26 ms, Eclat with the second fastest runtime at 0.33 ms, and finally Apriori is the slowest runtime at 0.95 ms. All 3 algorithms generate 11 rules. The memory usage for all 3 algorithms is 0.01. The memory usage is found using a recursive function that measures the size of each object. All 3 algorithms generate 7 Frequent Itemsets.

Conclusion and reflection

- In conclusion, this project covers an interactive pipeline that can ingest and normalize data. Its simple UI allows seemingless interaction in order to run these complicated algorithms. However, we have noticed a couple potential future implementations. There is room for growth in this project. For example, the database can be expanded to more accurately reflect larger supermarkets. With a larger database, we can also add a search module that would make manual transactions much easier. Nevertheless, this project has given us a chance to work hands-on with various different data-mining techniques.