Report 2: Itemsets

Group 150: Lorenzo Deflorian, Riccardo Fragale

November 17, 2025

1 Introduction

This homework was about discovering frequent itemsets and generating association rules. These two problems are important especially in the field of sales transaction databases, since companies want to discover and understand the logic behind customers' behavior. The procedure is quite straightforward and we tested it on a dataset provided by the teacher that includes 100,000 generated transactions (baskets) of hashed items. We decided not to use Apache Spark since our algorithms performed efficiently enough.

2 Our Methods

We needed to develop solutions to address the following two problems:

- How to find frequent itemsets with support at least s
- How to generate association rules with confidence at least c from the itemsets found

2.1 A-Priori Algorithm Implementation

We implemented the A-Priori algorithm (standard level-wise frequent itemset mining procedure) in the **apriori** class to solve the first problem. The constructor stores baskets and support threshold s, computing an absolute threshold = $s \times$ len(baskets). The *run* method controls the iterative passes $k = 1, 2, \ldots$ and stops when no new frequent itemsets are found. We call *run_pass(k)* to orchestrate each single pass: it generates singletons for k=1, and for higher k it performs candidate generation, counting, and filtering.

For efficiency, we count itemsets basket-by-basket rather than materializing all combinations at once, which minimizes peak memory usage. The function *has_frequent_subsets(candidate, prev_frequent)* implements Apriori's pruning strategy: all (k-1)-subsets must be frequent for a candidate to be retained. This pruning greatly reduces the number of candidates stored in memory.

We implemented *frequent_by_size* to store persistent frequent itemsets grouped by size, releasing ephemeral candidate data after each pass. The *frequent_items_table* method flattens all discovered frequent itemsets for reporting in tests. Overall, our algorithm follows classic Apriori heuristics to discover frequent itemsets while maintaining controlled memory usage.

2.2 Association Rule Generation

After testing the Apriori implementation, we implemented association rule generation using the **association_rules** class. The *AssociationRule* class serves as a simple data holder containing: antecedent (frozenset), consequent (int), support, confidence, and interest metrics. We initialize the *AssociationRuleGenerator* with frequent itemsets, baskets, and threshold parameters. Internally, we maintain an *ItemsetAnalyzer* instance to compute support, confidence, and interest metrics on demand.

Our *generate()* method iterates through all frequent itemsets and skips singletons (only itemsets of size ≥ 2 produce rules). For each itemset, we call *process_itemset()*, which builds candidate rules by using each item as a consequent and the remaining items as the antecedent. The *generate_rule()* method constructs the antecedent frozenset and uses *ItemsetAnalyzer* to compute the support, confidence, and interest values. It returns an *AssociationRule* object populated with these metrics.

Finally, *filter_rule()* applies minimum support, confidence, and interest thresholds to accept or reject each rule. Accepted rules are appended to the final list returned by *generate()*. Our generator does not create large intermediate data structures; it simply iterates through existing frequent itemsets and computes metrics as needed. The computational cost is mainly determined by *ItemsetAnalyzer* calls, which either scan baskets or reuse precomputed summaries. The output is a flat list of *AssociationRule* objects, ready for sorting or grouping by the caller.

2.3 Testing

We implemented comprehensive tests on the provided dataset to validate our implementation and measure algorithm performance.

3 Results

We ran two main tests: **test_apriori** and **test_rule_generator**.

3.1 Apriori Results

The Apriori test produced the following results:

- 381 frequent itemsets

- Itemsets by size: [375, 6]

The algorithm completed in 0.460 seconds.

3.2 Association Rule Results

In the second test, which includes the Apriori procedure followed by rule generation as described above, we discovered 12 association rules:

```
INFO   AssociationRule({227} -> 390, s=0.0105, c=0.5770, i=0.5502)
INFO   AssociationRule({390} -> 227, s=0.0105, c=0.3907, i=0.3725)
INFO   AssociationRule({346} -> 217, s=0.0134, c=0.3850, i=0.3313)
INFO   AssociationRule({390} -> 722, s=0.0104, c=0.3881, i=0.3296)
INFO   AssociationRule({217} -> 346, s=0.0134, c=0.2486, i=0.2139)
INFO   AssociationRule({682} -> 368, s=0.0119, c=0.2887, i=0.2104)
INFO   AssociationRule({789} -> 829, s=0.0119, c=0.2771, i=0.2090)
INFO   AssociationRule({722} -> 390, s=0.0104, c=0.1783, i=0.1514)
INFO   AssociationRule({829} -> 789, s=0.0119, c=0.1753, i=0.1322)
INFO   AssociationRule({368} -> 682, s=0.0119, c=0.1524, i=0.1111)
INFO   AssociationRule({829} -> 368, s=0.0119, c=0.1753, i=0.0971)
INFO   AssociationRule({368} -> 829, s=0.0119, c=0.1525, i=0.0844
```

The average confidence of the generated rules is **0.2824**, with an average interest of **0.2328**. The total execution time was **0.483** seconds, indicating that the Apriori algorithm is the more computationally expensive step compared to rule generation.

4 Conclusion

This assignment helped us thoroughly understand the steps required to identify association rules. We gained insight into the importance of these techniques, their potential economic impact, and their widespread use, particularly regarding scalability concerns.

In future work, we would like to test our implementation on larger datasets and possibly on data streams to verify whether our expectations regarding performance and scalability hold in practice.