# Report 2: Itemsets

Group 150: Lorenzo Deflorian, Riccardo Fragale

November 15, 2025

## 1 Introduction

This homework was about discovering frequent itemsets and generating association rules. This two problems are important especially in the field of sales transaction databases since companies wants to discover and understand the logic that stands behing customers' behaviours. The procedure is in a way quite straightforward and will be tested on a dataset given by the teacher that includes 100000 generated transactions (baskets) of hashed items. We decided not to use Apache Spark in this homework since our algorithm were doing their job quite fast.

## 2 Our methods

We needed to develop a way to solve the following two questions;

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

In order to solve the first problem we implemented the A-Priori algorithm (standard level-wise frequent itemset mining procedure) in the class **apriori**. The constructor stores baskets, support s and computes an absolute threshold = s * len(baskets). The method *run* controls the iterative passes k = 1, 2, ... and stops when no new frequent itemsets are found. This function is calling *run_pass(k)* that orchestrates a single pass: singletons for k=1, otherwise candidate generation, counting, and filtering. Counting is done basket-by-basket rather than materializing global combinations, minimizing peak memory. Another important function is *has_frequent_subsets(candidate, prev_frequent)* which implements Apriori's pruning: all (k-1)-subsets must be frequent. This pruning step greatly reduces the number of candidates kept in memory. Regarding the storing of data, we have implemented *frequent_by_size* which stores persistent frequent itemsets per size; ephemeral candidate data is released after each pass. while

*frequent_items_table* flattens all discovered frequent itemsets for reporting in tests. The algorithm limits memory by generating candidates only from previous frequent itemsets, not from all combinations. Overall, the code follows classic Apriori heuristics to find frequent itemsets while keeping memory usage controlled.

After testing its functionalities we used that methods and set of new ones in the class called **association_rules** to address the second question. AssociationRule is a small data holder: antecedent (frozenset), consequent (int), support, confidence, interest. AssociationRuleGenerator is initialized with frequent_itemsets, baskets, and threshold paramters. The generator keeps an ItemsetAnalyzer instance to compute support, confidence, and interest on demand. Then there is the method *generate()* that iterates all frequent itemsets and skips singletons (only itemsets of size ¿= 2 produce rules). For each itemset it calls *process_itemset()*, which builds candidate rules from each item by removing that item to form the antecedent. Instead, *generate_rule()* constructs the antecedent frozenset and uses ItemsetAnalyzer to compute support, confidence, and interest. It returns an AssociationRule object populated with these metrics. Finally, *filter_rule()* applies *min_support, min_confidence, and min_interest thresholds* to either accept or reject a rule. Accepted rules are appended to the final list returned by generate(). The generator does not create large intermediate structures, so it's okay in terms of memory usage; it just iterates existing frequent itemsets and computes metrics as needed. Computation cost comes mainly from analyzer calls (support/confidence/interest), which scan baskets or reuse precomputed summaries. The output is a flat list of AssociationRule objects, ready for sorting or grouping by the caller (as done in the test). Logging calls provide progress and debug information but the core API returns pure rule objects for downstream use.

Finally we implemented a set of tests on the dataset given to show that our procedure it is working and also measuring how much time do our algorithms take to find solutions.

# 3 Results

The two main tests are called **test_apriori** and **test_rule_generator**. The first one gave us the following results:

- 381 frequent itemsets
- itemsets by size: [375, 6]

It took 0.460 seconds for the procedure.

Regarding the second test, where actually the first part includes the Apriori procedure as explained before, we found 12 association rules, that are shown below.

```
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
```

Statistically speaking we have an average confidence of **0.2824** and an average interest of **0.2328**. Looking at the time requested it is **0.483** which implies that the longer procedure is the apriori algorithm with respected to the rule generation phase.

# 4 Conclusion

This assignment helped us thoroughly understand the steps required to identify assocaition rules. We also realized how important these techniques are, what might be their economical impact and why they are used especially looking at the possible scalability.

Moreover, we could try in the future to test our modules on bigger datasets, maybe even on data streams to verify in reality whether our expectations on performance and scalability are respected.