# DataMining ID2222 - Homework 2 Discovery of Frequent Itemsets and Association Rules

Sherly Sherly and Anna Martignano

November 25, 2019

## 1 Our Solution

In this section, we will describe the approach used to discover association rules between itemsets in a sales transaction database (a set of baskets). In Section 1.1 the dataset we will analyse the baskets dataset, and in further Sections 1.2, 1.3 we will be describing the two sub-problems to solve the assignment tasks: finding frequent itemsets with support at least s and generating association rules with confidence at least c from the itemsets found in the first step, respectively.

### 1.1 Dataset

The dataset used for this assignment consists of a collection of baskets or transactions, i.e. a list of items bought within a single transaction. Each basket $t$ is a small subset of the set $I$ of all the available items: $t \subset I$.

| Dataset Analysis | |
|---|---|
| Number of Baskets | 100'000 |
| Number of Distinct Items | 870 |
| Largest Basket | 29 items |
| Shortest Basket | 1 item |
| Average Basket size | 10 items |
| Frequent Singletons (s $\geq$ 1%) | 375 |
| Frequent Doubletons (s $\geq$ 1%) | 9 |
| Frequent Tripletons (s $\geq$ 1%) | 1 |
| Frequent Quadruplets (s $\geq$ 1%) | None |

## 1.2 Finding frequent itemsets

In order to find the frequent itemsets it has been implemented a two-pass approach called *A-Priori* which limits the memory demand.
The underlying idea of the A-Priori algorithm is to exploit the monotonicity of support.
The itemset support is the number of baskets containing all the items in that particular itemset.
The monotonicity property of support means that:

- If a set of items appears at least s times, so does every subset.
- The support of a subset is at least as big as the support of its superset.

The previous observations lead to the **downward closure property** of frequent patterns which says:

- Any subset of a frequent itemset must be frequent.
- Contrapositive for pairs: if item i-th does not appear in s baskets, then no pair including i-th can appear in s baskets.

This property it is very useful when implementing the A-Priori algorithm since it is possible to apply a pruning principle: "If there is any itemset which is infrequent, its superset should not be generated/tested, because it's also infrequent".
Let us consider now the two-steps of the A-Priori algorithm:

1. Read baskets and count in main memory the occurrences of each individual item.
   Requires O(n) memory, where n is items.
2. Read baskets again and count only those pairs where both elements are frequent (discovered in Pass 1).
   Requires memory proportional to square of frequent items only (for counts) – $2^m$ instead of $2^n$, where m is $\#frequentitems$ retrieved in the previous step. Plus a list of the frequent items (so you know what must be counted).

It is possible to generalize the previous two steps and build a pipeline, by increasing the set size k of the frequent items. For each k, we construct two sets of k-tuples (sets of size k):

- $C_k$ = candidate k-tuples = those that might be frequent sets (support $> s$) based on information from the pass for k–1

- $L_k$ = the set of truly frequent k-tuples, i.e. filter only those k-tuples from $C_k$ that have support at least s

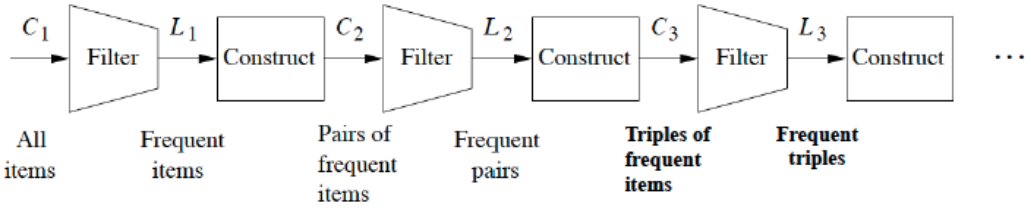


Figure 1: A-Priori Algorithm Pipeline

The code snippet below describes the implementation of the A-Priori algorithm.

```
def get_frequent_Lset(previous_Lset, baskets, s, k):
    """
    This function performs both the "construct" and
    "filter" operations of the A-Priori Algorithm
    pipeline.
    It first generates the set of frequent items based
    on the latest L computed or the initial one, which
    is passed as an input when calling the function.

    Then it constructs the candidates k-tuples:
    (1) For each basket, look in the frequent-items
        table to see which of its items are frequent.
    (2) In a double loop, generate all k-tuples of
        frequent items in that basket.
    (3) For each such k-tuple, add one to its count
        in the data structure used to store counts.

    And finally, it filters the valid candidates k-tuples
    maintaining only the ones with enough support
    """

def apriori_algo():
    """
    This function first initialize L1 by calling the
    previous parse_data() function which reads the
    dataset, and creates the singleton frequent itemset
```

```
    L1 while reading creating couple of
    <singleton,support>.

    Then it simply call the get_frequent_Lset function
    in a loop until the the newest L set contains elements
    in it, and save all the elements in all L sets generated.
    The data structure returned "generated_set" is an array
    of dictionaries in which the k-th position in the array
    contains the dictionary of all the <k-tuple, support>
    pairs generated.
    The 0 position has been initialized bet it is not used.
    """
```

## 1.3 Generating Association Rules

An association rule is an implication $X \to Y$, where $X$ and $Y$ are itemsets such that $X \cap Y = \emptyset$. Support of the rule $X \to Y$ is the number of transactions that contain $X \cup Y$. Confidence of the rule $X \to Y$ is the fraction of transactions containing $X \cup Y$ in all transactions that contain $X$.

The code snippet below describes the generation of association rules. The approach used start by looking only at the frequent itemsets of at least 2-tuples, and evaluates the possible association rules within the element of every frequent tuples.

```
def get_confidence(item_sets, s1, s2):
    """
    This function computes the confidence of the association
    rule s1 \rightarrow s2, where s1 and s2 as given as input
        conf(s1 -> s2) = support(s1 union s2) / support(s1)

    It takes as input as well as input the "generated_set"
    in the item_sets field to retrieve the support of tuples
    computed before.
    """

def generate_associations(item_sets, c):
    """
    This function considers all the frequent itemsets
    of at least 2-tuples, and for each tuple, generates
```

```
    the associations rules among the elements of tuple.
    Then we compute the confidence of associations rules by
    calling the get_confidence() function.
    Only rules with a confidence greater than the threshold c
    are kept.
    """
```

## 2 How to run the code

To develop our application, we have used Jupyter Notebook. Therefore, it is sufficient to launch the Jupyter Notebook environment from the conda shell (3.x) and simply run all the code cells in a consecutive order. Please check that the input dataset .dat is in the same file_path level of the .ipynb notebook.