Data Mining - Assignment 2

Group 18
Gustavo Teodoro D. Beck - Fernando García Sanz

November 13, 2020

1 Data

The data employed correspond to a set of virtual baskets composed of different items. There is a total of 100.000 baskets within the dataset, consisting of different numbers of items. I.e. one transaction can have 5 items in its basket, while another can have 15 items in their basket.

2 Methods

In order to find suitable association rules, the *a-priori* algorithm has been used. This algorithm limits the memory demand by employing several passes over the suitable combinations, filtering them before exploring. It is based on two key ideas:

- If a set of items appears at least **S** [1] times, so does every subset.
- Any subset of a frequent itemset must be frequent. Therefore, if item **i** does not appear in **S** baskets, then no pair including **i** can appear in **s** baskets.

In this case, the objective is finding the association rules of the kind $\{A, B\} \rightarrow C$ that surpass a threshold **S**.

3 Our Application

Our application has mainly 3 phases that will be discussed in the following section and are shown in figure 1.

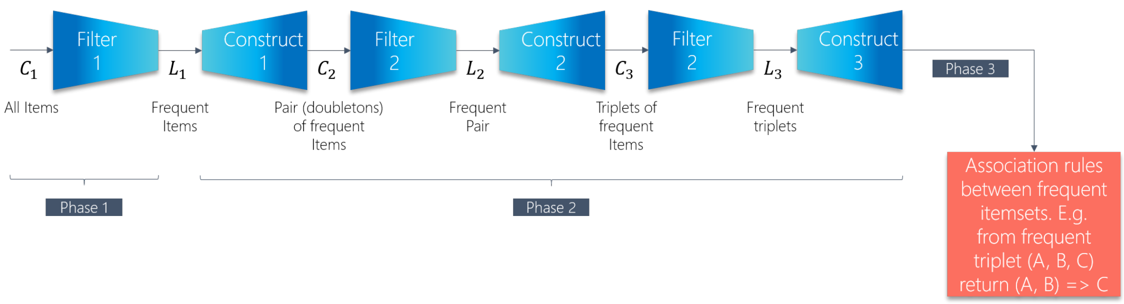

Figure 1: Application workflow.

3.1 Phase: 1 - Frequency of Items and One-Hot-Encoded Matrix

Once the data has been loaded, the first task of the implementation is to identify all of the frequent items. To achieve this, we counted the frequency of all of the items within the data and filtered

[1]Support for itemset **I** is the number of baskets containing all items in **I**

out those that had a frequency below $\mathbf{S} = 1\%$ of the data length, i.e. $\mathbf{S} = 1,000$. In addition, we propose an extra step - build an one-hot-encoded matrix (table 1) of *transactions x all items* - that assessed our application to perform dramatically faster in future steps when identifying the union count between pairs and/or triplets. Our implementation can be seen in figure 2.

$$\text{Boolean Matrix} = \begin{bmatrix} Transaction & Item_1 & Item_2 & \dots & Item_{m-1} & Item_m \\ t_1 & 0 & 1 & \dots & 1 & 0 \\ t_2 & 1 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ t_{n-1} & 1 & 1 & \dots & 1 & 0 \\ t_n & 1 & 1 & \dots & 1 & 1 \end{bmatrix}$$

Table 1: Boolean Matrix

```python
# Read transactions and baskets' items
transactions = pd.read_csv("data/T10I4D100K.dat", header=None, names=["basket"])
transactions.index.names = ['transaction']
transactions["basket"] = transactions["basket"].str.split()

# Define variables
n_transactions = transactions.shape[0]  # Get the total number of baskets
support_threshold = 0.01  # 1% of frequency of the singleton in the total set
min_support = n_transactions * support_threshold  # Get the minimum amount of basket needed as threshold

print("Threshold", min_support)
frequent_items, boolean_matrix = item_counts(transactions, min_support)  # Obtain frequent items to consider
save_items(boolean_matrix, "boolean_matrix")
print("Amount of frequent items", len(frequent_items))
save_items(frequent_items, "1")  # Save frequent singletons
```

```python
def item_counts(data, min_support):
    transaction_set = []
    # Flatten items sets
    for row, transaction in tqdm(data.iterrows()):
        transaction_set.extend(transaction[0])
    mlb = MultiLabelBinarizer()
    boolean_matrix = pd.DataFrame(mlb.fit_transform(data["basket"]),  # Boolean matrix for elements in vocabulary
                                  columns=mlb.classes_,
                                  index=data.index)
    # Count the items
    item_cnt = Counter(transaction_set)  # Dictionary of frequency of each item
    # Filter items that are not frequent
    frequent_items = dict(filter(lambda elem: elem[1] >= min_support, item_cnt.items()))

    return frequent_items, boolean_matrix
```

Figure 2: Phase 1 implementation.

3.2 Phase: 2 - Frequency of Items Combinations

After computing the filtered frequent items we decided to create a function that sequentially computes the modules *Construct* and *Filter* from figure 1. For instance, initially, we get all of the filtered frequent items (in our case 375) and compute all of the possible pairs (70,125 candidates). Then we compute the frequency of these candidates upon the original transactions. This is when the proposed one-hot-encoded matrix makes a huge difference. Suppose that we have the candidate pair $(Item_1, Item_m)$; our solution sums the rows of the columns of both items and counts all the rows with sum equals 2, which indicates that both items are part of that transaction. The same would happen for triplets, but we would be looking for the sum equals to 3 and so on. Following, we filter the candidates again based on the threshold $\mathbf{S}$, and only those pairs/triplets that have a frequency bigger than $\mathbf{S}$ continue to the next phase of *Construct* and *Filter*.

Since computing the frequency of the candidates is an asynchronous procedure, we decided to parallelize this task, which made our solution 7 times faster, although this, of course, depends on the amount of cores the execution machine has. Our implementation of phase 2 can be seen in the following figure. It takes into account how "deep" the user wants to go in terms of clustering items (pairs, triplets, quadruplets, ...).

```python
# Analyze up to tuples of k elements
for k in range(2, k_tuple + 1):
    frequent_items = candidate_k_pairs(frequent_items, k, min_support, boolean_matrix)
    save_items(frequent_items, str(k))  # Save frequent k-element combinations
    print(frequent_items)
```

```python
def candidate_k_pairs(frequent_items, combinatory_factor, min_support, boolean_matrix):
    if combinatory_factor == 2:  # First k-element tuple pass
        keys = list(frequent_items.keys())
    else:
        keys = Counter(sum(list(k for k in frequent_items.keys()), ()))  # Dictionary of item frequencies
        # The support of a subset is at least as big as the superset one
        keys = list(dict(filter(lambda elem: elem[1] >= combinatory_factor - 1, keys.items())).keys())

    candidates = list(itertools.combinations(keys, combinatory_factor))  # Get candidate combinations
    pool = mp.Pool(mp.cpu_count())
    start = time()
    # Parallelize the search of valid candidates
    result = pool.starmap(check_candidate, [(c, min_support, boolean_matrix) for c in candidates])
    end = time()
    print("Time required for parallelization ", end - start)
    pool.close()

    return {k: v for d in result for k, v in d.items()}  # Merge result dictionaries
```

```python
def check_candidate(candidate, min_support, boolean_matrix):
    candidate_items = {}
    compare_columns = len(np.where(boolean_matrix[list(candidate)].sum(axis=1) == len(candidate))[0])
    if compare_columns >= min_support:  # Save candidates that surpassed the threshold
        candidate_items[candidate] = compare_columns
    return candidate_items
```

Figure 3: Phase 2 implementation.

The outcomes of the second and third filtering are:

- Filtered frequent pairs: ('368', '682'): 1193 times, ('368', '829'): 1194 times, ('825', '39'): 1187 times, ('825', '704'): 1102 times, ('39', '704'): 1107 times, ('227', '390'): 1049 times, ('390', '722'): 1042 times, ('217', '346'): 1336 times, ('789', '829'): 1194 times.
- Filtered frequent triplet: ('825', '39', '704'): 1035 times.

3.3 Phase: 3 - Association Rule between Frequent Items.

Once the frequent items are obtained after all the filtering stages, it is time to build the association rules.

In order to do so, first, all possible subsets of the size of each frequent itemset minus one are computed. Then, for each one of those combinations, the disjunctive union is computed, i.e. the element that belongs to the bigger set and not to the smaller one is collected. Finally, the frequency of the bigger set appearing in those baskets that also contain the smaller set is computed: $support((A,B) \cup C)$ / $(support((A,B))$. This returns the probability (*confidence*) of a client buying the elements contained in the set (A, B) and also buy the item C. In our implementation, (figure 4), we assumed confidence higher than 0.6 would determine that a customer buy a pair of items would likely buy a specific third one.

```python
finally:
    association_rules(list(frequent_items.keys()), boolean_matrix)  # Compute association rules for frequent items
```

```python
def association_rules(items, boolean_matrix):
    associations = []
    combinations = []
    for item in items:
        combinations.append(list(itertools.combinations(item, len(item) - 1)))

    for i in range(len(items)):
        for pair in combinations[i]:
            associated2 = set(items[i]) - set(pair)
            num = len(np.where(boolean_matrix[list(items[i])].sum(axis=1) == len(items[i]))[0])
            denom = len(np.where(boolean_matrix[list(pair)].sum(axis=1) == len(pair))[0])
            confidence = num / denom
            if confidence >= CONFIDENCE:
                associations.append(str(pair)+"--->"+str(associated2)+" = "+str(confidence))

    with open(PATH+"/associations.txt", 'w') as f:
        for i in associations:
            f.write(i+"\n")
```

Figure 4: Phase 3 implementation.

For our implementation, the associations of the items of tuples that surpassed the confidence level are:

- ('704') $\rightarrow$ {'825'} = 0.6142697881828316
- ('704') $\rightarrow$ {'39'} = 0.617056856187291

And the association of the combinations of pairs with and item of the triplets that surpassed the confidence level are:

- ('825', '39') $\rightarrow$ {'704'} = 0.8719460825610783
- ('825', '704') $\rightarrow$ {'39'} = 0.9392014519056261
- ('39', '704') $\rightarrow$ {'825'} = 0.9349593495934959

4 How to Run

In order to execute the code, having the dataset inside a `data` folder, it is just needed to execute the `Apriori.py` class via `python3 Apriori.py` command.

This class will compute the frequent items per each stage and the boolean matrix, and will save them into pickle files. Besides, the associations for the last frequent items set will be saved into a `.txt` file. All these files will be saved into a folder called `results` automatically by the program.