Information Systems assignment 3

Koen Bolhuis (s3167895)
Germán Calcedo (s5381541)

December 20, 2022

The full Jupyter notebook for this assignment can be found at `https://github.com/gcalcedo/is-labs/blob/master/assignment-3/assignment3.ipynb`.

1 Loading the CSV file

First, we load the transactions' data. For that, we simply store each transaction in a list. We also retrieve the item names in a separate list, which we will index later on.

```
1  import csv, collections
2
3  with open('myDataFile.csv', 'r') as csvfile:
4      csvreader = csv.reader(csvfile)
5
6      items = next(csvreader)
7      transactions = [
8          set(items[i] for i, v in enumerate(row) if v == 't')
9          for row in csvreader
10     ]
11
12 N_TRANSACTIONS = len(transactions)
13
14 print('Transactions:', N_TRANSACTIONS)
```

To ease readability, we declare constants for both the required minimum support and confidence values. These are the parameters of the Apriori algorithm.

```
1  MIN_SUPPORT = 0.005
2  MIN_CONFIDENCE = 0.6
```

2 Apriori Algorithm

2.1 Generation of the frequent items: L_1

The Apriori algorithm runs on a given set of frequent items, which can be thought of as the first layer of the algorithm. The algorithm subsequently adds layers based on the one immediately before. To obtain this set, we first count the number of appearances of each item in the list of transactions.

```
L = []

histogram = collections.defaultdict(int)
for transaction in transactions:
    for item in transaction:
        histogram[(item,)] += 1
```

Then, we compute L_1 only including the items that exceed or equal the minimum support. `L` contains all the obtained layers.

```
supports = {}

layer_1 = set()
for itemset, count in histogram.items():
    support = count / N_TRANSACTIONS
    if support >= MIN_SUPPORT:
        layer_1.add(itemset)
        supports[itemset] = support

L.append(layer_1)
```

2.2 Generation of subsequent layers: $L_2, L_3, ..., L_k$

We now run a loop until we reach a layer L_k with no elements. To obtain new layers we compute a list of candidate item sets. To do so, we perform a **self-join** on the previous layer. For L_2, we compute the **self-join** of L_1.

```
k = 0
while len(L[k]) != 0:
    k += 1

    candidates = []
    for itemset1 in L[k-1]:
        for itemset2 in L[k-1]:
            if itemset1[:-1] != itemset2[:-1] or itemset1[-1] >=
               itemset2[-1]:
                continue

            candidate = itemset1 + (itemset2[-1],)
```

Then we **prune** item sets that contain at least one subset (item set with all elements of the parent but one) which is not present in the previous layer.

```
            include = True

            for subset in itertools.combinations(candidate, k):
                if subset not in L[k-1]:
                    include = False
                    break

            if include:
                candidates.append(candidate)
```

Finally, in a similar manner to the base layer, we compute the frequency of each item set in the transactions and we only include in the new layer those who meet the required minimum support.

```
    histogram = collections.defaultdict(int)
    for transaction in transactions:
        for candidate in candidates:
            if set(candidate).issubset(transaction):
                histogram[candidate] += 1

    new_layer = set()
    for itemset, count in histogram.items():
        support = count / N_TRANSACTIONS
        if support >= MIN_SUPPORT:
            new_layer.add(itemset)
            supports[itemset] = support

    L.append(new_layer)
```

We can visualize the result of the algorithm in the following way.

```
for k, layer in enumerate(L):
    print(f'L({k+1}):', len(layer))
```

When running on the given data set, our implementation produces 4 layers.

- $\mathbf{L_1}$: 120
- $\mathbf{L_2}$: 605
- $\mathbf{L_3}$: 264
- $\mathbf{L_4}$: 12

3 Rule generation

As confidence is non-increasing as the number of items in the rule consequent grows, we can generate the rules in such a way that enables us to avoid checking unnecessary rules. To do so, for a given itemset, we recursively produce all of the rules whose consequent is a direct successor of the initial rule. If we perform this procedure starting with the largest consequent possible, we can stop the generation for a given rule once the minimum confidence is not met, as we can ensure that all successors of that rule will also not meet the minimum confidence.

For example, if we start with the rule $\mathsf{ABC} \rightarrow \mathsf{D}$, we can stop the generation as soon as we encounter a rule which does not meet the minimum confidence. In figure 1, rules $\mathsf{AC} \rightarrow \mathsf{BD}$ and $\mathsf{BC} \rightarrow \mathsf{AD}$ do not have the required confidence, and thus we can prune the generation of their successor rules.

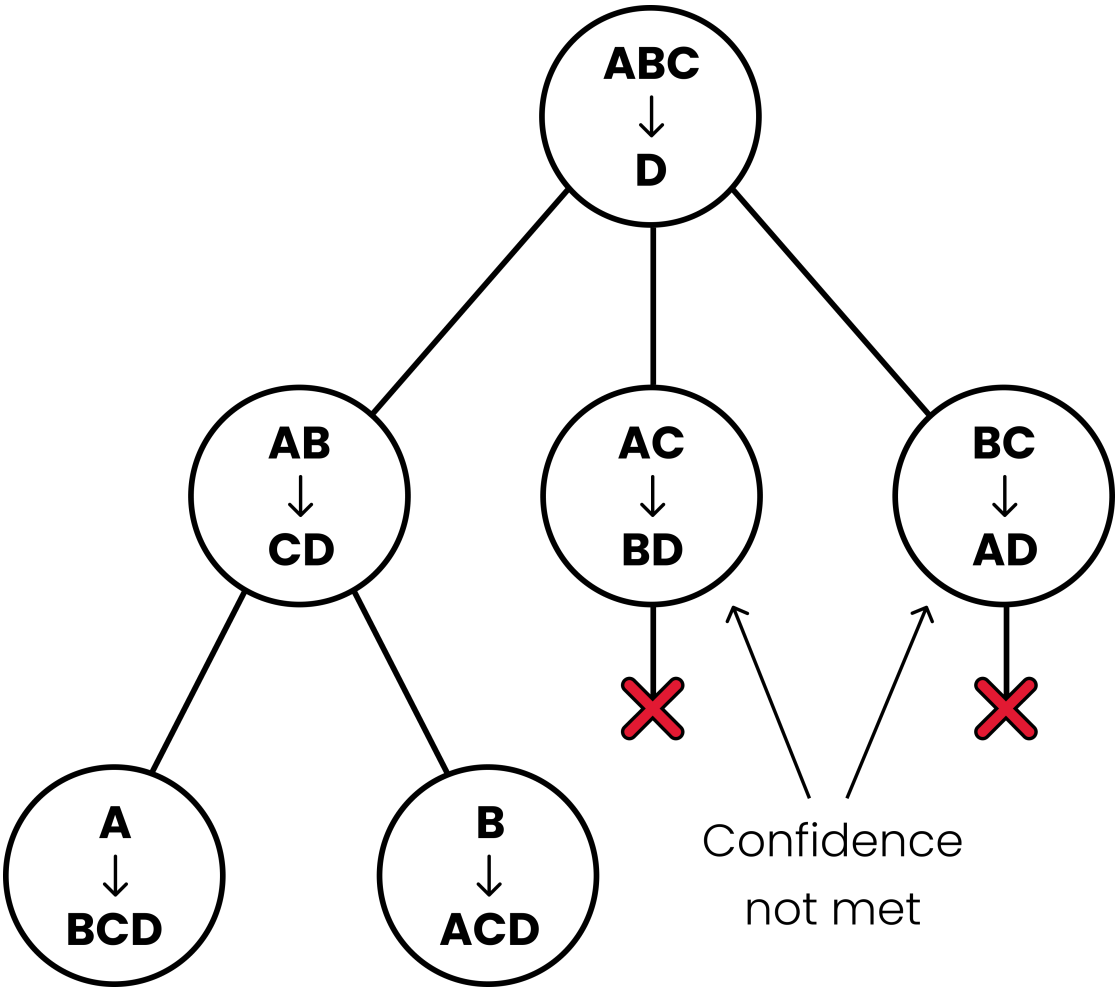

Figure 1: Recursive rule generation.

The implementation of this technique is achieve through the following piece of code.

```
rules = set()

def subsets(size, itemset):
    if size == 0:
        return

    for subset in itertools.combinations(itemset, size):
        confidence = supports[itemset] / supports[subset]

        if confidence >= MIN_CONFIDENCE:
            rules.add((subset, frozenset(itemset) - frozenset(subset)
                , confidence))

            subsets(size - 1, subset)

for k, layer in zip(range(len(L), -1, -1), reversed(L)):
    for itemset in layer:
        subsets(k - 1, itemset)

print('Rules:', len(rules))
for rule in rules:
    print(', '.join(rule[0]), '=>', ', '.join(rule[1]), f'(confidence
        ={rule[2]:.3f})')
```

3.1 Results

After rule generation, our implementation finds 22 rules. The rules and associated confidence levels can be found in table 1, as well as in the Jupyter notebook output.

Rule	Confidence
curd, tropical_fruit ⇒ whole_milk	0.634
domestic_eggs, margarine ⇒ whole_milk	0.622
pip_fruit, whipped_sour_cream ⇒ whole_milk	0.648
butter, yogurt ⇒ whole_milk	0.639
onions, root_vegetables ⇒ other_vegetables	0.602
butter, tropical_fruit ⇒ whole_milk	0.622
fruit_vegetable_juice, other_vegetables, yogurt ⇒ whole_milk	0.617
butter, root_vegetables ⇒ whole_milk	0.638
other_vegetables, pip_fruit, yogurt ⇒ whole_milk	0.625
domestic_eggs, tropical_fruit ⇒ whole_milk	0.607
butter, domestic_eggs ⇒ whole_milk	0.621
citrus_fruit, root_vegetables, whole_milk ⇒ other_vegetables	0.633
other_vegetables, tropical_fruit, yogurt ⇒ whole_milk	0.620
pip_fruit, root_vegetables, whole_milk ⇒ other_vegetables	0.614
butter, whipped_sour_cream ⇒ whole_milk	0.660
other_vegetables, root_vegetables, yogurt ⇒ whole_milk	0.606
root_vegetables, tropical_fruit, yogurt ⇒ whole_milk	0.700
domestic_eggs, pip_fruit ⇒ whole_milk	0.624
other_vegetables, root_vegetables, whipped_sour_cream ⇒ whole_milk	0.607
pip_fruit, whipped_sour_cream ⇒ other_vegetables	0.604
bottled_water, butter ⇒ whole_milk	0.602
other_vegetables, pip_fruit, root_vegetables ⇒ whole_milk	0.675

Table 1: Generated association rules.