Frequent Itemsets, Association Rules

16. října 2012

Introduction

The aim of this tutorial is to use algorithm apriori for mining frequent itemsets from data and subsequently from them association rules.

1 Data

1. Load file "marketBasket.mat"in Matlab (it contains transaction database of market baskets of customers of a supermarket). Variable "tranDb"is transaction database Boolean matrix form. Every row is a transaction - one market basket of one customer. Every column is one possible item in the market basket. The description of these items is in variable "info".

2 Frequent Itemsets

2. **First, complete the main loop of the algorithm apriori in the function *my_apriori*:**

```
function [frequent_itemsets] = my_apriori(database, min_frequency)
candidates = {};
for i = 1:size(database,2)
    candidates{i} = [i];
end
candidates = prune_patterns(candidates, database, min_frequency);
frequent_itemsets = candidates;

% Here, you should fill in the code of the apriori algorithm
% (see slides of the lecture: "APRIORI algorithm")
while ...
```

```
end
```

On the input of the function *my_apriori* there is a transaction database in Boolean form (*database*) and minimum support (*min_frequency*). On the output of the function there are frequent itemsets. Itemsets are represented as vectors of numbers. For example `itemset = [1 3 5];` is a set containing items 1, 3, 5, i.e. items represented in transaction database by columns 1, 3 and 5. Since we can have itemsets of different length, we will use Matlab data structure *c*ell array to store them (you can read about *c*ell arrays at the end of this document or in Matlab help). An example of storing two itemsets in a *c*ell array is shown here:

```
a = {}; a{1} = [1 3 5]; a{2} = [1 5 7];
```

There are functions *apriori_gen* and *prune_itemsets* in file *my_apriori.m* that you can use for implementation. Function *apriori_gen* generates new candidates of itemsets. It corresponds to the function *Apriori-Gen* from lecture slides. Function *prune_itemsets* removes from a cell array of itemsets those itemsets, which are not frequent (minimum support is an input parameter of this function).

Run function *my_apriori* for minimum support 0.03. If you did everything correctly, you should get 305 frequent itemsets.

3. Function *apriori_gen* works correctly, nevertheless, it does not contain the step in which the itemsets which have an infrequent subset are removed (see lectures). For example if we have itemset `itemset = [1 3 5 7 8];` and we know that set `[1 5 7 8]` is not a frequent itemset, then `[1 3 5 7 8]` cannot be frequent and we can remove it from the list of candidate itemsets. We can do this without computing the support explicitly, because this could be computationally expensive in case of big databases. **Implement this pruning method into the function *apriori_gen* and check if you can improve runtime. If you did not succeed, explain why.**

3 Association Rules

4. **Implement function *associationRules.m*, which will generate association rules from frequent itemsets.** Input of the function *associationRules.m*: list of frequent itemsets from the output of function *my_apriori*, minimum confidence and transaction database. There will be association rules on the output in cell array form. In the first column

there will be the vector of antecedents, in the second column the vector of succedents, in the third column the corresponding support and in the fourth the corresponding confidence.

There is function *newGenerationRules* that you can use for implementation. Function creates "new"association rules from "old"by moving items from antecedents to consequents (always one item). This function can be used for iterative generation of association rules. It holds that it is not possible to increase confidence by moving items from antecedent to consequent (i.e. we can use pruning similarly as we did it with minimum support in the case of frequent itemsets).

Run function *asociationRules.m* for minimum confidence 0.7. Print association rules using function *printRules*.

You should get a set of rules such that the rule with the highest confidence will be: (Hot Dog Buns and Sweet Relish) → (Hot Dogs), Support = 0.03, Confidence = 0.84, and the rule with the lowest confidence will be: (White Bread and 2pct. Milk) → (Eggs), Support = 0.04, Confidence = 0.70.

3.1 Some useful tip for work with cell-arrays

Cell arrays differ from matrices in such way that they can contain not only numbers, but also text strings or structures or other nested arrays. We can create an empty array as `a = {};`, which is very similar to the way we create empty vectors/matrices in Matlab.

We access the entries of an array as follows: `a = {'a', 'b', 'c'}; x = a{1}; % x = 'a'`. Notice that unlike for matrices we need to use curly brackets.

Sometimes you may need to concatenate arrays. For example:

```
a = {'a','b'}; b = {'c','d'}; merged = [a b];.
```

If we used parentheses instead of curly brackets, i.e. `merged = {a b};`, then we would not get an array with 4 elements, but an array containing two nested arrays.