TI2736-C Datamining
Assignment 5: Frequent Itemsets

Delft University of Technology, February–February 2016

Thomas Abeel, Marcel Reinders
Zekeriya Erkin, Julian Kooij
Chantal Olieman, Gijs Weterings, Alex Salazar

Pattern Recognition and Bioinformatics Group

TUDelft

5 Frequent Itemsets

In the following exercises you will work on implementing algorithms to detect frequent itemsets from a set of baskets using the A-Priori algorithm. In addition we will be adding efficiency to the A-Priori algorithm using the PCY algorithm. Finally, we will be using the MapReduce framework to parallelize the algorithm (optional).

Exercise 5.1. A-Priori algorithm

The A-Priori algorithm consists of three phases that are iterated until some number of frequent itemsets of some size have been found. The steps are described below:

1. Construct a set of candidate itemsets C_k.
2. Go through the data and construct for each basket subsets of size k. For each of these subsets, increment their support value if that subset exists in C_k.
3. Filter the set of candidate itemsets to get the set of truly frequent itemsets L_k. That is, check if their support value is equal to or larger than the support threshold.
4. Go to step 1 for $k = k + 1$, until you found frequent itemsets for the size that you requested.

In the `APriori.java` file you will find the framework of the A-Priori algorithm. Some pieces are left out however, which we will implement one by one.

Step 1. Implement the functionality of the `APriori.constructCandidates` method. This method performs the first step of the process, constructing C_k with all candidate itemsets of size k, given the set L_{k-1} of filtered candidate itemsets of size $k - 1$. For the initial case $k = 1$, where no filtered candidates set is present yet, it returns all sets of size 1. For larger k, it should check each union of all possible pairs of itemset in L_{k-1}. If the size of a union is k, then this union is a candidate itemset. Note that the size of the union could also be larger than k, in which case it is not a candidate.

Note: This very often creates more candidate itemsets than necessary, but for the purpose of this exercise, it will suffice.

Hint: In order to walk through the `filteredCandidates` you should convert this `Set` into a datastructure that allows you to do so.

Step 2. Implement the functionality of the `APriori.countCandidates` method. This method performs the second step of the process.

Hint: For creating subsets of size k, you may use the `APriori.getSubsets` method.

Step 3. Implement the third step in `APriori.filterCandidates`.

Step 4. Implement the `APriori.getFrequentSets` method. This method implements the full process by combining the previously created methods. For each size from 1 to k, it should construct candidate itemsets, count these itemsets and filter them.

Note: On the last iteration, no candidate sets need to be computed.

Step 5. In `main.java`, create a new `APriori` object and add the following baskets to this object:

“Cat and dog bites”
“Yahoo news claims a cat mated with a dog and produced viable offspring”
“Cat killer likely is a big dog”
“Professional free advice on dog training puppy training”
“Cat and kitten training and behavior”
“Dog & Cat provides dog training in Eugene Oregon”
“Dog and cat is a slang term used by police officers for a male female relationship”
“Shop for your show dog grooming and pet supplies”

Hint: You can copy these sentences from `input_example/basket.txt`.

Set the support threshold to 3 and run the algorithm on the given set of baskets. Check that the frequent singletons are:

```
[[training], [cat], [a], [dog], [and]].
```

Check that the candidate doubletons are:

```
[[training, and], [training, cat], [training, a], [training, dog],
[cat, a], [cat, dog], [cat, and], [a, dog], [a, and], [dog, and]].
```

Question 5.1. What are the frequent doubletons?

Question 5.2. If we want to compute frequent itemsets of size k, how many passes through the data do we need to do using the A-Priori algorithm?

Question 5.3. An alternative would be to read through the baskets and immediately construct subsets of size k and count how many times each occurred, thereby avoiding calculating the frequent itemsets of size 1 to $k-1$. Why is this not feasible for larger datasets?

Exercise 5.2. PCY algorithm

Next we will be making a small adjustment to the A-Priori algorithm, which leads to the PCY algorithm. `PCY.java` extends the A-Priori algorithm of the previous exercise and it overloads two of its methods; the `constructCandidates` method and the `countCandidates` method. The PCY algorithm affects the choosing of candidate pairs as frequent itemsets, that is it affects C_2.

Step 1. Complete the implementation of the `countCandidates` method. The implementation is very similar to the implementation in the `APriori` class. However, when iterating over the data during $k = 1$, you should also generate subsets of size $k + 1 = 2$, hash these subsets and increment the value in the bucket array to which they hash to.

Step 2. Next we will be implementing the `constructCandidates` method for the `PCY` class. Again, this implementation is very similar to the implementation in the `APriori` class. However for $k = 2$, before adding an itemset to the set of candidates, also test that the itemset hashes to a frequent bucket (i.e. a bucket with a count of at least `supportThreshold`). If this is not the case, the itemset should be skipped.

Step 3. In `main.java`, replace the previously created `APriori` object with a `PCY` object. Set the support threshold again to 3 and the bucket size to 256.

Question 3.1. Compared to the A-Priori algorithm, what is the difference in number of candidate sets that the algorithm tests?

Question 3.2. What is the advantage of the PCY algorithm over the A-Priori algorithm?

Question 3.3. What is the influence of the buckets size? For example, what would happen if the bucket size would be too low?

Exercise 5.3. Optional: SON algorithm

Note: This assignment is **not** required for the practicum, and will **not** be checked at the practicum sessions. It is only intended as an advanced exercise for those who want to test their skills on a more realistic problem.

We will be using the `MapReduce` framework to parallelize the A-Priori algorithm. For this exercise you are expected to set up your own Hadoop installation. Once installed, check that all Hadoop nodes can be started (`start-all.sh` for Linux/OSX systems). You should then be set up to use the code supplied.

As is written in the book, the functions work as follows:

1. **First Map Function:** Take the assigned subset of the baskets and find the itemsets frequent in the subset using the A-Priori algorithm. Lower the support threshold from s to ps if each Map task gets fraction p of the total input file.
2. **First Reduce function:** Each Reduce task is assigned a set of keys, which are itemsets. The value is ignored, and the Reduce task simply produces those keys (itemsets) that appear one or more times.
3. **Second Map Function:** The Map tasks for the second Map function take all the output from the first Reduce Function (the candidate itemsets) and a portion of the input data file. Each Map task counts the number of occurrences of each of the candidate itemsets among the baskets in the portion of the dataset that it was assigned. The output is a set of key-value pairs (C, v), where C is one of the candidate sets and v is the support for that itemset among the baskets that were input to this Map task.
4. **Second Reduce Function:** The Reduce tasks take the itemsets they are given as keys and sum the associated values. The result is the total support for each of the itemsets that the Reduce task was assigned to handle.

Note: For this exercise, the supplied project contains a run configuration for Eclipse. If you want to work in a different IDE you might have to do some work to get it running. Also, MapReduce does not overwrite existing files, so in between each test you should remove the output folders.

Step 1. In `SON.java`, implement the first Map method. This method should load all text into the `APriori` object. Once all text is received (can be checked with `context.getProgress()`, it should compute the frequent pairs and publish these to the reducers.

Step 2. Implement the first Reduce function.

Step 3. Implement the second Map function. This function should create pairs of each basket it receives and increment the support value in case this pair is also a candidate pair. Again, if all data has been processed, this function should publish the support value of all pairs to the reducers.

Step 4. Implement the second Reduce function. This function receives some pair from one or multiple Map methods and sums up their computed support values.

Step 5. Run the `SON Example` run configuration, check the file `second_output/part-r-00000` and verify that the result is equal to that of the regular

A-Priori algorithm.

Step 6. Inspect the `input_basket/retail.dat` file. This file contains information on purchases from a Belgium retail store. Each number represents a certain item in their store. Run the `SON Shopping Basket` run configuration. Which item numbers are apparently frequently bought together in their store?