HW2 - Discovery of Frequent Itemsets

Tianxiao Zhao <tzh@kth.se>
Yantian You <yantian@kth.se>

1. Build and run

To run our program, change the variable "*data_path"* to the path of the input documents, and then run the Main class (including main function). The Main class will automatically execute testing function and print out the results.

2. Solution

Our code contains the following six classes: *Main*, *DataReader*, *FirstPass*, *BetweenPasses*, *SecondPass* and *CreateMap*. Each of them serves as a functional module of the A-Priori algorithm. Figure 1 below illustrates the basic workflow of this algorithm.

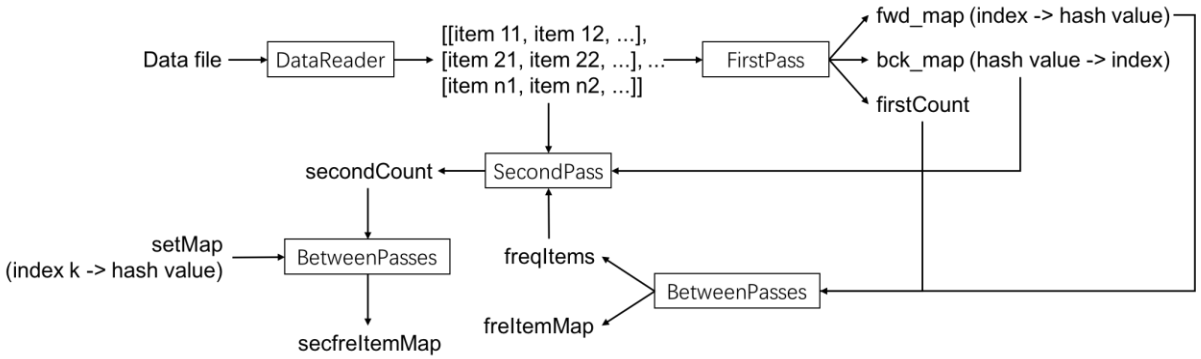

Figure 1: workflow diagram of the A-Priori algorithm

To be exact,

- *Main* class serves as an entry point of the whole algorithm. It includes a *main* function which links different modules together and walks through the whole process of finding frequent items;
- *DataReader* class has two methods - *Read* scans the data file line by line and stores data in a nested *ArrayList<ArrayList<Integer>>*. Each element in the inner *ArrayList* denotes an item in a given basket, while each element in the outer *ArrayList* denotes a basket in the basket set; *GetUnique* returns an ordered set which contains all the unique items in the basket set;
- In the *FirstPass* class, we create a forward map (key: index -> value: hash value of items) and a backward map (key: hash value -> value: index of items), and return them by method *GetFwdMap* and *GetBckMap*. Method *FirstCount* counts the occurrences of items in the basket set and returns a integer array as an output;
- *BetweenPasses* class has two different constructors and they respectively post-process the outputs (thresholded by a predefined support s) from *FirstPass* and

SecondPass, and return frequent singletons and pairs separately; Plus, we also create a frequent-items table which converts the index of items in such a way: if the item is not frequent, the corresponding entry in the frequent-items table is zero while that is set to an unique integer in the range 1 to m if frequent. This table (in form of an integer array) is returned by method *GetFreItemTable*. Also another two maps *freItemMap* (key: hash value of items -> value: counts of occurrences of frequent singletons) and *secfreItemMap* (key: hash values of item-pairs -> value: counts of occurrences of frequent item-pairs) are created and returned by method *GetFreItemMap* and *GetSecFreItemMap*;

- *SecondPass* only contains method *Count* which counts the occurrences of frequent item pairs and returns that in form of an integer array. Note here to save storage space, we use a one-dimensional triangular array *itemPairs[k]* to store counts instead of a two-dimensional array *itemPairs[i, j]*, with 1 <= i < j <= n. The result of this layout is that the pairs are stored in lexicographic order, see Table 1. The relation between *k* and *{i, j}* can be derived as

$$k = 0.5*(i - 1)*(2n - i) + j - i - 1$$

Table 1: frequent item pairs in lexicographic order

k	0	1	...	n - 2	n - 1	n	...	2n - 4	2n - 3	...	3n -7	...	0.5n(n - 1) - 1
{i, j}	{1, 2}	{1, 3}	...	{1, n}	{2, 3}	{2, 4}	...	{2, n}	{3, 4}	...	{3, n}	...	{n - 1, n}

- *CreateMap* class serves as a help class that creates a map between the index *k* in *secondCount* and the hash values of the item pairs. This map is returned by method *GetMap*. To get this map, another help function *getArrayIndex* is used to obtain the index of item given a value in the frequent item table;

3. Results

The dataset we used to test our program is “T10I4D100K.dat”, which contains 10000 baskets and 870 individual items. Each item is a positive number. Here we showed the top five baskets in the dataset:

Table 2: Examples of baskets in basket set

Basket 1	25 52 164 240 274 328 368 448 538 561 630 687 730 775 825 834
Basket 2	39 120 124 205 401 581 704 814 825 834
Basket 3	35 249 674 712 733 759 854 950
Basket 4	39 422 449 704 825 857 895 937 954 964
Basket 5	15 229 262 283 294 352 381 708 738 766 853 883 966 978
…	…

We have tried several supports to see how different support will affect the results.This results shows as below:

Table 3: Number of frequent items with respect to support

Support	0.030	0.025	0.020	0.015	0.010
Number of frequent itemSet(1-tuple)	60	107	155	237	375
Number of Candidates (2-tuples)	1770	5671	11935	27966	70125
Number of frequent itemSet(2-tuples)	0	0	0	0	8
Number of frequent itemSet3-tuples)	0	0	0	0	1

From this table, we can find out that only when support is below or equal to 0.01, we can achieve frequent item set consists of two or three items. Otherwise, only 1-tuple frequent item set can be achieved. When support is 0.01, we got the following 2-tuples frequent item sets and 3-tuples frequent item set :

Table 4: Frequent item pairs and their corresponding frequency

Item Set	Frequency (Support*Number of baskets)
[39, 825]	1187
[39, 704]	1107
[217, 346]	1336
[227, 390]	1049
[368, 682]	1193
[368, 829]	1194

[390, 722]	1042
[704, 825]	1102
[789, 829]	1194
[704, 39, 825]	1035

4. Conclusion

Our implementation of the A-Priori algorithm could successfully find frequent itemsets with an adjustable size. We also play around with the support value *s* and investigate its effect on the selected frequent itemsets. The next move for us is to find the association rules between frequent itemsets given a predefined support *s* and confidence *c*.