**Chapter Six**

**Implementation**

**6.1 : Why Use Java**

Java is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform doesn’t need to be recompiled to run on another. Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture. [9]

**6.2 : General View**

In this implementation we use OpenJDK (Java 7 SE), the official Java reference implementation that is licensed under the GPL , and Linux Ubuntu 12.04 LTS OS environment. For IDE we use Eclipse platform to generate java project called “Association” and this project that implementing Apriori algorithm is divided into three *.java* source files:

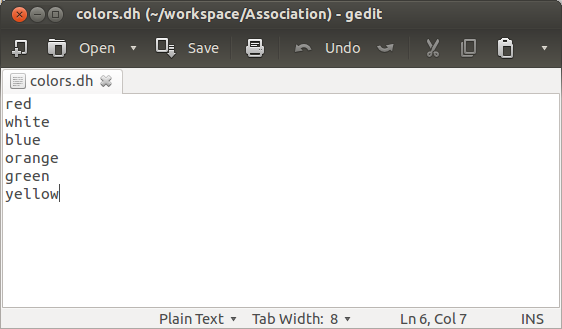
* apriori.java
* Pineapple.java
* TabledString.java

Program main class are in the file apriori.java in name ***apriori*** and it is contain the main( ) method, to run the program, end-user need three arguments must passed to it via command line: min\_support, min\_confidence and dataset\_file.ds .

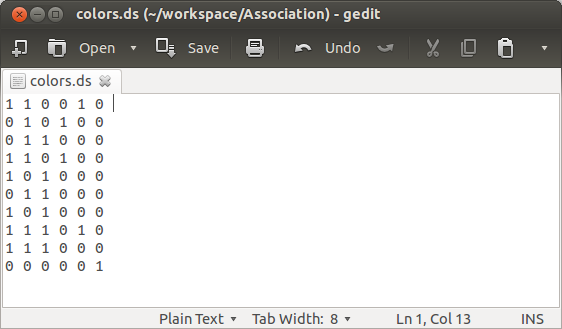
min\_support is an integer number refer to the required minimum support in percentage (relative support) must be for itemsets candidates to be a frequent itemsets, it is turns to an absolute support value by count of transactions in dataset file.

min\_confidence is an integer number refer to the percentage of minimum confidence of an rule to be a strong association rule.

Finally, dataset\_file.ds is an ordinary file of extension *.ds* (dataset) contains transactions data in binary matrix format as Figure 5.5 but columns names that refer to items names of that table are not included in dataset file; So, there is a second file has the same name of dataset file with a different extension *.dh* (dataset header) contain all items names, each one in separated line as showing below in Figure 6.1, the program doesn’t need header file name as a fourth argument because it can conclusion it. Naturally, both of them must be in the same directory.

****

**Figure (6.1): Header file format**

****

**Figure (6.2): Dataset file format**

In case editing dataset file manually, there must be a space at the end of first transaction line, and should be also one empty line after the last transaction. Following these instructions make the program running without unexpected fatal errors could be happens.

These are two files to implement the case study “Purchases of Phone Faceplates” on the program from the example in section 5.5: colors.ds and colors.dh, this program start by type this line on terminal (console) :

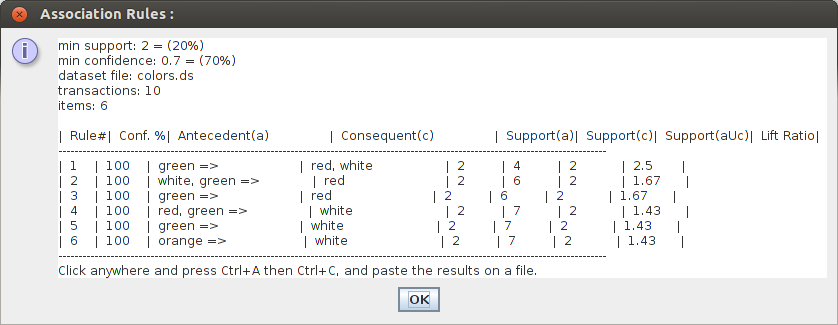
java apriori 20 70 color.ds

It is start running in terminal and print the frequent itemsets from 1-itemsets to k-itemsets , items are named temporally to a,b,c and so on with details, then print strong association rules with confidence if they existing.

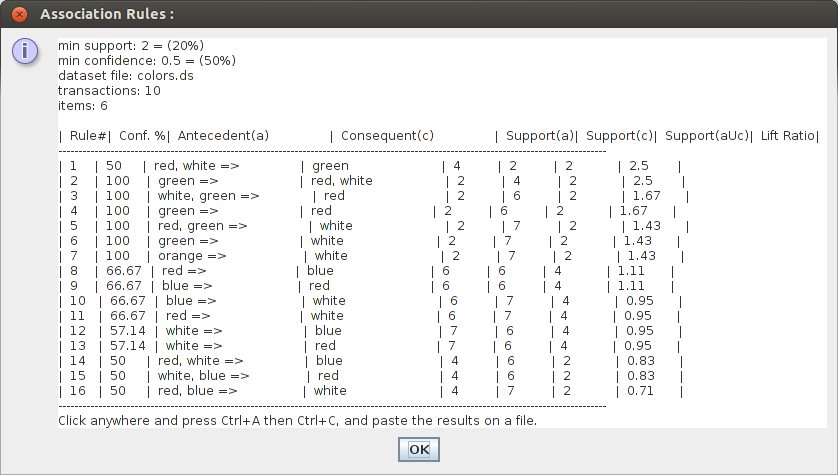


**Figure (6.3): The program running in Windows OS terminal**

After that, a message dialog box titled by “Association Rules:” will appear with a completed report about the input data and number of transactions in dataset file and items, followed by table present the strong association rules with these two sides: if (antecedent) => then (consequent) , the items names like a,b,c are transformed to understandable names to the end-user from the header file, rules are printed with its details and ordered in descend way by the lift ratio. Clicking on OK button will terminate the program.

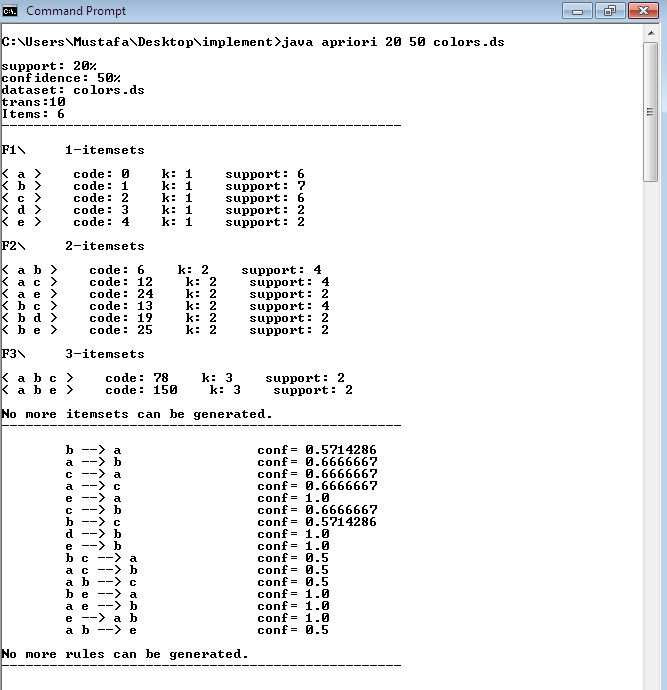


**Figure (6.4): Message dialog box titled by “Association Rules:” with report**



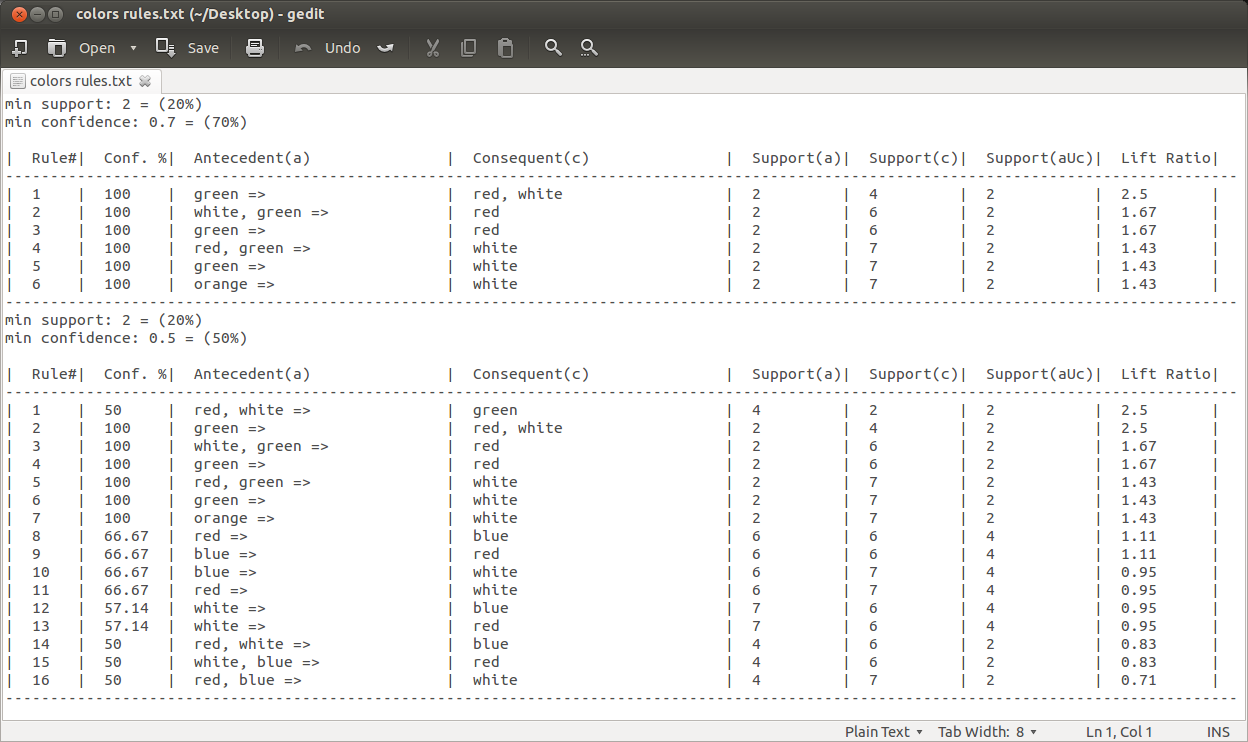
**Figure (6.5): Same case study report but with less confidence 50%**

The report table may it is look unorganized, but in fact it is not. As it written at last line, copy all data in this window to any file will prove that and make the end-user able to save the results and/or print it on a paper.



**Figure (6.6): Same case study terminal but with less confidence 50%**

Figure 6.7 in next page shows the two previous reports saved in an ordinary text file.



**Figure (6.7): The report look organized after copy it to a file**

**6.3 : The Implementation Results**

The report table is similar to the XLMiner tool output and it is presented in eight columns (see previous Figure 6.7):

**Rule#** : rules sequence.

**Conf. %** : the percentage of rule confidence.

**Antecedent(a)** : the left side of the rule (IF side).

**Consequent(c)** : the right side of the rule (THEN side).

**Support(a)** : count of antecedent itemset appear in dataset.

**Support(c)** : count of consequent itemset appear in dataset.

**Support(aUc)** : count of both rule itemsets sides appear together in dataset.

**Lift Ratio** : the lift ratio of the rule.

These rules are sorted by their lift ratio in descending order, all rules are greater than or equals the minimum confidence which are determined by the end-user, the relationship between the minimum confidence and the generated rules are an inverse relationship, high confidence generate less strong rules, and low confidence generate a lot of strong rules; usually, minimum confidence of 70% are recommended.

But we can’t judge about all the rules that pass the minimum confidence test are indeed strong rules, because one item could presented frequently in the dataset will impact the accuracy of rules confidence. Lift values greater than 1 indicate that the rule is more effective and real to predicting the probability than the relative frequency of the consequent itemset. In this case the IF and THEN parts of that rule are positively associated. Conversely, if the lift ratio is less than 1 the rule is less effective than the confidence value that came from relative frequency across the dataset.

By taking the rule number 1 of first table in Figure 6.7 to explain, it has a confidence of 100% and that means if the green item (rule antecedent) is purchased, then the red and white items (rule consequent) will be also purchased by customer certainly according to this confidence value, this rule has 2.5 as a lift ratio.

**6.4 : Programming Concepts**

There are nine java classes in the program: ***S\_Itemset*** class and its inner ***Symb*** class, ***apriori*** class (the main class) and its inner ***Rule*** class, all these four previous classes are presented in apriori.java file, ***Pineapple*** class and its inner ***Itemset***, ***Port*** and ***Slice*** classes in Pineapple.java file, and ***TabledString*** class in TabledString.java file.

The main class ***apriori*** has all basic methods (functions) for implementation of Apriori algorithm and generate the association rules processes, these methods are static (no object from its class) and works on the objects of the other classes. ***Rule*** is an inner static class for presenting association rule objects that matches the minimum confidence and appears in the final report.

Apriori adopts a hash-tree to store the candidates itemsets.[2] But this implementation have a deferent approach similar to hash-table with some improvement. Instead of storing itemsets in leaf nodes, each itemset will convert to an integer value, and from hashing this value determine which slice will store it; this value called “code”, and this is what ***Pineapple*** class do principally.

Any counting system has a base number and group of symbols coded from zero to base-1, like binary system (0,1), decimal system (0 to 9), hexa system (0 to 15) and so on. Here having the base equal to the count of items and each single item has a unique code (from 0 to count of items -1), in program this base value defined as an integer value called “items”. Consequently if items=6 and an itemset code=150, that means three items in this itemset their codes are 0,1 and 4. See carefully on Figure 6.3 at itemsets codes.

***Pineapple*** class (the structure similar to this kind of fruit) has one crust and a number of slices connected to this crust. In a traditional tree, nodes are connected to one root node, but slices connected to multi indexes of crust (multi roots) because the “crust” defined as array of ***Slice*** with length of “csys” (an integer value abbreviated of counting system that passed to class constructor, this argument is “items”).

Every slice could connected to other cross one direction port from source slice to target slice and not in obverse way, Slices have a ***Port*** pointer called “ports” point to group of their ports or none at all, each port has an integer value called “number” and a ***Slice*** pointer called “path”. Slices also have an ***Itemset*** pointer called “data”. New itemset are stored into a specific slice on its “data” pointer and the slice linked to list in the class called “linker”.

Slices are been generated depended on the “code” (integer value) of the itemset, the “code” will divided on “csys” in loops until the quotient equals zero, this process called *itemset hashing*. Starting from the crust array, in each loop program transfer the remain of division from fractional to integer value, and the value of last loop will determine which S slice to store the itemset on it, let’s called every one of these values V for now.

To determine S ,V0 (first V) point to the slice S0 in crust[V0] index, V1 point to the slice S1 from port number V1 on S0 , V2 point to S2 from port number V2 on S1 and so on until reach slice S. If no slice in index V0, it will point to a new slice. If any of required port number doesn’t exist, new ports point to new slices will be generated.

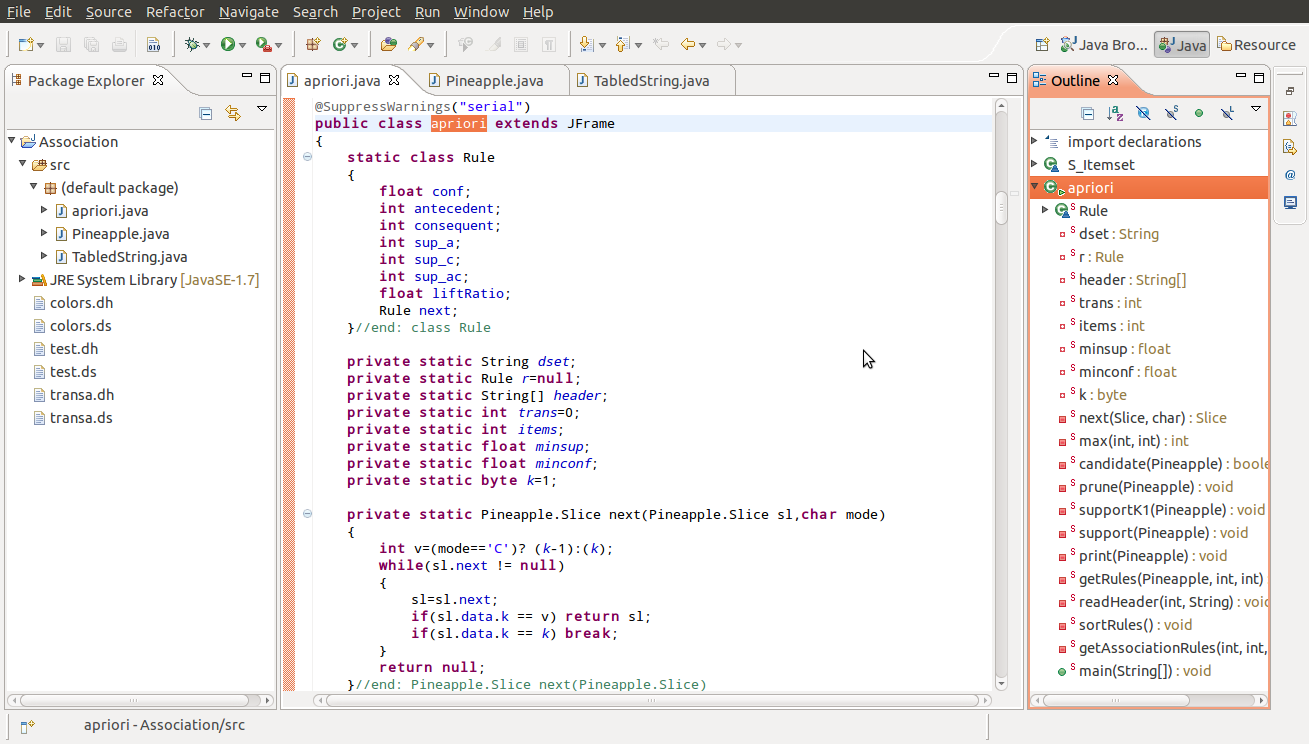
So, ***Pineapple*** class are designed specially for Apriori memory management because this algorithm has to generate a lot of itemsets candidates to storing it and this probably (in case dataset file too big) make the heap memory reach to it limits, this class improves operational performance of this type of implementation such insert, search, fetch and sort operations, but delete operation is not included because it will cause memory management complexity, so and for the purpose of performance the deleting currently done by markup technique, this problem has been mentioned in section 7.2. It is possible to ably this class also on other kind of implementation, like manage data with primary or unique numeric keys, and only in case that keys are integer’s values.

***S\_Itemset*** class used to be itemset separator or dismantling. Basically, it can separate any counting system code value to group of coded symbols by knowing the base system value. The coded symbols are objects of ***Symb*** class connected together in linked list, every object has an integer value called “code”.

For the program, it is generate a number of objects of this class as needed to process itemsets such candidate, prune, mining rules and other operations, also an others objects for *probabilities scan* such pruning an itemset candidate to make sure all sub items are frequent itemsets.

Finally, ***TabledString*** are class for generate reports in tabled format, this class receive groups of java ***String*** class objects as inputs, each object formatted and organized in a table columns, at the end program will receive this table from class as one tabled ***String*** object contains all these inputs and display it to end-user as a total report. Table columns doesn’t defined by names, it defined by it sizes. When a ***TabledString*** object generated, it constructor receive two parameters: an integer type array contains the size of each columns and the space between them (integer array and integer variable), array length refer to the number of table columns.

That was a quick tour around this implementation in general, next will review the methods and the important parts of ***apriori*** class codes.



**Figure (6.8): Eclipse platform and on right showing apriori main class methods**

**6.5 : The Main Class Methods – apriori class**

The full source code presented in the appendix section, it is also available to download implementation project folder from: **http://goo.gl/IFeX3r**

Class ***apriori*** has eight data members: *{lines 98-105}*

**String dset** : contain the name of dataset file.

**Rule r=null** : head pointer to Rule objects linked list.

**String[] header** : array contain the names of items from the header file.

**int trans=0** : number of transaction.

**int items** : number of items.

**float minsup** : absolute minimum support count.

**float minconf** : minimum confidence percentage.

**byte k=1** : current number of items in a candidate itemsets.

6.5.1 : Method - main(String[]) *{lines 563-576}*

Parameters:

* String[] args : program arguments array.

Return value: void

In case length of program arguments array “args” equals 3, the method in line 568 called, else the message in line 572 appear describe what to do to start the program.

6.5.2 : Method - getAssociationRules(int,int,String) *{lines 506-561}*

Parameters:

* int para1 : minimum support count from first program argument.
* int para2 : minimum confidence from second program argument.
* String para3 : dataset file name from third program argument.

Return value: void

In line 511, ***Pineapple*** class reference called “mem” (memory) defined and line 519 open dataset file, line 521 calculate “items” and in line 523 “mem” are point to a new instant of the class. In line 520, “line” object holds one dataset transaction and from line 524 to 536 there is a loop to read all transactions, line 526 count the number of transactions. In line 527, a loop scan each transaction char by char, if a char equals ‘1’, it will stored in “mem” as an itemset with it code “j”, it length “k” and it support as 1. If it already stored before, support will increase by 1.

Line 540 setting “minsup” by transfer minimum support count “para1” from been percentage to an absolute value, next line calling the method *supportK1* to delete any 1-itemsets having support less than “minsup”. Line 546 sorting 1-itemsets by their codes in ascending order and next line call method *print* to print all itemsets details of length “k” whatever k=1 or else.

Most of program time will be in a loop from line 549 to 554, calling method *candidate* that return a boolean value, if true the loop continue else stopped. In loop body, if current itemset length “k” greater than 2, pruning process start by calling the method *prune*.

After the frequent itemsets are generated, these which passed the prune and support tests; time now for the rules. Line 558 setting “minconf” as value of one (75%=0.75) and line 559 calling the method *getRules* to generate the association rules.

6.5.3 : Method - next(Pineapple.Slice,char) *{lines 107-117}*

Parameters:

* Pineapple.Slice sl : an itemset pointer.
* char mode : specify how the method work to determine next itemset.

Return value: Pineapple.Slice

If mode equals ‘C’, this method will return the next slice has itemset of length k-1 to *candidate* method, otherwise (mode equals ‘S’) return next slice has itemset of current k length to *support* method. If no itemset slice is compatible, the method will return none (null value).

6.5.4 : Method - max(int) *{lines 119-128}*

Parameters:

* int code : code of an itemset.

Return value: int

Doing itemset hashing process and return the maximal item code, it is always the last item code of the itemset because 1-itemsets are already sorted in line 546.

6.5.5 : Method - candidate(Pineapple) *{lines 130-170}*

Parameters:

* Pineapple mem : point to the memory object “mem”.

Return value: boolean

Increase current k by 1 and start generate k-itemsets candidates from k-1 frequent itemsets, return true value if there is one k-itemset at least.

6.5.6 : Method - prune(Pineapple) *{lines 172-217}*

Parameters:

* Pineapple mem : point to the memory object “mem”.

Return value: void

Make sure that all k-itemsets subsets (k-1itemsets group) are frequent itemsets.

6.5.7 : Method - supportK1(Pineapple) *{lines 219-230}*

Parameters:

* Pineapple mem : point to the memory object “mem”.

Return value: void

If one of 1-itemsets group support count less than “minsup”, itemset deleted.

6.5.8 : Method - support(Pineapple) *{lines 232-280}*

Parameters:

* Pineapple mem : point to the memory object “mem”.

Return value: void

If one of k-itemsets group support count less than “minsup”, itemset deleted.

6.5.9 : Method - print(Pineapple) *{lines 282-306}*

Parameters:

* Pineapple mem : point to the memory object “mem”.

Return value: void

Method for showing program status while processing by print the frequent itemsets details in terminal after each k level as a background work.

6.5.10 : Method - readHeader( ) *{lines 436-460}*

Parameters: noun

Return value: void

Both of dataset (*.ds*) and header (*.dh*) files have the same name but different extension, lines 438 to 444 conclusion the full name of header file, line 446 set “header” array length to “items” and the rest of method lines store items names in “header” array.

6.5.11 : Method - getRules(Pineapple,int,int) *{lines 308-434}*

Parametrs:

* Pineapple mem : point to the memory object “mem”.
* int para1 : minimum support count from first program argument.
* int para2 : minimum confidence from second program argument.

Return value: void

Biggest method in this implementation, and it is the method which display the final end-user report, it has two sections. First section lines from 320 to 372, scanning all itemsets that length higher than 1 to generate association rules, the loop in line 326 hold one frequent itemset and run probabilities scan on it for left and right sides (antecedent and consequent) of a group of rules are belong to this itemset.

Line 339 calculate the confidence of those rules, lines 340 to 355 testing if the rule “conf” value greater than or equals “minconf” value, it will be added to the list “r” with it details: confidence, the code and support count of both rule sides, support count of the rule itself and lift ratio.

Second section lines from 377 to 432, start sort the rules of the linked list “r” in descending order by their “liftRatio” value, then in line 378 an integer array called “colsize” defined with eight indexes to determine the size of table columns (final report table). Next line define ***TabledString*** object called “report” with two parameters: the array “colsize” and an integer number refer to the space between columns. By using methods of object “report”, program insert columns names and its data fields from all rules in the ordered list “r” to the table. After that from line 425, a ***String*** object called “s” will contain some inputs data (such “para1” & “para2” values), is joined with another string returned from the called method *getString* of object “report”. Line 432 display the results in message dialog box titled by “Association Rules:”.

6.5.12 : Method - sortRules( ) *{lines 462-504}*

Parameters: none

Return value: void

Sorting the rules in linked list “r” in descending order by their “liftRatio” values.

In the appendix section, will review those three *.java* source files marked by lines number.