**Tour of Data Mining Algorithms**

by Armando Navarro and John Sanchez

**Abstract**

blah blah blah blah blah sumary goes here blah blah blah blah blah blah

blah blah blah blah blah sumary goes here blah blah blah blah blah blah

blah blah blah blah blah sumary goes here blah blah blah blah blah blah

1. The Apriori implementation is composed of an Apriori class, a HashTree class, and classes that make up the components of the hash tree, including Node, Item, and ItemSet classes. Before the Apriori class is run, the CLI class in the UI package processes command line arguments, then calls AprioriSession in the Application package, which reads the input files and stores the transactions as a Set of ItemSets. ItemSet objects are self-ordering, and are simply Java’s TreeSet extended to include a frequency count. The Apriori class in the domain package is then called to run the Apriori algorithm, operating entirely on data in memory. The implementation works primarily by making calls to the HashTree class, in order to manipulate the hash trees storing frequent itemsets and their occurrence counts.

The Apriori implementation creates a separate hash tree for 1-itemsets, 2-itemsets, and so on. The first (1-itemset) hash tree is created by scanning the transactions, adding and counting 1-itemsets as it goes. The remaining hash trees are created by self-joining the previously generated tree. The self-join as implemented only joins two itemsets in which the last item differs, but all preceding items are the same, which avoids generating duplicates (recall that items are ordered). The *prune* method takes as argument the previously generated tree, and relies on the *areAllSubsetsFrequent* method of said previous tree for its primary logic.

The majority of the functionality for Apriori is contained in the Node class. It contains methods for the following: adding/counting itemsets; checking to see if it has a bucket and is therefore a bucket node or is otherwise a hash node; converting itself to a hash node from a bucket node; removing itemsets that don’t meet minimum support; and pruning. It also contains an *areAllSubsetsPresent* method that compliments the previously mentioned and similarly named method of the HashTree class. It takes a single ItemSet and either checks its bucket for a match, or hashes the next item and passes it to its appropriate child node if it is a hash node.

Some interesting problems included determining how many child nodes each hash node should be allowed to create. The implementation uses a default value of 3 children per node, but a different value can be passed in from the command line. Similarly, the maximum number of itemsets per bucket node uses a default value of 5, which can also be overwritten with a command line option. Also, there was uncertainty of what to do when an itemset needed to be added to a bucket, the max bucket size had been reached, and there were no more items to hash on. This was resolved by simply surpassing the bucket size limit in this case. As a result, each k-itemset tree has a maximum level of k (with the root being level 0).

Note that the command line interface uses the Apache Commons CLI library to process command line options and generate help messages. The Apache Commons Math library was also used for its efficient *binomialCoefficient* method, so that a node could determine whether its bucket size limit would be breached by adding all possible k-itemsets ultimately generated from its given set.

1. ID3 stuff goes here. Work in progress…