**Tour of Data Mining Algorithms**

Implementation Project by Armando Navarro and John Sanchez

**Abstract**

The following three algorithms have been implemented in Java: Apriori (section 1), ID3 (section 2), and Xmeans (section 3). Section 4 describes the group work breakdown. For Apriori and ID3, familiarity of the algorithms is assumed and therefore the general algorithm descriptions are omitted. The Apriori and ID3 algorithms have most of their functionality in the Node classes of their corresponding tree data structures: hash trees and decision trees, respectively. Apache Commons libraries were used for the command line interface and for an efficient binomial coefficient method. These two implementations take advantage of a three-layer architecture, and operate on data in memory. Apriori makes default assumptions about the number of children nodes per hash node and the maximum size of bucket nodes. A particularly tricky problem of ID3 was determining how to provide each node with the information needed to accurately compute information gain, since only the root node can directly determine the frequencies of particular attribute values, which are needed to determine probabilities. XMEANS STUFF TO COME (IN 86 WORDS OR SO)…

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.

The Apriori implementation was tested on a small data set example from the text book, as well as the Belgium Retail Market data set. For the latter set, the processing time can be quite large depending on the minimum support specified; a minimum support of 1200 or above is recommended for quick results. The output displays each k-itemset tree as it is generated. Note that the two data sets use different delimiters to separate attributes, so the delimiter option should be specified (see User Manual).

Extensions to the algorithm could be made so that it can reliably work on transaction data with non-numerical identifiers. Work has actually already been done towards this goal, as can bee seen by the StringItem class. However, development of this ability is ultimately untested and likely unreliable. Development of this algorithm took many weeks, with constant refactorings while trying to figure out such issues as whether the Node class was better implemented as a family containing HashNode/BucketNode classes, whether the funtionality should be within the Node classes or have them be “dumb” classes with complex logic elsewhere. It was very much an exercise in trial and error, more so than ID3, for example.

1. Similar to the Apriori algorithm, the ID3 class in the domain package is not called until after the CLI and ID3Session classes read the data and store it as an ArrayList of String[] (string arrays). The ID3 implementation works entirely with data in memory. It is composed of the following classes: ID3, InformationGain, DecisionTree, Node, and Attribute. Most of the functionality exists in the Node class. The *run* method of the ID3 class creates the decision tree from the training set, while the *classify* method takes a set of unlabeled tuples and lables them using the previously created decision tree.

The DecisionTree class is simply an extended Node class that handles root creation differently than other nodes. The difference is in the way attribute information is passed. This difference extends to the Information Gain’s overloaded *run* class. Each node is passed a list of data tuples, the index of the class label attribute, general attribute information, and the attribute value that was used to create the given partition of tuples (if any). For the root node, the attribute information is an array of attribute titles. Since the root node is given the entire set of input data tuples, it uses this information to count the amount of times each particular value occurs in the data for each particular attribute. These counts are stored as an array of Attribute instances. Each Attribute instance contains an attribute title, along with every possible value for that attribute, obtained from the original complete data set. This Attribute array is passed to every child Node constructor and InformationGain *run* method from then on. This is necessary because each non-root node is only passed a partition of the data tuples, which may not contain every possible value for the remaining attributes.

The remaining Node and InformationGain methods follow the ID3 algorithm in a straightforward manner. The Node class has methods for the following: determining if the given data tuples are all of the same class (*isHomogenous*); determining if any attributes remain to split the data on; returning a class label based on majority voting; splitting the given tuples into partitions based on the previously determined splitting criterion and passing those partitions to newly created child nodes; and classifying an unlabeled tuple. The InformationGain class has methods for: counting the frequencies of attribute values as mentioned above; calculating the entropy of the given tuples; and calculating the entropy of the given tuples after being split on a particular attribute.

The ID3 implementation was tested on two small training set examples. The correct trees are generated. They were tested on a testing version of the same data set, with class labels removed. The program outputs the generated decision tree, as well as the labeled tuples of the specified unlabeled testing data set. If no testing data set is specified, only the decision tree is generated and outputted.

A further extension could be made to enable the algorithm to work with non-categorical, continous-valued attribues. As implemented, the algorithm will currently treat each value of a numerical attribute equally, resulting in an integer attribute with range 1-100 possibly being split into 100 partitions. This is far from ideal, and could be remedied with further modification. Much was learned from figuring out how to overcome issues such as how to pass attribute data to all nodes for probability calculations, how to handle more than two classes, and how best to eliminate attributes from consideration in later levels of the decision tree.

1. Xmeans stuff goes here

and here and here and here

1. Apriori was written by Armando Navarro, while XMeans was written by John Sanchez. A starting version of Information Gain was written by Sanchez, initially being limited to two classes, requiring the “positive” class to be specified by the user, and being unable to differentiate between identical values between different attributes. Information Gain was rewritten and the remaining ID3 implementation created by Navarro. Presentation slides and final report work for Apriori and ID3 was created by Navarro, with slides and report work for Xmeans by Sanchez.

**References**

Information on Concepts and Algorithms:

Agrawal, Rakesh, and Ramakrishnan Srikant. "Fast algorithms for mining association rules." *Proc. 20th int. conf. very large data bases, VLDB*. Vol. 1215. 1994.

Coffman Jr, Edward G., and James Eve. "File structures using hashing functions." *Communications of the ACM* 13.7 (1970): 427-432.

*Association Analysis: Basic Concepts and Algorithms* from Introduction to Data Mining by Tan, Steinbach, Kumar.

*An intoductory tutorial on kd-trees* by Andrew W. Moore published with Carnegie Mellon University

*Data Mining: Concepts and Techniques*, 3rd Ed, by Jiawei Han.

Data Sets:

*Data Mining: Concepts and Techniques*, 3rd Ed, by Jiawei Han.

*Belgium Retail Market, http://www.cs.rpi.edu/~zaki/Workshops/FIMI/data/*