THE LUCS-KDD IMPLEMENTATIONS OF THE CBA ALGORITHM

Frans Coenen

Department of Computer Science

The University of Liverpool

Thursday 7 April 2005

Updated: Tuesday 31 July 2007, Monday 7 January 2008, Wednesday 3 December 2008, Wednesday 21 January 2010, Saturday 3 Martch 2012.

DISCLAIMER! The following description of the CBA algorithm (Liu et al 1998) is that implemented by the author of this WWW page, i.e. it is not identical to that first produced by Bing Liu, Wynne Hse and Yiming Ma, but certainly displays all the "salient" features of the CBA algorithm.

CONTENTS

1. INTRODUCTION

CBA (Classification Based on Associations) is a Classification Association Rule Mining (CARM) algorithm developed by Bing Liu, Wynne Hsu and Yiming Ma (Liu et al. 1998). CBA operates using a two stage approach to generating a classifier:

1. Generating a complete set of CARs (Classification Association Rules).
2. Prune the set of CARs to produce a classifier.

In Liu et al.'s implementation the first stage is implemented using CBA-RG (CBA-Rule Generator). CBA-RG is a multi-pass Apriori style algorithm. Rules, once generated, are pruned using the "pessimistic error rate based pruning method" described by Quinlan with respect to C4.5 (Quinlan 1992), and place in a rule list `R`. The second stage is implemented using CBA-CB (CBA - Classifier Builder). The operation of CBA is described in more detail in Section 4 below.

The two stage approach is a fairly common approach used by many CARM algorithms, for example the CMAR (Classification based on Multiple Association Rules) algorithm (Li et al. 2001).

1.2. Overview of LUCS-KDD implementation of CBA

The LUCS-KDD implementation of CBA operates in exactly the same manner as described by Liu et al except that the CARs are generated using the Apriori-TFP algorithm (Coenen et al. 2001, Coenen et al. 2004 and Coenen and Leng 2004) and placed directly into the rule list `R`.

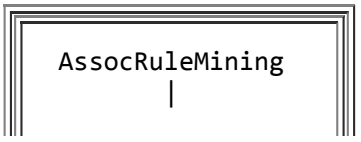
---

2. DOWNLOADING THE SOFTWARE

The LUCS KDD CBA software comprises nine source files. These are provided from this WWW page together with three application classes. The source files are as follows:

1. `AssocRuleMining.java`: Set of general ARM utility methods to allow: (i) data input and input error checking, (ii) data preprocessing, (iii) manipulation of records (e.g. operations such as subset, member, union etc.) and (iv) data and parameter output.
2. `PartialSupportTree.java`: Methods to implement the "Apriori-TFP" algorithm using both the "Partial support" and "Total support" tree data structure (P-tree and T-tree).
3. `PtreeNode.java`: Methods concerned with the structure of Ptree nodes.
4. `PtreeNodeTop.java`: Methods concerned with the structure of the top level of the P-tree which comprises, to allow direct indexing for reasons of efficiency, an array of "top P-tree nodes".
5. `RuleNode.java`: Class for storing binary tree of CARs as appropriate. (Used to be defined as an inner class in AssocRuleMining class.)
6. `TotalSupportTree.java`: Methods to implement the "Apriori-T" algorithm using the "Total support" tree data structure (T-tree).
7. `TtreeNode.java`: Methods concerned with the structure of Ttree nodes.
8. `AprioriTFPclass.java`: Parent class for classification rule generator.
9. `AprioriTFP_CBA.java`: Methods to produce classification rules using CBA algorithm but founded on Apriori-TFP.
10. `AprioriTFP_CARgen`: Methods to produce classification rules using a Apriori-TFP appraoch.

The `PtreeNode`, `PtreeNodeTop` and `TtreeNode` classes are separate to the remaining classes which are arranged in a class hierarchy of the form presented in Figure 1.



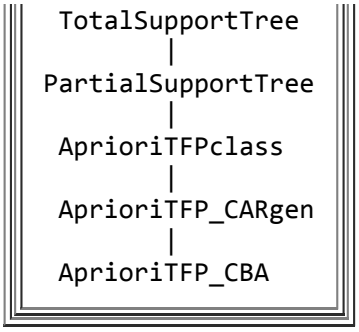


Figure 1: *Class Hierarchy*

The Apriori-TFPC CBA application classes included here are as follows:

1. `ClassCBA_2file_App.java`: Prooduces a single-class clasifier from a given data set given particular support and confidence thresholds as input using the CMAR algorithm. Takes two input files: (i) training set, (ii) test set.
2. `ClassCBA_App.java`: Fundamental LUCS-KDD CBA application using a 50:50 training/test set split.
3. `ClassCBA_App10.java`: LUCS-KDD CBA application using TCV (Ten Cross Validation).

There is also a "tar ball" cba.tgz that can be downloaded that includes all of the above source and application class files. It can be unpacked using `tar -zxf cba.tgz`.

2.1. Compiling

The LUCS-KDD CBA software has been implemented in Java using the Java2 SDK (Software Development Kit) Version 1.4.0, which should therefore make it highly portable. The code does not require any special packages and thus can be compiled using a standard Java compiler:

```
javac *.java
```

2.2. Documentation

The code can be documented using *Java Doc*. First create a directory `Documentation` in which to place the resulting HTML pages and then type:

```
javadoc -d Documentation/ *.java
```

This will produce a hierarchy of WWW pages contained in the `Document` directory.

---

3. RUNNING THE SOFTWARE

When compiled the software can be invoked in the normal manner using the Java interpreter:

```
java APPLICATION_CLASS_FILE_NAME -F FILE_NAME -N NUMBER_OF_CLASSES
```

The `-F` flag is used to input the name of the data file to be used, and the `-N` flag to input the number of classes represented within the input data.

If you planning to process a very large data set it is a good idea to grab some extra memory. For example:

```
java -Xms600m -Xmx600m APPLICATION_CLASS_FILE_NAME -F FILE_NAME
                                   -N NUMBER_OF_CLASSES
```

The input to the software, in all cases is a (space separated) *binary valued* data set `T`. The set `T` comprises a set of `N` records such that each record (`t`), in turn, comprises a set of *attributes*. Thus:

```
T  = {t | t = subset A}
```

Where `A` is the set of available attributes. The value `D` is then defined as:

```
D = |A|
```

We then say that a particular data set has `D` *columns* and `N` *rows*. A small example data sets might be as follows:

```
1 2 3 6
1 4 5 7
1 3 4 6
1 2 6
1 2 3 4 5 7
```

where, in this case, `A = {1, 2, 3, 4, 5, 6, 7}` of which `6` and `7` represent the possible classes. **Note that attribute numbers are ordered sequentially commencing with the number `1`** (the value `0` has a special meaning). **This includes the class attributes** which should follow on from the last attribute number as in the above example. It is not a good idea to assign some very high value to the class attributes that guarantees that they are numerically after the last attribute number as this introduces inefficiencies into the code. For example, if in the above case, the class numbers are `1001` and `1002` (instead of `6` and `7` respectively) the algorithm will assume that there are `1002` attributes in total in which case there will be `2^1002 - 1` candidate frequent item sets (instead of only `2^7 - 1` candidate sets).

The program assumes support and confidence default threshold values of 20% and 80% respectively. However the user may enter their own thresholds using the `-S` and `-C` flags. Thresholds are expressed as percentages.

Some example invocations, using a discretized/ normalised version of the Pima Indians data set (also available from this site, the raw data can be obtained from the UCI Machine learning Repository (Blake and Merz 1998)) and the two application classes provided by this WWW site, are given below:

```
java ClassCBA_App -Fpima.D38.N768.C2.num -N2 -S1 -C50
java ClassCBA_App10 -Fpima.D38.N768.C2.num -N2
```

(note that the ordering of flags is not significant). The output from each application is a set of CRs (ordered according to the prioritisation given in Sub-section 1.1) plus some diagnostic information (run time, number of rules generated etc.).

3.1. Attribute Reordering

To enhance the efficiency of the CBA algorithm the attributers are ordered according to frequency. This means that the original set of input attributes is transformed (for example the original attribute 1 may no longer be attribute 1, while some other attribute N has been redesignated as attrinute 1). To avoid

ordering users can comment out the lines:

```
newClassification.idInputDataOrdering();
newClassification.recastInputData();
```

in the application source code files (`ClassCBA_App.java` and `ClassCBA_App10.java`).

3.2. Output Options

A number of output options are available (listed below) that may be included in the `ClassCBA_App.java` files. Inspection of this file will indicaye that many of these output options are already there. For example:

```
newClassification.outputNumFreqSets();
```

Note that it would not make sense to include them in ClassCBA_App10.java because different classifiers will be generated for each 10th!

Method Summary	
`public static void`	`outputDataArraySize()` Outputs size (number of records and number of elements) of stored input data set read from input data file.
`public static void`	`outputConversionArrays()` Outputs conversion array (used to renumber columns for input data in terms of frequency of single attributes --- reordering will enhance performance for some ARM algorithms).
`public static void`	`outputSettings()` Outputs command line values provided by user.
`public static void`	`outputRulesWithDefault()` Outputs contents of rule linked list (if any), with reconversion, such that last rule is the default rule.
`public static void`	`outputCBArules()` Outputs contents of CBA rule linked list (if any), includes details of generation process.
`public static void`	`outputNumCBArules()` Outputs number of generated rules.
`public static void`	`outputFrequentSets()` Commences the process of outputting the frequent sets contained in the T-tree.
`public static void`	`outputNumFreqSets()` Commences the process of counting and outputing number of supported nodes in the T-tree. A supported set is assumed to be a non null node in the T-tree.
`public static void`	`outputTestSet()` Outputs test set once it has been split off from the input data. Include in application class after `createTrainingAndTestDataSets()` method.

4. THE CBA ALGORITHM IN MORE DETAIL

The CBA algorithm commences by generating a list of CARS (potential CRs in the target classifier) ordered according to the following schema:

1. **Confidence**: A rule `r1` has priority over a rule `r2` if `confidence(r1) > confidence(r2)`.
2. **Support**: A rule `r1` has priority over a rule `r2` if `confidence(r1)==confidence(r2) && support(r1)>support(r2)`.
3. **Size of antecedent**: A rule `r1` has priority over a rule `r2` if `confidence(r1)==confidence(r2) && support(r1)==support(r2) &&` $|A_{r1}|<|A_{r2}|$.

In Liu et al.'s original paper the third rule is actually expressed as "(if) both the confidence and support of `r1` and `r2` are the same, but `r1` is generated earlier than `r2` (then select `r1`)". However, the effect of this is that rules with lower antecedent cardinality are selected before rules with higher cardinality because of CBA's apriori style CAR generation algorithm (CBA-RG) where low cardinality rules are generated first. In otherwords CBA operates using the same ordering schema as used in the CMAR algorithm (Li et al. 2001).

This rule list (`R`) is then processed, using a variation of the "cover" principle, to produced a classifier. A very much simplified version of the CBA algorithm is given in Figure 2.

```
D <-- distribution array, array of length equal to the number
        of classes in the input dataset which records the
        number of records associated with each class
Rerr <-- 0 (error rate)
For each r in R
   L <-- Local distribution array recording the number of
        records, for each class satisfied by r
   Remove from data set all records satisfied by r
   Rerr <-- Rerr + Number of miss classifications produced by
        the application of rule r
   D <-- D-L
   r.DefaultClass <-- The majority class in D
   Derr <-- Number of miss classifications produced by use
        of the DefaultClass
   r.Terr <-- Rerr+Derr
   if no more records break
End Loop
```

Figure 2: *Simplified version of the CBA algorithm*

On completion of the above the first rule `r` in `R` with the lowest totalError value is identified. All rules after `r` are discarded and a default rule with the class `r.DefaultClass` is added to the end of the classifier. An example of how the above works is given in sub-section 4.1 below.

4.1. Example of basic CBA concept

Given the data set:

```
a b e
a e
```

The rule list is then processes to identify the `DefaultClass` and `Terr` value for each rule. This processing is illustrated in the following table (processing is stopped after rule 4 because there are no more records to consider):

```
b e
c d f
c f
d f
b c e
a d f
```

where e and f are class attributes. This would give a distribution array of [4,4] (the first element representing the class e and the second the class f). Using this dataset the following ordered rule list (with confidence and support values given in parenthesise) may be generated:

```
b -> e (100%, 3)
d -> f (100%, 3)
b c -> e (100%, 1)
a d -> f (100%, 1)
a -> e (67%, 2)
c -> f (67%, 2)
c -> e (33%, 1)
a -> f (33%, 1)
```

Note that the list (R) is ordered according to the above schema.

Rule	L	Rerr	D	DefaultClass	Derr	Terr
b -> e	[3,0]	0	[1,4]	f	1	1
d -> f	[0,3]	0	[1,1]	e	1	1
b c -> e	[1,0]	0	[0,1]	f	0	0
a d -> f	[0,1]	0	[0,0]	e	0	0

The first rule with the lowest total error rate (Terr) is b c -> e so the final classifier becomes:

```
b -> e
d -> f
b c -> e
default f
```

The problem with the above basic algorithm it that it requires |R| passes through the dataset --- clearly undesirable. The actual algorithm described by Liu et al. significantly elaborates on the above with the aim of reducing the number of passes of the data set by determining in advance the values the for the local distribution array L associated with each CAR. This is described in more detail in sub-section 4.2. below.

4.2. Worked Example

In this sub-section the CBA algorithm will be considered in further detail. It is difficult to contrive a simple example (such as that presented in Sub-section 4.1) that serves to demonstrate all the features of CBA so the example given here uses a discretized/ normalised version of Pima Indians set (also available from this site, the raw data can be obtained from the UCI Machine learning Repository (Blake and Merz 1998)). The data was discretized/ normalised using the LUCS-KDD DN software. For the example a confidence threshold of 75% and a support threshold of 1% is used. The data is split so that the first half was used as the training set and the second half the test set. A maximum size of antecedent of 5 attributes was also used. In the example 708 CARs (Classification Association Rules) are generated by Apriori-TFP before the antecedent size limit is reached. The ordered list of CARs is presented in Table 1.

The process of generating the local distribution arrays and producing the final classifier is carried out in three stages:

```
(1) {9 15} -> {38} (100.0%, 6.0)
(2) {6 9 13} -> {38} (100.0%, 6.0)
(3) {2 9 15} -> {38} (100.0%, 6.0)
(4) {3 9 15} -> {38} (100.0%, 6.0)
(5) {6 9 15} -> {38} (100.0%, 6.0)
.................
(248) {2 4 6 8 10} -> {37} (84.61%, 11.0)
(249) {1 2 5 6 9} -> {38} (84.09%, 37.0)
(250) {1 2 6 9} -> {38} (83.67%, 41.0)
(251) {8 17} -> {37} (83.33%, 5.0)
(252) {6 21} -> {38} (83.33%, 5.0)
(253) {1 8 17} -> {37} (83.33%, 5.0)
.................
(329) {2 4 5 6 9} -> {38} (80.95%, 34.0)
(330) {1 4 5 10} -> {37} (80.95%, 17.0)
(331) {1 5 6 10} -> {37} (80.95%, 17.0)
.................
(343) {23} -> {37} (80.0%, 4.0)
(344) {34} -> {38} (80.0%, 4.0)
(345) {7 17} -> {37} (80.0%, 4.0)
.................
(462) {1 4 5 6 8} -> {37} (79.37%, 177.0)
(463) {2 3 9} -> {38} (79.36%, 50.0)
```

1. C AND W RULE IDENTIFICATION: Identify all cRules (correct rules) and wRules (wrinfg rules) in the rule list. Record in the appropriate distribution arrays the number of records covered by strong cRules. Where the wRule has higher precedence than the corresponding cRule store details in a set of records (the set `A`). A "strong" cRule is one with a higher precedence than the corresponding wRule.
2. CONSIDER WRONGLY CLASSIFIED RECORDS: Process the set `A` (the list of records that are wrongly classified). If the "offending" wRule can safely be removed from `R` identify other rules that *override* (have higher precedence) than the desired cRule correspomding to the wRule. If the offending wRule cannot be removed then adjust local distribution array for the wRule accordingly.
3. DETERMINE DEFAULT CLASSES AND TOTAL ERROR COUNTS: Adjust local distribution arrays with respect to overridden rules identified in Stage 2. Then for each "strong" cRule determine the deafault class and the total error count (`Terr`).
4. GENERATE CLASSIFIER: Create classifier.

Each stage is described in greater detail below.

```
(464) {5 6 7 8} -> {37} (79.35%, 173.0)
.................
(525) {3 5 8} -> {37} (77.77%, 203.0)
(526) {5 7 8} -> {37} (77.77%, 189.0)
(527) {1 5 7 8} -> {37} (77.77%, 189.0)
.................
(533) {8 18} -> {37} (77.77%, 7.0)
(534) {1 16} -> {38} (77.77%, 7.0)
(535) {6 16} -> {38} (77.77%, 7.0)
.................
(564) {2 6 8} -> {37} (77.51%, 193.0)
(565) {1 2 3 4 8} -> {37} (77.51%, 193.0)
(566) {2 4 6 8} -> {37} (77.48%, 179.0)
.................
(643) {4 7 9} -> {38} (75.0%, 33.0)
(644) {3 4 7 9} -> {38} (75.0%, 30.0)
(645) {4 5 7 9} -> {38} (75.0%, 30.0)
.................
(664) {6 22} -> {37} (75.0%, 6.0)
(665) {4 20} -> {38} (75.0%, 6.0)
(666) {4 21} -> {38} (75.0%, 6.0)
.................
(704) {1 6 7 8 15} -> {37} (75.0%, 6.0)
(705) {3 6 7 8 15} -> {37} (75.0%, 6.0)
(706) {1 2 7 8 18} -> {37} (75.0%, 6.0)
(707) {1 4 6 7 14} -> {38} (75.0%, 6.0)
(708) {2 3 4 5 21} -> {38} (75.0%, 6.0)
```

Table 1: *Initail CBA rule list with confidence presented as a percentage and support as a number of records*

4.2.1. STAGE 1: C and W rule identification

During Stage 1 the training set is processed and appropriate c and w rules identified for each record. For every identified c rule the appropriate element in the local distribution array is incremented so that the the sum of all the local distribution arrays will be equivalent to the number of records in the training set (384 in this case). For those records where the wRule has higher precedence than the associated cRule the record is stored in a struture (`SetA`) with the following fields:

1. `tid`: The TID (Transaction ID) number of the record
2. `cRule`: The corresponding cRule.
3. `wRule`:The corresponding wRule.
4. `next`: Link to the next structure if any.

At the same time every rule (`r`) which is a

```
1. TID = 358, Itemset =  {1 2 3 4 6 7 8 14 38}
cRule =  {6 7 14} -> {38}
wRule =  {1 2 3 6 8} -> {37}
2. TID = 330, Itemset =  {1 2 3 4 6 7 8 14 38}
cRule =  {6 7 14} -> {38}
wRule =  {1 2 3 6 8} -> {37}
3. TID = 324, Itemset =  {1 2 3 4 6 7 8 14 38}
```

cRule for at least one record is marked (`r.isAcRule <-- true`), and every rule `r`) which is a "strong" cRule is marked (`r.isAstrongCrule <-- true`). Thus for each record if:

1. No cRule, do nothing
2. cRule exits but no wRule, mark cRule as a "Strong" cRule
3. cRule and wRule exist, and cRule has greater precedence than wRule, mark cRule as a "Strong" cRule.
4. cRule and wRule exist, and wRule has greater precedence than cRule, create a SetA structure and add to linked list of SetA structures.

On completion of stage 1: (i) all records which are wrongly classified (but for which there is a corresponding cRule) will be collated into a linked list of `SetA` structures ready for further consideration, (ii) all rules which are "strong" cRules with respect to at least one record will be identified, and (iii) for each cRule the number of records classified correctly will be known.

Some sections of the rule linked list, with cRules and strong cRules marked, as of the end of Stage 1, are presented in Table 2. The sum of the distribution array elements ("class cases discovered") should equal the number of records for which a cRule was identified --- 326 in this case.

The last field shown in Table 2 represents the local distribution array for the rule (`L`). Thus rule `1` satisfies 0 records with class `41` and 6 records with class `42`. Some rulers, such as rule 1 are strong cRules; others, such as rule 330, are simply cRules because there exists a corresponding wRule for a given record which has higher precedence (in the case of rule 330 the higher precedence rule is rule 249 with respect to record 166). This can be confirmed from the set `A (containing details of all miss-classified records such a record no 166) some sections of which are given in Table 3. Note also that many rules are are not cRules with respect to any record.`

`From Table 3 it can be seen that record number 166, which has the class lable 37, is wrongly classified by the wRule {1 2 5 6 9} -> {38} (rule number 249) which has higher precedence than the corresponding`

```
cRule =  {6 7 14} -> {38}
wRule =  {1 2 3 6 8} -> {37}
4. TID = 313, Itemset =  {1 2 3 4 5 6 8 13 38}
cRule =  {5 6 13} -> {38}
wRule =  {1 2 5 6 8} -> {37}
5. TID = 241, Itemset =  {1 4 5 7 8 12 20 28 38}
cRule =  {4 20} -> {38}
wRule =  {5 7 8} -> {37}
6. TID = 224, Itemset =  {1 2 3 4 5 6 8 13 38}
cRule =  {5 6 13} -> {38}
wRule =  {1 2 5 6 8} -> {37}
7. TID = 219, Itemset =  {2 3 5 6 7 9 21 22 37}
cRule =  {6 22} -> {37}
wRule =  {6 21} -> {38}
8. TID = 194, Itemset =  {1 2 3 4 6 7 9 17 37}
cRule =  {7 17} -> {37}
wRule =  {1 2 6 9} -> {38}
9. TID = 182, Itemset =  {1 2 3 4 7 8 12 16 37}
cRule =  {1 2 3 4 8} -> {37}
wRule =  {1 16} -> {38}
10. TID = 166, Itemset =  {1 2 3 4 5 6 9 10 37}
cRule =  {1 4 5 10} -> {37}
wRule =  {1 2 5 6 9} -> {38}
11. TID = 118, Itemset =  {1 2 3 4 6 7 8 14 38}
cRule =  {6 7 14} -> {38}
wRule =  {1 2 3 6 8} -> {37}
12. TID = 116, Itemset =  {1 2 3 4 5 6 8 13 38}
cRule =  {5 6 13} -> {38}
wRule =  {1 2 5 6 8} -> {37}
13. TID = 79, Itemset =  {1 3 4 5 6 9 10 19 37}
cRule =  {1 4 5 10} -> {37}
wRule =  {1 6 9} -> {38}
14. TID = 74, Itemset =  {1 2 3 4 6 7 8 16 37}
cRule =  {1 2 3 6 8} -> {37
wRule =  {1 6 16} -> {38}
15. TID = 22, Itemset =  {1 2 3 4 5 6 8 34 37}
cRule =  {1 2 5 6 8} -> {37}
wRule =  {34} -> {38}
```

Table 3: *Miss classified records (the set A) ordered in reverse*

It may be the case that there are some records which are not satisfied by any rules, or which have only a cRule or a wRule. In the example given here there are 134 records that do not have an associated cRule (i.e. cannot be correctly satisfied by any rules in the rule list). To increase the likelyhood of every record having a cRule a low support threshold is recommended, however this will result in the generation of many more CARS.

cRule {1 4 5 10} -> {37} (rule number 330) --- and so on.

```
(1) {9 15} -> {38} (100.0%, 6.0) * STRONG cRule * [0,6]
(2) {6 9 13} -> {38} (100.0%, 6.0) * STRONG cRule * [0,5]
(3) {2 9 15} -> {38} (100.0%, 6.0) [0,0]
(4) {3 9 15} -> {38} (100.0%, 6.0) [0,0]
(5) {6 9 15} -> {38} (100.0%, 6.0) [0,0]
.................
(248) {2 4 6 8 10} -> {37} (84.61%, 11.0) [0,0]
(249) {1 2 5 6 9} -> {38} (84.09%, 37.0) * STRONG cRule * [0,29]
(250) {1 2 6 9} -> {38} (83.67%, 41.0) * STRONG cRule * [0,1]
(251) {8 17} -> {37} (83.33%, 5.0) * STRONG cRule * [3,0]
(252) {6 21} -> {38} (83.33%, 5.0) [0,0]
(253) {1 8 17} -> {37} (83.33%, 5.0) [0,0]
.................
(329) {2 4 5 6 9} -> {38} (80.95%, 34.0) [0,0]
(330) {1 4 5 10} -> {37} (80.95%, 17.0) * cRule * [2,0]
(331) {1 5 6 10} -> {37} (80.95%, 17.0) [0,0]
.................
(343) {23} -> {37} (80.0%, 4.0) * STRONG cRule * [4,0]
(344) {34} -> {38} (80.0%, 4.0) * STRONG cRule * [0,1]
(345) {7 17} -> {37} (80.0%, 4.0) * cRule * [1,0]
.................
(462) {1 4 5 6 8} -> {37} (79.37%, 177.0) [0,0]
(463) {2 3 9} -> {38} (79.36%, 50.0) * STRONG cRule * [0,2]
(464) {5 6 7 8} -> {37} (79.35%, 173.0) [0,0]
.................
(525) {3 5 8} -> {37} (77.77%, 203.0)
(526) {5 7 8} -> {37} (77.77%, 189.0)
(527) {1 5 7 8} -> {37} (77.77%, 189.0)
.................
(533) {8 18} -> {37} (77.77%, 7.0) [0,0]
(534) {1 16} -> {38} (77.77%, 7.0) [0,0]
(535) {6 16} -> {38} (77.77%, 7.0) [0,0]
.................
(564) {2 6 8} -> {37} (77.51%, 193.0) [0,0]
(565) {1 2 3 4 8} -> {37} (77.51%, 193.0) * cRule * [1,0]
(566) {2 4 6 8} -> {37} (77.48%, 179.0) [0,0]
.................
(643) {4 7 9} -> {38} (75.0%, 33.0) [0,0]
(644) {3 4 7 9} -> {38} (75.0%, 30.0) [0,0]
(645) {4 5 7 9} -> {38} (75.0%, 30.0) [0,0]
.................
(664) {6 22} -> {37} (75.0%, 6.0) * cRule * [1,0]
(665) {4 20} -> {38} (75.0%, 6.0) * STRONG cRule * [0,2]
(666) {4 21} -> {38} (75.0%, 6.0) [0,0]
.................
(704) {1 6 7 8 15} -> {37} (75.0%, 6.0) [0,0]
(705) {3 6 7 8 15} -> {37} (75.0%, 6.0) [0,0]
(706) {1 2 7 8 18} -> {37} (75.0%, 6.0) [0,0]
(707) {1 4 6 7 14} -> {38} (75.0%, 6.0) [0,0]
(708) {2 3 4 5 21} -> {38} (75.0%, 6.0) [0,0]
```

Table 2: *Rule list at the end of Stage 1*

4.2.2. STAGE 2: Consider wrongly classified records

ii. If there are intervening "strong"

In stage 2 we process the set A, the set of records that are not classified correctly. Note that for each "unreachable" cRule referenced in the set A the associated local distribution array has already been incremented accordingly. For each record in A there are two posiblilities:

1. **Option 1**: The wRule is a "strong" cRule for some other record, and thus the wRule cannot be omiited from the desired classifier. The appropriate element in the distribution array associated with the wRule must therefore be incremented, and that for the cRule decremented. One example of this is the case for TID 22 in the set A (klast entry given in table 3) whose wRule {34} -> {38} (rule number 344) is a "strong" cRule for another record. Thus the class record array for this wRule will be increased by 1 and the classRecord for the cRule reduced by 1.
2. **Option 2**: The wRule associated with the set A record is not marked as a cRule for any other record and thus will be left out of the final classifier (TIDS 182, 219 and 241 in the above example). In this case:
    i. If there are no intervening "stong" cRules between the wRule and the desired cRule, make the cRule a "strong" cRule (it may already be marked as such). TIDs 241 and 182 are examples of this.
    ii. If there are intervening "strong" cRules, increment the appropriate element in the local distribution array for the intervening "strong" cRule with the higest precedence, and decrement the cRule local distribution array accordingly. TID 219 is an example of this where the intervening strong c rule is rule 463, {2 3 9} -> {38} (the w rule in this case, {6 21} -> {38}; is number 252 and the c rule, {6 22} -> {37}, is number 664.

At the end of stage 2 we should have the sum of the counts in the local distribution arrays for each rule, this will be equivalent to the number of records in the training set.

Note that the intervening rules, if identified, are stored in an `Overides` structure which has the following four fields:

1. `cRule`: The overriden cRule
2. `tid`: The record TID number in the training set which is correctly classified by the cRule.
3. `classLabel`: The associated class label.
4. `next`: Link to next record

Some sections of the rule linked list as of the end of stage 2 are presented in Table 4 below. Note that some rules have list of rules they overide. For example rule 463 overides rule 664 with respect to TID 219. Note also that "strong" cRules are the only rules that may potentially be included in the final classifier. For example rule 565 which is a cRule, but not a "strong" cRule, will not be included.

```
(1) {9 15} -> {38} (100.0%, 6.0) * STRONG cRule [0,6]
(2) {6 9 13} -> {38} (100.0%, 6.0) * STRONG cRule * [0,5]
(3) {2 9 15} -> {38} (100.0%, 6.0) [0,0]
(4) {3 9 15} -> {38} (100.0%, 6.0) [0,0]
(5) {6 9 15} -> {38} (100.0%, 6.0) [0,0]
.................
(248) {2 4 6 8 10} -> {37} (84.61%, 11.0)[0,0]
(249) {1 2 5 6 9} -> {38} (84.09%, 37.0) * STRONG cRule * [1,29]
(250) {1 2 6 9} -> {38} (83.67%, 41.0) * STRONG cRule *[1,1]
(251) {8 17} -> {37} (83.33%, 5.0) * STRONG cRule * [3,0]
(252) {6 21} -> {38} (83.33%, 5.0) [0,0]
(253) {1 8 17} -> {37} (83.33%, 5.0) [0,0]
.................
(329) {2 4 5 6 9} -> {38} (80.95%, 34.0) [0,0]
(330) {1 4 5 10} -> {37} (80.95%, 17.0) * cRule * [0,0]
(331) {1 5 6 10} -> {37} (80.95%, 17.0) [0,0]
.................
(343) {23} -> {37} (80.0%, 4.0) * STRONG cRule * [4,0]
(344) {34} -> {38} (80.0%, 4.0) * STRONG cRule * [1,1]
(345) {7 17} -> {37} (80.0%, 4.0) * cRule * [0,0]
.................
```

```
(462) {1 4 5 6 8} -> {37} (79.37%, 177.0) [0,0]
(463) {2 3 9} -> {38} (79.36%, 50.0) * STRONG cRule * [1,2]
        Overrides linked list: {6 22} -> {37} , TID = 219, classLabel = 37
(464) {5 6 7 8} -> {37} (79.35%, 173.0) [0,0]
.................
(525) {3 5 8} -> {37} (77.77%, 203.0) [0,0]
(526) {5 7 8} -> {37} (77.77%, 189.0) [0,0]
(527) {1 5 7 8} -> {37} (77.77%, 189.0) [0,0]
.................
(533) {8 18} -> {37} (77.77%, 7.0) [0,0]
(534) {1 16} -> {38} (77.77%, 7.0) [0,0]
(535) {6 16} -> {38} (77.77%, 7.0) [0,0]
.................
(564) {2 6 8} -> {37} (77.51%, 193.0) [0,0]
(565) {1 2 3 4 8} -> {37} (77.51%, 193.0) * cRule * [1,0]
(566) {2 4 6 8} -> {37} (77.48%, 179.0) [0,0]
.................
(643) {4 7 9} -> {38} (75.0%, 33.0) [0,0]
(644) {3 4 7 9} -> {38} (75.0%, 30.0) [0,0]
(645) {4 5 7 9} -> {38} (75.0%, 30.0) [0,0]
.................
(664) {6 22} -> {37} (75.0%, 6.0) * cRule * [1,0]
(665) {4 20} -> {38} (75.0%, 6.0) * STRONG cRule * [0,2]
(666) {4 21} -> {38} (75.0%, 6.0) [0,0]
.................
(704) {1 6 7 8 15} -> {37} (75.0%, 6.0) [0,0]
(705) {3 6 7 8 15} -> {37} (75.0%, 6.0) [0,0]
(706) {1 2 7 8 18} -> {37} (75.0%, 6.0) [0,0]
(707) {1 4 6 7 14} -> {38} (75.0%, 6.0) [0,0]
(708) {2 3 4 5 21} -> {38} (75.0%, 6.0) [0,0]
```

Table 4: *Rule list at the end of Stage 2*

STAGE 3: Determine default class and total error counts

During stage 3 the final list classification rules is generated by processesing the CBA rule list so far. Note that default rule for the classifier is the last rule selected to be included.

For each "strong" cRule in the list that correctly classifies at least one record (as a result of stage 2 a strong c rule may no longer correctly classify any records) the rule is processed as follows:

1. If there are any overriding rules these are processed first. If the TID referenced by the current overridiing rule is already satisfied by a previous rule decrement the current rule distribution array, otherwise decrement overriding rule's distribution array.
2. Calculate the accumulated number of miss-classifications (`ruleErrors`) that will result if the current rule were the last rule in the classifier .
3. Adjust the distribution array so that it no

The final classifier (Table 6) is then constructed by finding the first strong c rule (that satisfies at least one record) with the lowest total error in the CBA rule list. In the example Rule 552 is the first rule with the minimum total errors of 34. The final classifier then comprises all the rules up to and including the identified rule pluss a default rule that produces the default class associated with the identified rule. The given classifier produces an overall accuracy of 73.44%.

```
(1) {9 15} -> {38}  100.0%
(2) {6 9 13} -> {38}  100.0%
(3) {30} -> {37}  100.0%
(4) {10 11} -> {37}  100.0%
(5) {9 16} -> {38}  100.0%
```

longer includes the number of records covered by the current rule.

4. Select a default class, the majority class displayed by the distribution array so far, an place it in the `defaultClass` field for the current rule.
5. Determine the number of errors `defaultError`, from the distribution array, that will result if all unsatisfied records are allocated to the default class.
6. Determine the total number of errors `totalErrors`:

```
totalErrors = ruleErrors+defaultErrors
```

and place in the `totalErrors` field for the currentrule.

The process commences with the generation of a global class distribution array which contains the number of training cases for each class. 250 Records for class 37, and 134 for class 38 in this case.

A number of sections from the rule list after stage 3 of processing are given in Table 5 below. Note that the total number of errors decreases as the number of potential rules in the classifier increases.

```
(6) {4 6 21} -> {38}  100.0%
(7) {4 5 8 10} -> {37}  92.3%
(8) {9 13} -> {38}  87.5%
(9) {1 3 18} -> {37}  87.5%
(10) {1 6 16} -> {38}  87.5%
(11) {6 8 10} -> {37}  86.66%
(12) {1 2 5 6 9} -> {38}  84.09%
(13) {1 2 6 9} -> {38}  83.67%
(14) {8 17} -> {37}  83.33%
(15) {6 8 22} -> {37}  83.33%
(16) {1 6 9} -> {38}  83.01%
(17) {1 3 5 7 9} -> {38}  82.85%
(18) {1 2 9} -> {38}  82.14%
(19) {23} -> {37}  80.0%
(20) {34} -> {38}  80.0%
(21) {2 4 20} -> {38}  80.0%
(22) {5 6 7 19} -> {37}  80.0%
(23) {4 5 6 7 8} -> {37}  79.5%
(24) {1 2 5 6 8} -> {37}  79.39%
(25) {2 3 9} -> {38}  79.36%
(26) {3 5 7 8} -> {37}  78.38%
(27) {2 5 7 8} -> {37}  77.96%
(28) {1 2 3 6 8} -> {37}  77.95%
(29) {2 3 6 7 8} -> {37}  77.92%
(30) {1 3 6 8} -> {37}  77.64%
(31)  null -> {38}  0.0%
```

Table 6: *The final classifier*

```
D <-- distribution array, array of length equal to the number
        of classes in the input dataset which records the
        number of records associated with each class
Rerr <-- 0
for each strong cRule in R
    if r.overrideRef != null
        for eech o in r.overridesRef
            if r.overidesRef.tid covered by previous rule then
                 adjust appropriate element in r distribution array
            else decrement appropriate element in o.cRule distribution array
        End loop
    Rerr <-- Rerr + Number of miss classifications produced by
        the application of rule r
    D <-- D-r.L
    r.DefaultClass <-- The majority class in D
    Derr <-- Number of miss classifications produced by use
        of the DefaultClass
    r.Terr <-- Rerr+Derr
End loop[
```

Figure 3: *CBA stage 3 algorithm*

```
(1) {9 15} -> {38} (100.0%, 6.0) * STRONG cRule * [0,6], D-Class = 37, T-errors = 128
(2) {6 9 13} -> {38} (100.0%, 6.0) * STRONG cRule * [0,5], D-Class = 37, T-errors = 123
(3) {2 9 15} -> {38} (100.0%, 6.0) [0,0]
(4) {3 9 15} -> {38} (100.0%, 6.0) [0,0]
```

```
(5) {6 9 15} -> {38} (100.0%, 6.0) [0,0]
.............
(248) {2 4 6 8 10} -> {37} (84.61%, 11.0) [0,0]
(249) {1 2 5 6 9} -> {38} (84.09%, 37.0) * STRONG cRule * [1,29], D-Class = 37, T-errors = 85
(250) {1 2 6 9} -> {38} (83.67%, 41.0) * STRONG cRule * [1,1], D-Class = 37, T-errors = 85
(251) {8 17} -> {37} (83.33%, 5.0) * STRONG cRule * [3,0], D-Class = 37, T-errors = 85
(252) {6 21} -> {38} (83.33%, 5.0) [0,0]
(253) {1 8 17} -> {37} (83.33%, 5.0) [0,0]
.............
(329) {2 4 5 6 9} -> {38} (80.95%, 34.0) [0,0]
(330) {1 4 5 10} -> {37} (80.95%, 17.0) * cRule * [0,0]
(331) {1 5 6 10} -> {37} (80.95%, 17.0) [0,0]
.............
(343) {23} -> {37} (80.0%, 4.0) * STRONG cRule * [4,0], D-Class = 37, T-errors = 81
(344) {34} -> {38} (80.0%, 4.0) * STRONG cRule * [1,1], D-Class = 37, T-errors = 81
(345) {7 17} -> {37} (80.0%, 4.0) * cRule * [0,0]
.............
(462) {1 4 5 6 8} -> {37} (79.37%, 177.0) [0,0]
(463) {2 3 9} -> {38} (79.36%, 50.0) * STRONG cRule * [0,2], D-Class = 38, T-errors = 44
       Overrides linked list:
       {6 22} -> {37} , TID = 219, classLabel = 37
(464) {5 6 7 8} -> {37} (79.35%, 173.0) [0,0]
.............
(551) {2 6 7 8} -> {37} (77.67%, 174.0) [0,0]
(552) {1 3 6 8} -> {37} (77.64%, 198.0) * STRONG cRule * [1,0], D-Class = 38, T-errors = 34
(553) {1 4 5 8} -> {37} (77.64%, 191.0) [0,0]
.............
(525) {3 5 8} -> {37} (77.77%, 203.0) [0,0]
(526) {5 7 8} -> {37} (77.77%, 189.0) [0,0]
(527) {1 5 7 8} -> {37} (77.77%, 189.0) [0,0]
.............
(533) {8 18} -> {37} (77.77%, 7.0) [0,0]
(534) {1 16} -> {38} (77.77%, 7.0) [0,0]
(535) {6 16} -> {38} (77.77%, 7.0) [0,0]
.............
(564) {2 6 8} -> {37} (77.51%, 193.0) [0,0]
(565) {1 2 3 4 8} -> {37} (77.51%, 193.0) * cRule * [1,0]
(566) {2 4 6 8} -> {37} (77.48%, 179.0) [0,0]
.............
(643) {4 7 9} -> {38} (75.0%, 33.0) [0,0]
(644) {3 4 7 9} -> {38} (75.0%, 30.0) [0,0]
(645) {4 5 7 9} -> {38} (75.0%, 30.0) [0,0]
.............
(664) {6 22} -> {37} (75.0%, 6.0) * cRule * [1,0]
(665) {4 20} -> {38} (75.0%, 6.0) * STRONG cRule * [0,2], D-Class = 38, T-errors = 34
(666) {4 21} -> {38} (75.0%, 6.0) [0,0]
.............
(704) {1 6 7 8 15} -> {37} (75.0%, 6.0) [0,0]
(705) {3 6 7 8 15} -> {37} (75.0%, 6.0) [0,0]
(706) {1 2 7 8 18} -> {37} (75.0%, 6.0) [0,0]
(707) {1 4 6 7 14} -> {38} (75.0%, 6.0) [0,0]
(708) {2 3 4 5 21} -> {38} (75.0%, 6.0) [0,0]
```

Table 5: *Rule list at the end of Stage 3*

---

5. CONCUSSIONS

The LUCS-KDD implementation of the CBA algorithm described here has been used successfully by the LUCS-KDD research group to contrast and compare a variety of classification algorithms and techniques. The software is available for free, however the author would appreciate appropriate acknowledgment. The following reference format for referring to the LUCS-KDD implementation of CBA, available from this WWW page, is suggested:

1. Coenen, F. (2004). *LUCS KDD implementation of CBA (Classification Based on Associations)*. http://www.csc.liv.ac.uk/~frans/KDD/Software/CMAR/cba.html, Department of Computer Science, The University of Liverpool, UK.

Should you discover any "bugs" or other problems within the software (or this documentation), do not hesitate to contact the author.

REFERENCES

1. Blake, C.L. and Merz, C.J. (1998). *UCI Repository of machine learning databases*. http://www.ics.uci.edu/~mlearn/MLRepository.html, Irvine, CA: University of California, Department of Information and Computer Science.
2. Coenen, F., Goulbourne, G. and Leng, P. (2001). *Computing Association Rules Using Partial Totals.* In de Raedt, L. and Siebes, A. (Eds), Principles of Data Mining and Knowledge Discovery, Proc PKDD 2001, Spring Verlag LNAI 2168, pp 54-66.
3. Coenen and Leng (2004). *Data Structures for Association Rule Mining: T-trees and P-trees* To appear in IEEE Transaction in Knowledge and Data Engineering.
4. Coenen, F., Leng, P., Goulbourne, G. (2004). *Tree Structures for Mining Association Rules.* Journal of Data Mining and Knowledge Discovery, Vol 15 (7), pp391-398.
5. Li W., Han, J. and Pei, J. (2001). *CMAR: Accurate and Efficient Classification Based on Multiple Class-Association Rules.* Proc ICDM 2001, pp369-376.
6. Liu, B. Hsu, W. and Ma, Y (1998). *Integrating Classification and Assocoiation Rule Mining.* Proceedings KDD-98, New York, 27-31 August. AAAI. pp80-86.
7. Quinlan, J.R. (1992). *C4:5: Progrm for Machine learninng*. Morgan Kaufmann.

Created and maintained by Frans Coenen. Last updated 5 May 2004