EasyChair Preprint

№ 501

Classification based on Associations (CBA) - a performance analysis

Jiří Filip and Tomáš Kliegr

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

September 12, 2018

Classification based on Associations (CBA) - a performance analysis

Jiří Filip and Tomáš Kliegr[0000−0002−7261−0380]

University of Economics, Prague
Department of Information and Knowledge Engineering
nám Winstona Churchilla 4
13067 Prague, Czech Republic
filj03@vse.cz, tomas.kliegr@vse.cz

Abstract. Classification Based on Associations (CBA) has for two decades been the algorithm of choice for researchers as well as practitioners owing to simplicity of the produced rules, accuracy of models, and also fast model building. Two versions of CBA differing in speed – M1 and M2 – were originally proposed by Liu et al in 1998. While the more complex M2 version was originally designated as on average 50% faster, in this article we present benchmarks performed with multiple CBA implementations on the UCI lymph dataset contesting the M2 supremacy: the results show that M1 had faster processing speeds in most evaluated setups. M2 was recorded to be faster only when the number of input rules was very small and the number of input instances was large. We hypothesize that the better performance of the M1 version can be attributed to recent advances in optimization of vectorized operations and memory structures in SciKit learn and R, which the M1 can better utilize due to better predispositions for vectorization. This paper is accompanied by a Python implementation of CBA available at `https://pypi.org/project/pyARC/`.

Keywords: CBA · Classification by Associations · Association Rule Classification · benchmark

1 Introduction

CBA (Classification Based on Associations) [9] is probably the most widely used algorithm from the Association Rule Classification (ARC) family.[1] The original work by Liu et al [10] proposes two different versions of CBA, M1 and M2, which differ in speed but should produce the same results. According to benchmarks in [10], the M1 version, also called a naive CBA-CB, is about 50% slower than M2.

In this paper, we investigate the proposition that M2 is faster than M1. Following the recommendation in [10], most CBA implementations adopt M2 as

[1] This statement is based on citation analysis of main ARC algorithms, such as CPAR, CMAR, FARC-HD and IDS.

the default (and often the only) supported CBA-CB algorithm. The essential difference between M1 and M2 is that M1 takes a straightforward approach to rule pruning resulting in more operations with data or rules than what is performed in the M2 version. The M2 version reduces the number of operations, but this comes at a cost of introducing multiple embedded cycles, counters and conditions. The hypothesis that we evaluate is that an efficient M1 implementation can take advantage of highly optimized vectorized operations available in modern scientific computing environments such as sci-kit learn [11] or R Matrix package [3], and thus overtake M2 in scalability.

The comparison between M1 and M2 can either be performed analytically or empirically. The first option would entail analysis of all operations, taking into account their costs. However, the costs can differ dramatically based on the chosen implementation. Therefore, we have decided to perform the comparison empirically through a benchmark involving our reimplementation of CBA M1 and M2, as well as existing maintained implementations of the CBA algorithm.

This paper is organized as follows. In Section 2 we briefly outline the CBA algorithm. The CBA implementations used in our benchmarks are covered in Section 3. Section 5 gives an overview of the benchmark setup. Finally, Section 6 presents the results and discusses their implications for choice of the right CBA version for given task. The conclusions present the limitations of our research, state availability of our CBA implementation and benchmarking code, and outline future work.

2 Background

While relatively dated, CBA is used as a reference state-of-the-art algorithm in many recent papers in the ARC field (e.g. [2,8]). One of the advantages of CBA is its speed, in a benchmark performed in a seminal paper by [2], CBA came out as one of the fastest algorithms.

The CBA algorithm consists of two distinct phases [10]: association Rule Generation phase (called CBA-RG) and the Classifier Building (CBA-CB) phase. In the CBA-RG phase, all class association rules that meet user-defined thresholds for confidence and support are discovered using the apriori algorithm [1]. The gist of CBA is the CBA-CB phase, which is responsible for removing redundant discovered rules, and creating a classifier from the pruned rule list.

3 CBA implementations

The base comparison between M1 and M2 implementations was performed using our *pyARC* package.[2] While we strived to equally well optimize both M1 and

[2] `https://github.com/jirifilip/pyARC`

M2 algorithm implementations in *pyARC*, for better representativeness of the benchmark we also decided to include other pre-existing CBA implementations: *arulesCBA* [5], arc [6] and *rCBA* [7]. An overview of all CBA implementations involved in our benchmark is provided in Table 1. For the R implementations, a detailed description can be found in the CRAN[3] package documentation. For *pyARC* we provide a short description below.

Table 1: Review of existing CBA implementations

software name	CBA-CB version	language
arulesCBA	M2	R (C)
rCBA	M2 (default), M1	R (Java)
arc	M1	R
pyARC	M1,M2	Python

For the CBA-RG phase, *pyARC* uses the *fim* (frequent itemset mining) package[4], which is a Python wrapper for one of the fastest available apriori implementations. The same apriori implementation is used, for example, in the popular *arules* R package [4]. The CBA-CB phase for both M1 and M2 was implemented in Python, using built-in optimized Python data structures, such as sets, lists and dictionaries.

4 M1 and M2 comparison

The M1 algorithm, as presented by its authors [9], is characterized by its greater time complexity and memory requirements. Figure 1 shows the M1 algorithm in pseudocode.

In the first step, all rules are sorted according to the precedence operator (line 1). In this way, it is ensured that rules with high confidence and support are chosen first. The rules are then iterated in this order – for each rule, it is tested how many instances the rule satisfies. If any instance satisfies rule's antecedent and the instance's class matches the consequent, the rule is marked and inserted into *temp* (line 2-6). After iterating through all the instances, we test if the rule is marked. If it is, it is inserted at the end of the classifier and all the instances in *temp* are removed from the dataset. The default class is selected (the majority class of the remaining instances). Next, it is evaluated how many errors are made by a classifier C consisting of all rules above the current rule (sorted according to the precedence operator), the current rule and a default rule. The number of errors is associated with the current rule.

[3] http://cran.r-project.org/
[4] `https://pypi.org/project/fim/`

1 $R = sort(R)$;
2 **foreach** *rule* $r \in R$ **do**
3     $temp = \varnothing$;
4   **foreach** *data case* $d \in D$ **do**
5   **if** *d satisfies the conditions of r* **then**
6   store d.id in *temp* and mark r if it correctly classifies d
7   **end**
8   **end**
9   **if** *r is marked* **then**
10   insert r at the end of C;
11   delete all the cases with the *ids* in *temp* from D;
12   select a default class for the current C;
13   compute the total number of errors of C;
14   **end**
15 **end**
16 Find the first rule p in C with the lowest total number of errors and drop all the rules after p in C;
17 Add the default class associated with p to the end of C and return C;

Fig. 1: M1 algorithm [9]

After all the rules have been iterated, the rule r with the lowest associated number of errors is found, rules below rule r are discarded. A rule with empty antecedent predicting the default class associated with rule r is then appended below r as the last rule of the final classifier C.

1 $Q = \varnothing; U = \varnothing; A = \varnothing$;
2 **foreach** *case* $d \in D$ **do**
3     $cRule = maxCoverRule(C_c, d)$;
4     $wRule = maxCoverRule(C_w, d)$;
5     $U = U \cup \{cRule\}$;
6   cRule.classCasesCovered[d.class]++;
7   **if** $cRule \succ wRule$ **then**
8         $Q = Q \cup \{cRule\}$;
9   mark cRule;
10   **end**
11   **else** $A = A \cup < d.id,\ d.class,\ cRule,\ wRule >$;
12 **end**

Fig. 2: M2 algorithm, stage 1 [9]

M1 is not always the right choice for pruning the mined rules. It traverses the dataset multiple times[5], keeping the whole dataset and rules in memory. This may not be the optimal solution for very large datasets.

In response, Liu98integratingclassification proposed the M2 version of CBA-CB. Liu98integratingclassification state that M2 needs to go through the dataset less than two times.

M2 can be divided into three distinct stages. For each instance, two rules are found by traversing the set of sorted rules – *cRule*, which classifies the instance correctly, and *wRule*, which classifies it incorrectly. If *cRule* has precedence before *wRule*, *cRule* is marked it means *cRule* will be chosen before *wRule* when building a classifier. If *wRule* has higher precedence, the situation is more complex and the *wRule* and *crule* need to be stored in a separate data structure.

Figure 2 shows the first step of the M2 algorithm in pseudocode. Let U be the set of all crules, Q the set of all crules that have precedence before their corresponding *wRule* and A the set of all conflicting data structures *¡id, y, cRule, wRule¿*, where *wRule* has precedence over *cRule*.

```
1  foreach entry < d.id, d.class, cRule, wRule >∈ A do
2  |  if wRule is marked then
3  |  |  cRule.classCasesCovered[d.class]–;
4  |  |  wRule.classCasesCovered[d.class]++;
5  |  end
6  |  else
7  |  |  wSet = allCoverRules(U, dID.case, cRule);
8  |  |  foreach entry w ∈ wSet do
9  |  |  |  w.replace = w.replace ∪ {< cRule, dID, y >};
10 |  |  |  w.classCasesCovered[y]++
11 |  |  end
12 |  |  Q = Q ∪ wSet;
13 |  end
14 end
```

Fig. 3: M2 algorithm, stage 2 [9]

The goal of the second stage is to traverse all conflicting data structures in A. For each one, we check if *wRule* is marked. If it is, *wRule* is a *cRule* of another instance. In that case, the support and confidence counts are adjusted so that they correctly reflect the support and confidence of said rule. The second stage is depicted in Figure 3.

[5] In subsequent iterations, the dataset shrinks since some instances have been removed.

```
1  classDistr = compClassDistr(D);
2  ruleErrors = 0;
3  Q = sort(Q);
4  foreach rule r ∈ Q do
5      if r.classCasesCovered[r.class] ≠ 0 then
6          foreach entry < rul, dID, y >∈ r.replace do
7              if the dID case has been covered by a previous r then
8                  r.classCassesCovered[y]–;
9              end
10             else rul.classCasesCovered[y]–
11         end
12         ruleErrors = ruleErrors + errorsOfRule(r);
13         classDistr = update(r, classDistr);
14         defaultClass = selectDefault(classDistr);
15         defaultErrors = defErr(defaultClass, classDistr);
16         totalErrors = ruleErrors + defaultErrors;
17         Insert < r, default − class, totalErrors > at end of C;
18     end
19 end
20 find the first rule p in C with the lowest total number of errors and drop
     all the rules after p in C;
21 Add the default class associated with p to the end of C and return C;
```

Fig. 4: M2 algorithm, stage 3 [9]

In the third stage, the final assembly of the classifier takes place. First, we iterate for every rule r in Q. For each such rule (line 7-11), all rules which could be replaced by rule r are checked. If the checked instance has already been covered by previous rule, support and confidence counts are adjusted. Next steps include – as in M1 – calculating total error count and finding the default rule. Then, the rule with the lowest number of total errors is found and the remaining rules are removed from the classifier.

5 Benchmark setup

We used the Lymph dataset from the UCI repository[6]. We chose lymph, because it is included in the original benchmark by [9], who reported M2 being about 30% faster than M1 on this dataset.

We perform two types of benchmarks: sensitivity of processing speed to a) changes in rule count and b) to changes in the data size.

[6] `https://archive.ics.uci.edu/ml/datasets.html`

For the rule count sensitivity comparison, input rules were generated using the *arules* R package with the following parameters: `confidence = 0, support = 0.01, minlen = 1, maxlen = 20`. The arules-based generation was applied to all implementations except for *pyARC*, which used the *topRules* function from the *arc* package to generate the same number of input rules as in the *arules*-based generation.

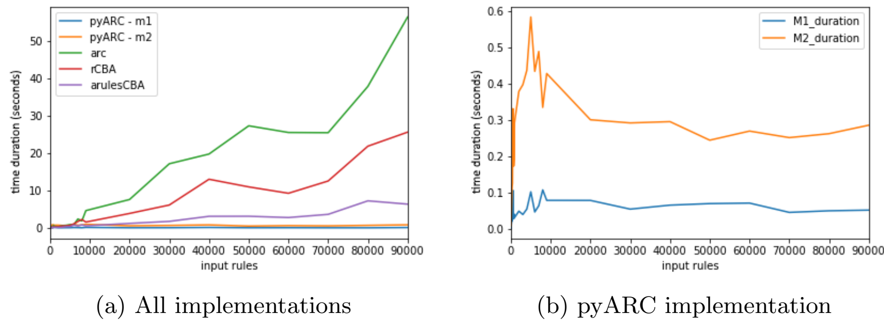
The list of generated rules was subsampled at different sizes and passed to the benchmarked packages. Only execution time of the classifier-building step (CBA-CB) was measured. The run time for each rule count threshold was measured ten times and the results were averaged. The benchmarking scripts are available on GitHub.[7]

For the data size sensitivity comparison, only the first 100 rules generated in the previously described setting were used. To generate various data size thresholds, the training data were doubled at each iteration by oversampling.

The benchmark was run on a machine with two cores (Intel Core M-5Y10 CPU @ 0.8GHz) and 8GB of RAM. The package versions used for the benchmark are: *arc*: 1.2, *rCBA*: 0.4.2, *arulesCBA*: 1.1.3-1.

6 Experimental results

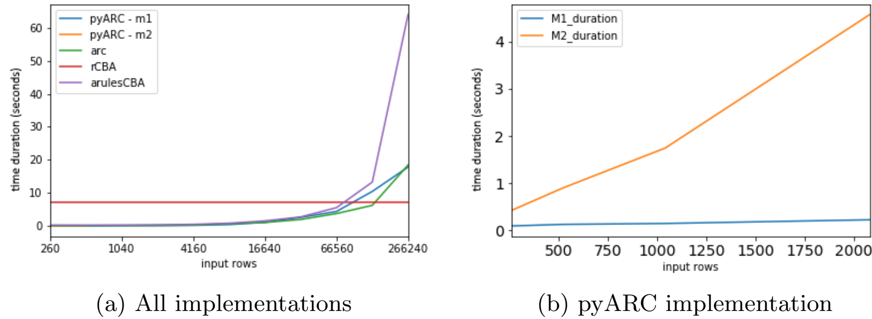
In the first benchmark, we varied the number of rules on the output of CBA-RG, keeping the data size constant. As Figure 5a shows, *pyARC-M1* followed by *pyARC-M2* were the fastest implementations irrespective of the number of input rules. A detailed comparison between *pyARC-M1* and *pyARC-M2* provided by Figure 5b shows that *pyARC-M1* was at least three times faster than *pyARC-M2* across the whole range.



(a) All implementations (b) pyARC implementation

Fig. 5: Runtime – number of input rules is varied

[7] `https://github.com/jirifilip/pyARC/tree/master/notebooks/benchmark`



(a) All implementations (b) pyARC implementation

Fig. 6: Runtime – number of input rows is varied

In the second benchmark, we varied the size of the input datasets, keeping the number of rules constant. The results depicted in Figure 6a show that the *arc* (M1) implementation, followed by *pyARC-M1* were fastest for data set size up to about 133.120 instances. A detailed comparison between *pyARC-M1* and *pyARC-M2* provided by Figure 6b shows that the performance gap increases in favour of M1 when the number instances grows.

Two somewhat outlying patterns can be observed from Figure 5a and Figure 6a. First, *arc* (M1) is the slowest implementation when the number of rules is high, and the fastest implementation (up to 133.120 instances) in the second benchmark focused on increasing the number of input rules. Deterioration of performance for high number of input rules is generally not an issue for CBA implementations, since classification accuracy starts to plateau between 60.000-80.000 input rules [10]. Second, *rCBA* had nearly constant time irrespective of the number of input instances, resulting in the lowest mining time for the largest data set sizes. This may be possibly explained by *rCBA* taking advantage of efficient Java structures for searching the instances, but suffering from a constant overhead relating to data exchange between Java and R (rCBA core is implemented in Java, but the software provides an R interface). However, note that the number of rules in the second benchmark was only 100.

Overall, the results show that M1 version of CBA is faster than M2 in most benchmarked combinations.

7 Conclusion

The experiments presented in this paper contest the previously held belief that M2 version of CBA is always faster than the M1 version. To ensure robustness of our results, we followed crossvalidation, averaged execution times from repeated experiments, and chose a dataset originally used in [9] to demonstrate better

performance of M2 over M1. We see a limitation of our results in that they are based on experiments performed on just one dataset. To aid extension to additional datasets, we made the benchmarking framework freely available under an open license. As for other future work, we plan to investigate the intepretability of classifiers produced by the CBA algorithm.

A by-product of our research is the first (to our knowledge) CBA implementation in Python. According to the performed benchmarks, *pyARC* is highly competitive with existing CBA implementations in other languages. The *pyARC* package provides a scikit-learn-like interface, which makes it easy to use and further extend. pyARC is available at GitHub, and can be downloaded via *pip* or *easy_install*.

Acknowledgments

This research was supported by grant IGA 33/2018 and institutional support for research activities of the Faculty of Informatics and Statistics, University of Economics, Prague.

References

1. Aggarwal, C.C., Han, J.: Frequent pattern mining. Springer (2014)
2. Alcala-Fdez, J., Alcala, R., Herrera, F.: A fuzzy association rule-based classification model for high-dimensional problems with genetic rule selection and lateral tuning. IEEE Transactions on Fuzzy Systems **19**(5), 857–872 (2011)
3. Bates, D., Maechler, M.: Matrix: Sparse and Dense Matrix Classes and Methods (2017), `https://CRAN.R-project.org/package=Matrix`, R package version 1.2-8
4. Hahsler, M., Grün, B., Hornik, K.: arules - a computational environment for mining association rules and frequent item sets. Journal of Statistical Software **14**(15), 1–25 (9 2005), `http://www.jstatsoft.org/v14/i15`
5. Johnson, I.: arulesCBA: Classification Based on Association Rules (2016), `https://CRAN.R-project.org/package=arulesCBA`, R package version 1.0.2
6. Kliegr, T.: Association Rule Classification (2016), `https://CRAN.R-project.org/package=arc`, R package version 1.1
7. Kuchar, J.: rCBA: CBA Classifier for R (2018), `https://CRAN.R-project.org/package=rCBA`, R package version 0.4.1
8. Lakkaraju, H., Bach, S.H., Leskovec, J.: Interpretable decision sets: A joint framework for description and prediction. In: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1675–1684. KDD '16, ACM, New York, NY, USA (2016)
9. Liu, B., Hsu, W., Ma, Y.: Integrating classification and association rule mining. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. pp. 80–86. KDD'98, AAAI Press (1998)
10. Liu, B., Ma, Y., Wong, C.K.: Classification using association rules: weaknesses and enhancements. Data mining for scientific applications **591** (2001)
11. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. Journal of machine learning research **12**(Oct), 2825–2830 (2011)