Venturini *et al. J Big Data (2017) 4:44*
https://doi.org/10.1186/s40537-017-0107-2

Journal of Big Data

RESEARCH Open Access

Scaling associative classification for very large datasets

Luca Venturini*, Elena Baralis and Paolo Garza

*Correspondence:
luca.venturini@polito.it
Department of Control and Computer Engineering, Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino, Italy

Abstract

Supervised learning algorithms are nowadays successfully scaling up to datasets that are very large in volume, leveraging the potential of in-memory cluster-computing Big Data frameworks. Still, massive datasets with a number of large-domain categorical features are a difficult challenge for any classifier. Most off-the-shelf solutions cannot cope with this problem. In this work we introduce DAC, a Distributed Associative Classifier. DAC exploits ensemble learning to distribute the training of an associative classifier among parallel workers and improve the final quality of the model. Furthermore, it adopts several novel techniques to reach high scalability without sacrificing quality, among which a preventive pruning of classification rules in the extraction phase based on Gini impurity. We ran experiments on Apache Spark, on a real large-scale dataset with more than 4 billion records and 800 million distinct categories. The results showed that DAC improves on a state-of-the-art solution in both prediction quality and execution time. Since the generated model is human-readable, it can not only classify new records, but also allow understanding both the logic behind the prediction and the properties of the model, becoming a useful aid for decision makers.

Keywords: Apache Spark, Associative classification, Big Data, Machine learning

Introduction

In the recent years, Big Data have received much attention by both the academic and the industrial world, with the aim of fully leveraging the power of the information they hide. The dimensions on which very large datasets usually extend are mainly the size, i.e. the disk storage occupied, the volume, i.e. the number of records, the dimensionality, i.e. the number of features a record can have, and the domain, i.e. the number of distinct values a feature can take. A special effort has been dedicated to Machine learning algorithms, with a profusion of solutions to tackle the scalability problem, on some or all of the dimensions mentioned above.

Scalability on the domain dimension is a special concern for the datasets in which most of the features are categorical. Categorical features have their values expressed in a discrete domain, and no concept of ordering or ranking can be assumed. Discrete or discretized features are a special case of categorical features where an order among the values is defined. The absence of a natural ordering increases the complexity of the treatment of categorical variables, as their values cannot be binned in groups or levels for example.

Associative classifiers are a special category of Machine learning algorithms, where association rule mining is exploited for the purpose of classification. In the past, they have proved to be able to produce classification models of high quality and outperform state-of-art algorithms like decision trees [1]. Moreover, the model produced is readable, as it is made of association rules, can be debugged and even manually tuned if needed, by modifying or deleting specific rules. In a world where the dimensions involved in a Machine learning process go far beyond the human control, the ability to understand and tune the model created should not be underestimated. The high readability can also foster a better understanding of the underlying processes and guide the decision-making operations towards effective actions.

Adapting an associative classifier to cope with very large data volumes has a number of obstacles. Some of these are inherited from association rule and frequent itemsets mining, which usually extract a large set of association rules or frequent itemsets, sometimes larger than the dataset itself. This behavior is clearly unsustainable with very large datasets, and state-of-art solutions strive to scale up to high-cardinality and high-dimensional data [2]. On the other hand, associative classifiers fit categorical domains particularly well [3], and they have the potential to outperform state-of-the-art algorithms on this task. Works like [4, 5] have already attempted to bring an Associative Classifier on a distributed computing framework, and proved the feasibility of such a system.

In this work, we propose a Distributed Associative Classifier, in short DAC, which trains an ensemble model in a distributed computing framework. Our reference architecture for the computing framework is an in-memory cluster computing framework like Apache Spark, on which we perform our experimental session. To achieve high scalability without sacrificing quality, we adopt several novel solutions that effectively exploit the advantages of in-memory computing, like a greedy, preventive pruning of rules in the extraction phase based on Gini impurity, a model consolidation phase to produce a lightweight model and a majority voting scheme based on multiple rules. We test our approach on a categorical dataset that is large in size (over 1TB), volume (more than 4 billion records) and domain (800 million distinct values among all the features). We evaluate the quality and the performance reached against a state-of-the-art solution. The code of DAC is freely available as open source.[1]

The article is organized as follows. "Background" section introduces the reader to important concepts behind associative classification. "The proposed approach" section explains how DAC works, and "Experimental evaluation" section describes the experimental evaluation of DAC. "Related work" section provides an insight on the related previous literature. Finally, "Conclusion" section draws conclusions.

Background

In this section, we introduce the reader to a set of concepts specific of association rule mining and Associative Classifiers (ACs). In association rule mining, a set of rules is automatically extracted from a dataset. The dataset is usually represented as a set of

[1] The code is publicly available at https://gitlab.com/dbdmg/dac.

transactions, where each transaction is itself represented as a set of items, called itemset. In associative classification, one of the items is the class item, or label.

Equivalently, in classification, the dataset is represented as a structured table of records and features. Each feature is identified by a *feature_id*, that is set to some value v for each record, or to a null value for not available information. In this work, we will mainly refer to the first notation, but the mapping between the two is straightforward: a simple concatenation of *feature_id* and v will serve as item for the transaction, or record. Not available data are represented again with a null value, or not represented at all, as transactions do not have a fixed structure. The common practice in classification is to define a training set, i.e. a part of the labeled dataset that is used to train the algorithm, and a test set, from which the labels are removed. The two sets are used together with other techniques, like cross-validation, to simulate the behavior of the algorithm towards unlabeled, new data and validate its performance.

Association rules are made of an *antecedent* itemset A, and a *consequent* itemset B, and are read as A yields B, or $A \Rightarrow B$. When the consequent is made of a single item, and specifically an item belonging to the set of class labels, the association rule can be used to label the record. We inherit the naming in [6] and call these rules Class Association Rules, or CARs.

Both association rules and CARs share a number of metrics that measure their strength and statistical significance [3]. We here list the ones mentioned in this paper. The support count (*supCount*) of an itemset is the number of transactions in the dataset D that contain the whole itemset. The support of a rule $A \Rightarrow B$ is defined as $supCount(A \cup B)/|D|$, where $|D|$ is the cardinality of D. The confidence of the rule is defined as $supCount(A \cup B)/supCount(A)$, and in CARs it measures how precise the rule is at labeling a record. The χ^2 of a CAR is the value of the χ^2 statistics computed against the distribution of the classes in D, which states whether the assumption of correlation between the antecedent and the consequent is statistically significant.

Another measure that is widely used in classification algorithms is the Gini impurity [7]. The Gini impurity measures how often a record would be wrongly labeled, if labeled randomly with the distribution of the classes in the dataset. It is used for example in decision trees, to evaluate the quality of the splits at each node. Given N classes, the Gini impurity of a dataset, or portion of it, is computed as

$$Gini = \sum_{i=1}^{N} f_i(1 - f_i)$$

where f_i is the frequency of class i in the dataset, or portion of it, for which we are computing the impurity. A portion of dataset is considered pure if its Gini is equal to 0, that happens when only a single label appears. We will refer to the Gini Impurity of an itemset, as the impurity of the portion of the dataset that contains the itemset.

The proposed approach

Traditionally, the training phase of an associative classifier is a memory-intensive process, often executed out-of-core. The vast majority of the techniques has at least an instant of time where a very large set of itemsets or rules has been extracted and not

yet pruned. This model cannot leverage the advantages of our reference architecture, an in-memory cluster computing framework like Apache Spark. In building a scalable associative classifier, we have been guided by the two following design principles: (i) anticipating pruning before the actual extraction of the rules, and (ii) moving from a large model that predicts with only the first matching rule toward a lightweight model, that compensates the loss in size by applying all the rules that match. These two principles aim at reducing the amount of rules contemporarily present in the main memory at any given instant of time, allowing for an effective exploitation of the in-memory computing platform.

The baseline framework on which we build for the training of our Distributed Associative Classifier, namely DAC, is as follows.

1. The dataset is split into N partitions, each one sampled from the original dataset with a ratio r;
2. Within each partition, a rule extraction phase occurs, that produces a model as a set of CARs. The CARs found are filtered by minimum support, minimum confidence and minimum χ^2 and optionally further pruned with a database coverage phase;
3. The generated N models are collected in an ensemble.

Following our first design principle, we aimed at devising an extraction phase that made the work of the posterior pruning extremely reduced or null, in the best case. We have therefore adopted a greedy approach based on the Gini impurity of an item, keeping in mind the second design principle presented before, that we finally want a smaller model where several rules can collaborate for the prediction, instead of a single first-match. This calls for shorter rules, that can more easily match new records and avoid over-fitting. In order to follow such a route without sacrificing predictive quality we designed several solutions that will be presented in the next sections, namely: (i) an FP-growth-like CAR extractor that produces only useful classification rules, in a greedy fashion, by exploiting the Gini impurity; (ii) an added model consolidation phase for the generation of the ensemble that reduces further the size of the final model; (iii) new voting strategies for the ensemble that exploit the before-mentioned novelties.

CAP-growth

The FP-tree is an effective solution for frequent itemsets extraction, and is often adapted to the extraction of CARs [3]. Moreover, it adapts well to in-memory computing, as its construction needs only two scans of the dataset and, once built, the FP-tree stores in the main memory all the necessary information for frequent itemsets or CARs extraction.

However, there is a twofold motivation behind designing an alternative to the FP-tree, like [8, 9], as method of storage for the patterns that will build the final CARs. First, the FP-tree is designed to build all frequent patterns, that are a superset of what we look for when we build CARs. Second, being frequent does not always coincide with being useful, and using the standard FP-growth algorithm would lead the growth of an overwhelming number of rules that would impede the descent to lower supports, where more useful information may dwell. Guided by these considerations, and keeping in mind the design

principles outlined in the beginning of the section, we designed an FP-growth-like algorithm called CAP-growth, for Class Association Patterns growth.

CAP-growth stores the information that is useful for extracting CARs in a CAP-tree. Similarly to an FP-tree, this structure allows to compactly store all the information needed to extract association rules reading the dataset only twice. Differently from the FP-tree, a CAP-tree stores in each node extra information useful to extract only CARs, as it is usually done in single-machine approaches [8, 9]. Moreover, the first phase of the CAP-tree's construction sorts the frequent items by their Gini impurity, which will help the extraction of more useful rules in the CAP-growth phase.

The algorithm that builds a CAP-tree is detailed in Algorithm 1.

Algorithm 1: CAP-tree building

Input : A transaction DB labeled with classes - DB
Input : A minimum support threshold - $minsup$
Output: A CAP-tree

1 Scan the DB once. Collect L, the list of frequent items ($support >= minsup$). Sort L by decreasing IG and filter out items with $IG \leq 0$.
2 Create the root of a CAP-tree T and label it as *null*.
3 **for** *each labeled transaction* t **do**
4 select only the items in t that appear in L and sort them according to the order in L, obtaining t'
5 call insert(t', T)
6 **end**
7 **Function** *insert (transaction* t*, node* T*)*
8 h = first item of t
9 **if** T *has a child* T' *s.t.* $T'.id = h.id$ **then**
10 T'.freqs[t.class]+=1
11 **else**
12 create a new node T'
13 init T'.id = h.id and T'.freqs to an array of zeros
14 T'.freqs[t.class]+=1
15 T'.parent = T
16 update the header table
17 **end**
18 $t' = t \backslash h$
19 **if** t' *is not empty* **then**
20 insert(t', T')
21 **end**

Given a minimum support threshold, which is used to recognize frequent itemsets, the algorithm scans the dataset twice. In the first pass (line 1), it builds a list L of frequent items, with decreasing and strictly positive Information Gain, which is computed as follows:

$$IG_i = Gini_D - [w_i Gini_i + (1 - w_i) Gini_D] \quad (1)$$

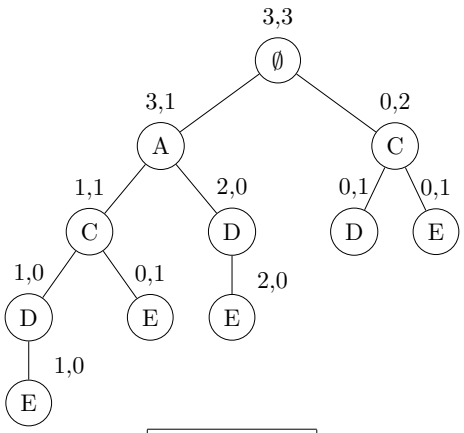
in which $Gini_D$ is the impurity of the global dataset, $Gini_i$ is the impurity of item i, and w_i is the ratio of dataset containing the item. Since we are considering the item alone, we assume that the $(1 - w_i)$-th part of the dataset not covered by the item has a distribution of the labels identical to global distribution (same Gini).

Equation 1 simplifies as

$$IG_i = w_i(Gini_D - Gini_i) \tag{2}$$

In this first passage, we can also obtain the frequency of the classes in the entire dataset, which is used in the CAP-growth's extraction phase. In the second pass (lines 2–6), we insert each read transaction in the CAP-tree (line 5), maintaining a header table that keeps track of the pointers to the nodes in the tree that store the frequent items, like in the original FP-tree (line 16). Before being inserted, the transaction is cleaned from the infrequent items and reordered according to the order of L (decreasing IG) (line 4). The insertion updates the structure of the CAP-tree to keep track of the label of the transaction in an array of frequencies (lines 9–17). This allows the direct extraction of CARs and the computation of the IG and the confidence of the rules in the CAP-growth. Figure 1 shows the CAP-tree built on the toy dataset in Table 1, with the minimum support threshold set to 0.3, that is 2 records. Each node of the tree is labeled with the array of the frequencies of the classes, positive and negative respectively. In this tree we see how item B has been pruned, since its IG is 0, and how the remaining items are sorted and inserted by their IG, with item A being the first and the most useful for classification.

CAP-growth extracts a set of CARs from the CAP-tree descending the tree greedily. This yields that, since the frequent items are sorted by decreasing IG, we evaluate the



Header Table		
Item	Freqs	
	+	-
A	3	1
C	1	3
D	3	1
E	3	2

Fig. 1 A CAP-tree example. Built over the toy dataset with minimum support equal to 0.3

Table 1 An example transactional dataset, binary-labeled

TID	Transaction	Class
1	$\{A, B, D, E\}$	+
2	$\{B, C, E\}$	−
3	$\{A, B, D, E\}$	+
4	$\{A, B, C, E\}$	−
5	$\{A, B, C, D, E\}$	+
6	$\{B, C, D\}$	−

rules made of high-IG items first. The rationale that guided the design of the algorithm is to avoid redundant rules, where possible, while keeping the length of the rules minimal.

The following example illustrates some ways in which redundancy affects CARs. In other approaches, this redundancy is often reduced after the extraction of CARs, as shown in "Related work". We provide this example so that the reader may later gain an intuition of where CAP-growth helps reducing redundancy before the extraction itself. In Fig. 2 we see all the CARs in the set of association rules extracted with the standard FP-Growth, with minimum support set to 0.3 (2 rows or more) and minimum confidence 0.51 on the toy dataset in Table 1. 18 CARs for a dataset of 6 records are clearly redundant. A first, evident source of this redundancy is item B, which is present in all the records in the dataset. This results in having, for any rule generated, an identical rule with B appended, that does not contribute to the classification and lengthens the model. A similar situation happens with item E. Likewise, item C appears in many rules, all of which agree in classifying a record as negative: item C itself would be sufficient as antecedent of the rule. The same holds for other rules as well.

As previously stated, CAP-growth aims to avoid the redundancy of the example above. Algorithm 2 shows the pseudocode for CAP-growth. Similarly to Eq. 2, we define the Information Gain for a node as

$$IG_T = w_T(Gini_{T.parent} - Gini_T) \tag{3}$$

where w_T is the ratio of transactions represented in node T with regards to its parent node, and Gini is computed on the frequencies of the labels stored in the node.

$$\begin{array}{cc} EDA \Rightarrow + & BEDA \Rightarrow + \\ ED \Rightarrow + & BED \Rightarrow + \\ EC \Rightarrow - & BEC \Rightarrow - \\ EA \Rightarrow + & BEA \Rightarrow + \\ DA \Rightarrow + & BDA \Rightarrow + \\ A \Rightarrow + & BA \Rightarrow + \\ E \Rightarrow + & BE \Rightarrow + \\ C \Rightarrow - & BC \Rightarrow - \\ D \Rightarrow + & BD \Rightarrow + \end{array}$$

Fig. 2 An example model with CARs for the dataset in Table 1

Algorithm 2: CAP-growth

```
   Input  : a CAP-tree
   Input  : A minimum support threshold - minsup
   Input  : A minimum confidence threshold - minconf
   Input  : A minimum chi2 threshold - minchi2
   Output: A list of CARs
 1 rules = ∅
 2 for each child T of CAP-tree.root do
 3 |  rules += extract(T)
 4 end
 5 return rules
 6 Function extract(node T)
 7 |  rules = ∅
 8 |  if IG(T) <= 0 then //negative Information Gain: do not generate any rule
 9 |  |  return ∅
10 |  end
11 |  if Gini(T) == 0 then //pure node: try to generate a rule
12 |  |  return generateRule(T)
13 |  end
14 |  for each child T' of T do
15 |  |  rules += extract(T')
16 |  end
17 |  if rules is ∅ then //none of the children has produced a rule: try to generate a rule
18 |  |  return generateRule(T)
19 |  end
20 |  return rules
21 Function generateRule(node T)
22 |  consequent = class with highest value in T.freqs[]
23 |  antecedent = set of items in the path from T to CAP-tree.root
24 |  tree = CAP-tree conditioned by the items in antecedent
25 |  freqs = tree.root.freqs
26 |  sup = freqs[consequent] / totCount
27 |  supAntecedent = freqs.sum / totCount
28 |  from sup, supAntecedent and the global frequencies of the classes computed in the first pass
   |  of Algorithm 1 compute support, confidence and χ² for the generated rule: antecedent ⇒
   |  consequent
29 |  if sup < minsup or conf < minconf or χ² < minchi2 then
30 |  |  return ∅
31 |  end
32 |  return rule
```

The algorithm is a recursive call to the function extract (line 6), which visits in a depth-first fashion the CAP-tree. The stopping criteria of this visit are:

1. A negative Information Gain for the current node. In this case, we do not generate any rule (line 9).
2. A Gini impurity for the current node equal to 0. Being the Gini impurity always strictly decreasing, this makes the current node the first pure node in the path from the root to this node, i.e. we see only one label for it. We try to generate a rule (line 12).

Whenever none of the children of a node does generate a rule, the node itself tries to generate a new rule (lines 14–19). This can occur when the children nodes do not see enough samples to satisfy the minimum support threshold, for example, or if the current node is a leaf.

The function that generates a new rule (lines 21–32) needs first to recollect the frequencies of the labels from all the nodes where the current pattern appears. Like in the original FP-growth, this is done by projecting the CAP-tree recursively on all the items of the pattern, that is all the nodes in the path to the root (line 24). At the end of the

projection, the root node contains the array of classes' frequencies for the pattern (line 25). With it, we can compute the support, the confidence and the χ^2 of the rule we are trying to generate (lines 26–28). If any of the measures does not satisfy the minimum constraints, the rule is not generated (line 29).

In Fig. 3 we see an example of the CAP-growth algorithm, run on the CAP-tree of Fig. 1, with minimum support, confidence and χ^2 set respectively to 0.3, 0.51 and 0. In the figure, each node is labeled by the array of frequencies of the classes and the resulting Gini impurity. The root of the tree has a Gini impurity of 0.5. Its first child to be explored stores item A with a Gini of 0.375 (Fig. 3a). Having a positive IG and a non-null Gini, we continue the descent to its children. The first to be explored describes the pattern A, C (Fig. 3b). This node has a Gini index of 0.5, thus a negative IG. This means that the addition of item C to the pattern only worsens the ability of A in predicting a label. We therefore do not explore anymore this pattern and its offsprings. The other sibling (Fig. 3c), storing item D, is pure for the positive class: continuing the descent further would only lengthen the rule without any improvement. We reconstruct the real frequencies of itemset $\{A, D\}$ to see if the rule $A, D \Rightarrow +$ is really worth to generate and compute its support, confidence, and χ^2. First, we need to project the CAP-tree for item D. The header table stores the pointers to the three nodes that store this item. Only the parts of the tree that end to these three nodes are kept, and all the surviving nodes and the header table are updated in their frequency arrays to reflect this change. Now we have a CAP-tree storing only the transactions that contain item D. We project again for item A. The header table points to a single node that stores this item, and its frequency array, updated in the step before, is [3,0]. By projecting, the CAP-tree reduces to the root node alone, whose frequencies also are updated to [3,0]. This is the frequency array for

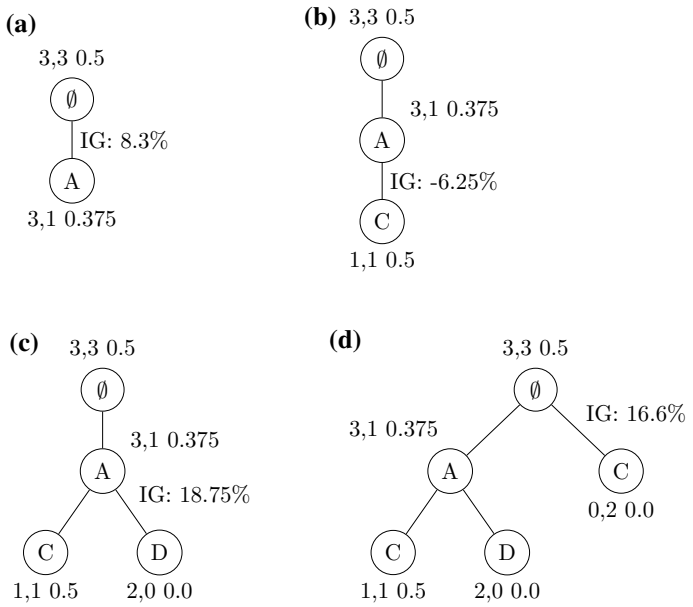


Fig. 3 CAP-growth. Here, the letters **a**–**d** refer to the order in which the figures must be read. Example visit of the CAP-tree in Fig. 1

itemset $\{A, D\}$. Thus, the rule $A, D \Rightarrow +$ has confidence 1 and support 0.5, and satisfies the minimum thresholds.[2] Rule $A \Rightarrow +$ is not generated, as one of the subpatterns of A has already produced one rule. Finally, we move to the second child of the root, storing item C (Fig. 3d). This is again a pure node. We recollect the frequencies of item C by projection as seen before and get the array [1,3], which produces the rule $C \Rightarrow -$ with support 0.5 and confidence 0.75. The final model is made of only two rules.

It is worth paralleling the strategy in CAP-growth with the one in the database coverage pruning [6]. The database coverage scans the rules extracted and already sorted by prediction quality, and keeps on adding them to the model if they predict correctly at least a transaction not yet covered, and until all the transactions have been covered at least once. Similarly, CAP-growth keeps on adding rules that cover transactions not yet covered, since they are extracted in different branches of the CAP-tree, and does so without extracting the entire set of CARs that satisfy the minimum thresholds. The main difference between the two strategies is in the moment when the pruning is performed: the database coverage acts at the end of the extraction, when all the rules have been already extracted, whereas CAP-growth anticipates the pruning in the extraction phase. The aim of both the strategies is the same, that is generating the least, shortest rules, avoiding redundancy in the model.

Model consolidation

CAP-growth generates a single model, in each partition of the dataset, that is at the same time compact and useful. Still, with massively large datasets, it may happen that the number of partitions to have a sufficient division of the workload is in the order of thousands, or more. Consequently, the number of single models in the ensemble explodes. This results in a larger model to store, more complex to be read and examined by a human, and with longer execution times when applied to predict new records.

To cope with these issues, we shrink the ensemble of the models to a unique model. This is done by merging the models, combining rules with identical antecedent and consequent into a single, new rule. The new rule will need to have an approximation for its support, confidence and χ^2, as it is too expensive to reconstruct the exact ones in this phase. In other words, we anticipate part of the voting that eventually classifies new records to this phase: establishing how two identical rules collapse to a single one is establishing how they would eventually vote in the classification, a priori. Algorithm 3 shows how the consolidation is done.

[2] The minimum χ^2 is set to 0 in this example.

Algorithm 3: Model consolidation

Input : A list of models - *models*
Output: A single model, as a list of CARs

```
1  model = ∅
2  for each m in models do
3  |  model = merge(model, m)
4  end
5  return model
6  Function merge(model m1, model m2)
7  |  m = new model
8  |  rules = m1.rules ∪ m2.rules
9  |  gr = group rules by same antecedent and consequent
10 |  for each group of rules i in gr do
11 |  |  m = m ∪ aggregate(i)
12 |  end
13 |  return m
14 Function aggregate(rules)
15 |  rule = new rule
16 |  r = rules.first
17 |  (rule.antecedent, rule.consequent) = (r.antecedent, r.consequent)
18 |  supports = ⋃_{r∈rules} r.support
19 |  confs = ⋃_{r∈rules} r.confidence
20 |  chis = ⋃_{r∈rules} r.chi2
21 |  (rule.support, rule.confidence, rule.chi2) = g(supports, confs, chis)
22 |  return rule
23 Function g(supports, confs, chis)
24 |  return (max(supports), max(confs), max(chis))
```

We recall that DAC 's training has split the dataset in N partitions and runs a CAP-growth over each partition, thus generating an ensemble of N models. These models are the input for the model consolidation algorithm (Algorithm 3). We reduce the models by applying, recursively two by two, a function *merge* (line 3). This function simply makes the union of the rules in the two models (line 8) and, for each set of identical (in the antecedent and consequent) rules found, applies function *aggregate* (line 11).

Function *aggregate* returns a new rule by choosing the new support, confidence and χ^2 with *g()* (line 21), which actually sets the strategy for the consolidation. The default behavior of $g()$ is returning the maximum of the supports, confidences and χ^2 in input, as an upper bound estimation (line 24). We have also experimented with other possibilities, namely functions that keep the property of associativity and commutativity, i.e. the minimum and the product. Associativity and commutativity in function $g()$ make the consolidation runnable in parallel. In "Evaluation of DAC parameters" section we give details on these experiments.

Voting

In associative classifiers, the models usually label a record by applying the first matching rule based on a quality ranking. Differently from other families of classifiers, associative classifiers usually do not have a score or a vector of probabilities for the prediction, but only the predicted class. Introducing a score for the prediction of the associative classifier, the predictions can express their strength in a continuous domain and we can use measures different from the accuracy to compare the model with others, like the Area Under Curve. Moreover, we have a way to weight the votes in the ensemble, whereas in its simplest implementation every model would have voted with an equal weight, independently of the confidence or support of the rules of each model. This last point is indeed partially covered by the consolidation, but we can still hold in the consolidated model rules, with different antecedents or consequents, that come from different models and contemporarily match a record. Defining a score would mean defining how these many rules contribute to strengthen our belief in predicting a class, when they all agree, or to mitigate our certainty, when they partially disagree.

Given an unlabeled record, for each label i, we define a score p_i as a function of some measure for all rules matching the record, i.e.

$$p_i = f(m(\vec{r}_i)), \qquad \forall i : \vec{r}_i \neq \emptyset$$

where $\vec{r}_i$ is the array of matching rules for label i, m is a measure, e.g. the support or the confidence, and f a function with domain in [0, 1]. If there are no matching rules for a label and the record, p_i is defined as

$$p_i = p_X/|X|, \qquad \forall i \in X$$

where X is set of labels for which we do not have a matching rule, and p_X is defined under a naive assumption of independence as

$$p_X = \prod_{j:\vec{r}_j \neq \emptyset} (1 - p_j), \qquad X = \{i : \vec{r}_i = \emptyset\}$$

If there are no matching rules at all, p_i is default to the probability of each label i in the original dataset. The score vector $\vec{p}$, containing the scores p_i as above defined, is finally normalized to sum to one.

The default setting for m is the confidence, that is a common choice in associative classifiers for the rules' ranking. In preliminary experiments, we tried several alternative choices for m, i.e. the support, its complement ($1 - support$) and the χ^2. We performed further experiments on the two most promising of these, that is the confidence and $1 - support$, which we report in "Evaluation of DAC parameters" section.

The default setting for $f()$ is the $max()$ function, which is an upper bound estimation of the quality of the rule, based on the measures from the models where it was found. Alternatives to this choice are, for example, the minimum or the mean, which are always valid scores whenever $m(\vec{r}_i)$ is defined between 0 and 1. We test and discuss these alternatives in "Evaluation of DAC parameters" section.

Experimental evaluation

In our experimental evaluation, we want to compare DAC with state-of-art approaches in a realistic, large-scale scenario. Among publicly available datasets, we found only one dataset to be very large (i.e. over the Terabyte) and with the characteristics of our problem (i.e. many categorical features), and is described below. As competitors to DAC, we choose the algorithms implemented in the Apache Spark Mllib library [10], as it is a well-proven framework for machine learning on distributed computing [11, 12].

The experiments were performed on a cluster with 30 worker nodes running Cloudera Distribution of Apache Hadoop (CDH5.8.2), which comes with Spark 1.6.0. The cluster has 2TB of RAM, 324 cores, and 773TB of secondary memory. Unless differently specified, all the single experiments are run on 100 executors and a master node with one virtual core and 7GB of RAM each. We used version 1.6.0 of Apache Spark Mllib[3] and version 2.1 of DAC, which is released as open source.[4]

The dataset used in the experiments is the Criteo dataset [13], which has already been used as a benchmark in classification tasks, although only on its continuous features, in [14]. The dataset counts more than 4 billion records, describing the behavior of users in 24 consecutive days towards web ads. The positive class is a click on the showed ad and the negative is a non-click. The records are described by 13 continuous features and 26 categorical features, whose semantics is not disclosed. For the experiments, we selected the categorical features only, as DAC does not handle continuous features without a discretization phase, which is outside the scope of this evaluation. The resulting dataset contains more than 800 million unique items, each appearing once or more, and is larger than 1.2 TB. The negative class appears 97% of the times.

The dataset is characterized by the presence of categorical features and the extreme imbalance of the classes. In the next paragraphs, we will describe our approach toward the two issues.

Managing categorical features To deal with categorical features, we need either an algorithm that supports them natively or a proper encoding of the features into integer or binary values. A common solution, which would enable the exploitation of many widely-used classification algorithms, like SVMs or artificial neural networks, is to use the so-called "one-hot" encoding. With it, all the distinct values appearing in the dataset are transformed to a binary feature, which represents the presence or absence of the value in the record. With all the categorical features mapped to binary ones, we would be able to try many solutions for classification.

We tried one-hot encoding as implemented in Mllib. Unfortunately, with so many unique values (more than 800 million) the preprocessing quickly grows in memory and fails. A possible reason is the fact that the records are stored in a dense vector. Since with this encoding only a few features would be non-zero, we tried to implement the encoding with a sparse matrix, but the dimensions involved (billions of records by billions of

[3] https://spark.apache.org/docs/1.6.0/mllib-guide.html.

[4] https://gitlab.com/dbdmg/dac/tags/v2.1.

features) showed to be too large also for this kind of representation, and our attempts exhausted the memory available to our testbed.

A different approach is selecting an algorithm that supports natively categorical features without a special encoding, like decision trees or random forests. Again, the number of distinct values in each feature is an issue, due to the metadata that these algorithms need to collect and store to decide the binnings and the splits at each iteration. Not surprisingly, all preliminary experiments again failed for out-of-memory errors. We decided therefore to exploit a technique known as "hashing trick" [15]. With this method, all values are hashed to reduce dimensionality, with inevitable collisions. We therefore progressively reduced the domain of each feature down to 100,000 categories, value that allowed the execution of the random forest algorithm without memory issues. After the reduction of dimensionality, the application of one-hot encoding was still impossible. This therefore excludes from our analysis the Mllib implementations of linear SVM, logistic regression and multilayer perceptron.

Dealing with class imbalance Preliminary experiments showed that neither Random Forests nor DAC were able to handle the highly unbalanced distribution of classes in this dataset. Indeed, the resulting models were respectively trees with all the leaf nodes predicting the majority class and sets of CARs where the minority class was highly underrepresented, when not absent. To cope with this issue, we investigated several techniques, among which instance-based weighting, oversampling, and subsampling. Instance-based weighting assigns a given weight w to each sample, that while building the model is thus counted as if present w times. In the decision tree and the random forest, this weight affects the sample counts of each node and the split decisions. When the weight w is equal to the inverse of the frequencies of the sample's class, this technique can balance the dataset without a physical replication of the records. Although implemented in several popular random forest implementations [16, 17], instance-based weighting is not implemented in Mllib. Oversampling replicates some of the records belonging to the minority class or classes, so that the dataset gets balanced [18]. In our scenario, the application of this technique to the training set did not converge successfully due to memory constraints. Conversely, subsampling extracts a fraction of the majority class or classes, to reduce their volume to a size comparable to the minority class [18]. We applied this technique to the negative class to have a cardinality roughly equal to the positive one, in the training set. The test set was not subsampled.

In these preliminary evaluations we have also tried several settings of the general architecture of DAC. In this phase, we found that sampling with replacement yields a better load balancing, as this operation triggers the shuffling of part of the training dataset and leads to equally-sized partitions, whereas the default partition can see blocks of very different sizes. We have set to $1 / N$ the sampling size for each one of the N models, to have a final training dataset sized as the original one. We have also tried several N, finding in 100 for each partition a value that allowed the CAP-tree of each model to be stored in memory.

Summarizing, our competitor to DAC will be a Random Forest with categorical features, with the values of the categories hashed down to 100,000 different values at most.[5] DAC will be instead evaluated without hashing trick, as it is not necessary. For both, we will subsample the majority class in the training set.

Experimental comparison of DAC and Random Forest

In this section, we evaluate the quality and the performance of DAC and a Random Forest. Our objective is to show how our proposed technique can manage a dataset characterized by a very large volume and domain, and compare the quality of the resulting model with the state of the art. We evaluate our results in the binary tasks with the AUROC, i.e. the area under the ROC curve [19]. DAC will be evaluated with its default settings, i.e. $f() = max$, $m =$ confidence and $g() = max$. The experiments are run with a fivefold cross-validation on the whole dataset, with the K-fold function implemented in the MLUtils of Mllib. Each one had a variable duration on our testbed between 2 and 30 h. All the confidence intervals shown in the plots were computed using a t-student distribution at 95% confidence.

Figure 4 shows the resulting AUROC for the candidate set of models, consisting of (i) DAC with $f() = max$, $g() = max$ and $m =$ confidence, varying minimum support thresholds from 0.01 to 0.0002; (ii) Random Forests with a depth of 4, varying the number of trees from 5 to 100; (iii) a single Decision Tree, with depth 4. The baseline for the results is set by the Decision Tree, which is almost two points below the Random Forest and DAC. The quite large confidence interval for DAC with 0.01 as minimum support makes uncertain the comparison with the two smallest forests, with 5 and 10 models respectively. These last three models are all clearly below the results of the 100-tree forest and DAC with minimum support 0.002, that have a comparable AUROC of 0.644. Significantly better are the results of DAC with minimum supports of 0.001 and 0.0002, this last one scoring the highest AUROC of 0.655, a good point above the 100-tree forest.

Notably, the experiments for the 100-tree forest lasted 30 h in our testbed, against the 20 h of the DAC with minimum support of 0.0002. These high computation costs would certainly be a heavy factor in the choice of a model, as the model with the highest score is not always a viable path. We therefore plot the same scores against the training and testing time of their models, in Fig. 5. In Fig. 5a we see how the training times of the Random Forest grow with the number of models. The Decision Tree shows times higher than both the 5-tree and 10-tree forests, as it does not perform any feature selection, whereas the forests randomly choose $\sqrt{n}$ features for each tree, where n is number of columns, i.e. 26 in our scenario. The half-point advantage in the AUROC of the 100-tree forest on the 10-tree one comes with a cost five times higher in terms of training times. This large gap could make the difference in a scenario where the model needs to be frequently updated, e.g. an application with nightly updates to the training dataset, and could lead to the choice of the shallower model. DAC here demonstrates a highly desirable behavior, as the best model trains in only 500 s, a time respectively 5 and 25 times smaller than the 5-tree and the 100-tree forests, which also have a worse AUROC.

[5] Hashing to larger values or not using hashing was not a viable option for the memory issues explained before.

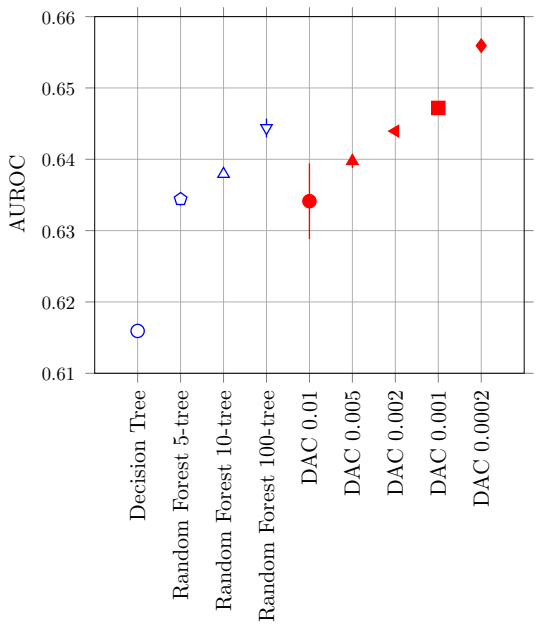


Fig. 4 DAC vs Random Forest. AUROC cross-validated with fivefold on the entire dataset

Moreover, the gap between the training times of the least and most accurate models for DAC is under the 15%, so the latter one is clearly preferable.

The testing times of DAC and the Random Forest have similar trends, depicted in Fig. 5b. Both appear to grow exponentially with the AUROC reached, symptom of models that are more and more complex. For the Random Forest, this complexity is proportional to the total number of splits, or equivalently to the number of trees, since we have a fixed depth. This justifies the alignment of the Decision Tree to the trend of the Random Forests, as in this phase it is practically identical to a Random Forest with one tree. For DAC, the complexity depends on the number of rules extracted and, in this strategy that applies *max* for both $f()$ and $g()$, on the position of the first matching rules in the model for each class, for we need only these for the score. This explains the slightly larger confidence interval on the time axes, whereas the forests all have negligible intervals, due to the constant number of splits traversed by each record for a prediction. Despite this, we can still safely affirm that DAC reaches the same quality of a Random Forest within smaller testing times, as it happens with DAC with minimum support 0.002 and the 100-tree forest. At the same time, we can say that, given a comparable testing time, DAC performs better, as in the case with minimum support 0.005 and the 10-tree forest.

Evaluation of DAC parameters

We tested the effect of the choice of the algorithms' parameters on the quality of the model, to eventually select one or several candidates for more thorough tests.

For DAC, we evaluated different choices for:

- The use of the database coverage technique, [yes/no]
- The function used in the voting, $f()$, [max/min/mean]
- The measure used in $f()$, m, [confidence/1-support]

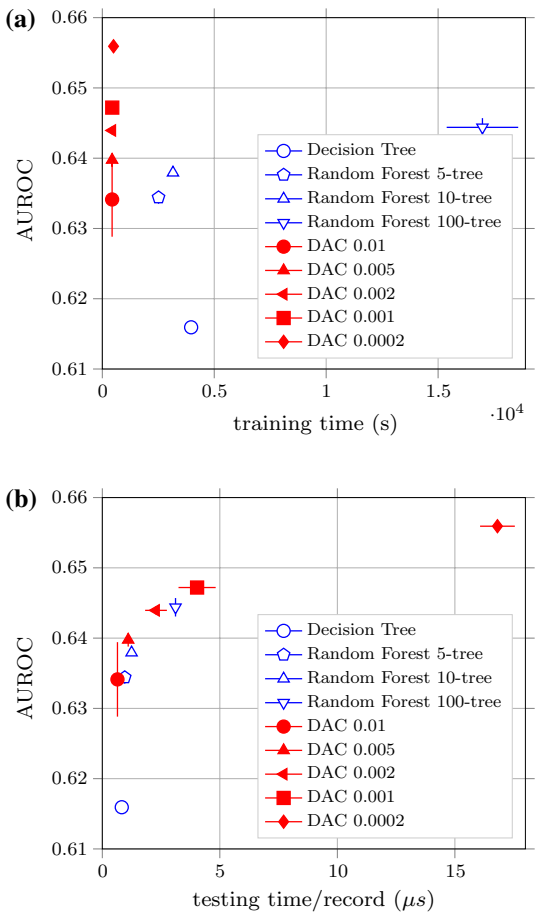


Fig. 5 DAC vs Random Forest. AUROC vs **a** training and **b** testing times, cross-validated with fivefold on the entire dataset

- The function used in the model consolidation phase, $g()$, [max/min/product]
- The minimum support threshold, [9 values from 5 to 0.01%]

for a total of 324 runs, considering all the combinations of values. $f()$ was chosen among *max*, *min* and *mean*. We tested two values for measure m, that is the confidence of the matching rules, which is a common choice in associative classifiers, and $1 - support$, following the intuition that a rule (a set of words) is the better in labeling the more is rare [20]. $g()$ was chosen among *min*, *max* and *product*, three functions that have the properties of associativity and commutativity, which are important for the distribution of the workload. We tried nine different values for the minimum support threshold, from 5 to 0.01%. The database coverage was either used or not. The minimum confidence has been set to 50%, for the rationale that any rule better than random guessing should positively contribute to the quality of the labeling. The minimum χ^2 was set to 3.841, corresponding to a p-value of 0.05 for the statistics.

We ran this session of experiments on the day 0 of the dataset, which is a 24th of the whole dataset, and without cross-validation, keeping 30% of the dataset out for testing. This reduced the execution time by more than two orders of magnitude, allowing us to test a larger selection of parameter values within some days of execution.

Database coverage The first, immediate finding was on the use of the database coverage, which did not show effects on the quality of the model trained. The amount of rules pruned by this technique has been constantly below 5%. Thus, CAP-growth is effective in selecting useful rules with limited overlapping. For example, with a minimum support of 0.1%, the number of rules of the model produced by DAC was 339, reduced to 328 with the database coverage. The training time with this technique grew instead with the number of rules, motivating us not to use it in the following experiments.

Figure 6 shows the results of the runs without the database coverage.

Function f() Choosing *min* as $f()$ (Fig. 6a) is comparable with other options only with shallow models (min sup 5%). With this support, the number of rules extracted (6) is so little that $f()$ rarely affects the voting. Decreasing the support, *min* does not show improvements, as only more confident and rare rules are being added to the models. Thus, the minimum m does not change. Both $f() = mean$ (Fig. 6c) and $f() = max$ (Fig. 6b), instead, improve their performance with a similar rate, with the top performers almost overlapping, and the top AUROC for *max* standing 0.4% above the one for *mean*.

Measure m() Preliminary experiments already led us not to choose the support itself and the χ^2 for $m()$. Confidence proves to be the best choice. Against the trend is the case where $g()$, in the consolidation function, is set to be the product of the measures. In this case $1 - support$ is the better choice for m, reaching an AUROC of 0.625 with *max* as $f()$, ranking third among all experiments but still two points below the best scenario.

Function g() As for what concerns $g()$, the function applied to two identical rules in the consolidation phase to choose the new confidence, support and χ^2, choosing either *min* or *max* is identical in this set of experiments. Choosing *product*, instead, shows contrasting outcomes. Together with the confidence as m, it never shows improvements with lower supports, reaching at most an AUROC of 60%. With $f()$ set to *min* (Fig. 6a), it has the worst quality among all the combinations, often below the AUROC of a random choice (50%). With $f()$ set to *max* (Fig. 6b) and $1 - sup$ as m, instead, as said above, it reaches the first quartiles of the results and is able to equal the AUROC of the alternatives at the lowest support.

Minimum support ith varying minimum support thresholds, from 0.02 to 0.0001, the best performing solution is stably with $f() = max$ (Fig. 6b) and $m =$ confidence, and indifferently *max* or *min* as $g()$. This, with an arbitrary choice of $g = max()$, is the solution we tested on the whole dataset and compared with the state of the art.

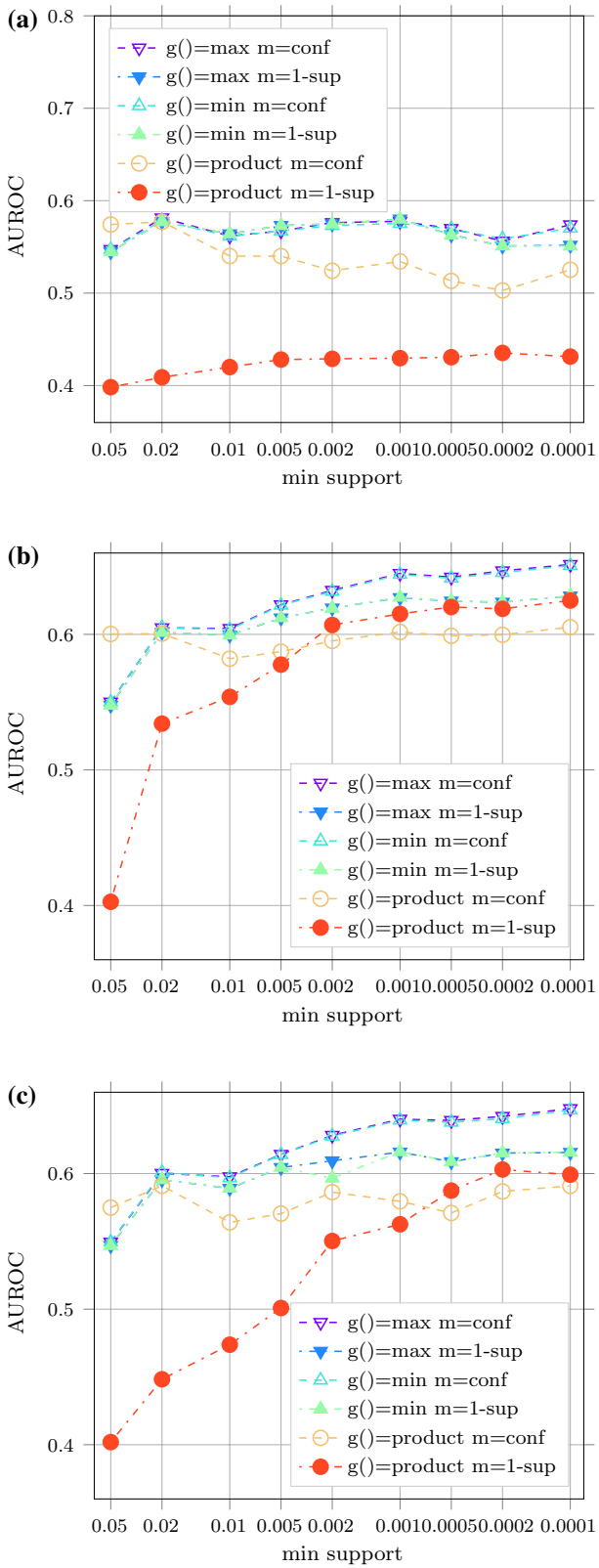


Fig. 6 DAC tuning. Comparison of different choices for *f*, *g*, *m* and *minsup*. **a** f() = min, **b** f() = max, **c** f() = mean

Model selection for Random Forest

With a Random Forest, the parameters that would affect the quality and the performance of the resulting model are mainly two, the number of trees and their depth. Similarly to the previous section, we run some preliminary tests to evaluate different choices for these parameters, on the same portion of the dataset, again without cross-validation.

We evaluated the AUROC for depths of 4, 8 and 16, starting with 10 trees and increasing their number until possible.

Figure 7 shows the results. With depth 4, the quality of the classification improves steadily until reaching a plateau after 100 trees. The execution with 170 trees repeatedly failed, raising an `OutOfMemoryError` on our testbed. With depth 8 and 10 trees, the quality improves of a not negligible point over the shallower version, and the gap still augments with more trees. Unfortunately, the `OutOfMemoryError` appears even faster, with only 60 trees. Finally, the only execution attempted with depth 16 scored 65.7%. This result was obtained in an experiment lasting more than 17 h, which would become, assuming linear scalability, more than 100 days for the tests with the complete dataset. Similarly, building any forest with depth 8 has an unfeasible expected duration. The solutions we tested on the whole dataset were thus focused on the forests with depth 4.

Experimental validation of a single instance of CAP-growth

In order to compare our work with previous works, we have evaluated a local, single-model version of DAC on a number of medium-size datasets from the UCI repository, on which results for other associative classifiers were available. The experiments showed that DAC performs similarly to CBA [6], reaching higher accuracies as often as not. Moreover, DAC reaches these results with a significantly lower number of rules, without any posterior pruning. This sets the single-model DAC as a good choice for a base model in an ensemble, where usually shallow models are preferred as baseline models.

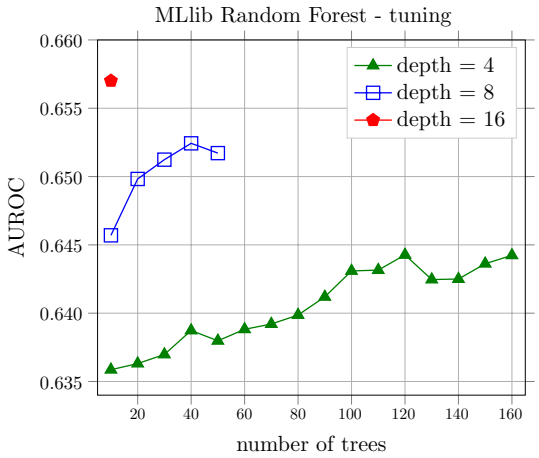


Fig. 7 Random Forest tuning. Performance (AUROC) with different parameter settings

Related work

Associative classifiers exist in a number of fashions, and a precise taxonomy has been already made in [3]. Among all, we can distinguish classifiers exploiting CARs (Class Association Rules), as introduced in [6], and others exploiting EPs (Emerging Patterns), like [21]. Our approach falls in the first category, together with works like [4, 5, 8, 9, 22–28]. Since its introduction in [6], the database coverage technique has been exploited with success by many classifiers, e.g. [8, 9, 25, 26], and several others have also exploited similar techniques, e.g. [8, 21, 29]. One of these is the redundant rule pruning, which scans again the set of rules found to delete the extensions of a rule that follow the rule itself, and that therefore are never applied [8]. These techniques have proved to be very effective in the reduction of the model and the improvement of the quality of the classifier. However, the amount of rules that are first extracted and then reordered is often enormous, demanding proportionate resources both in terms of memory and CPU. We argue that, in order to scale to very large dimensions and effectively exploit the potentials of a MapReduce-like framework, an effective associative classifier should aim at reducing, if not eliminating, the contribution of these techniques to the reduction of the model size, and focus on the extraction of a small, good quality subset of the rules.

Attempts to bring the training of an associative classifier onto a framework for parallel computing and scale to large datasets have been done in [4, 5]. The authors of [4] proposed a MapReduce solution based on a parallel implementation of FP-growth [30], modified to extract CARs, followed by two pruning phases that are slight variants of the database coverage and the above-mentioned rule pruning. This solution was implemented and tested in the Hadoop framework. The model proposed by [5] is instead an ensemble of associative classifiers. In this solution, many associative classifiers train their models in parallel on different samples of the original dataset, thus exploiting bagging. Each one of the associative classifiers exploits FP-growth to generate CARs and the database coverage for pruning. The implementation runs on Apache Spark. Both works follow the strategy of generating the complete set of CARs (for the portion of the dataset seen, in [5]) and prune in a second phase. While the general structure of our framework and the use of bagging is similar to [5], we addressed its main limitation, namely the large amount of memory used by the storage of the CARs extracted and not yet pruned.

We could not attempt a direct comparison with [4] as their code is not publicly available. Furthermore, most of the datasets used in their experiments are characterized by continuous features. Thus, the application domain is much different from the one of DAC, that is designed to work on large-scale and large-domain categorical datasets. The code from [5] is instead open-source and publicly available,[6] and we attempted a direct comparison with DAC on the Criteo dataset. Unfortunately, this algorithm cannot cope with such a very large domain, and runs out of memory in our testbed even with reduced samples of the dataset (a single day of logs).

Several works have already explored the possibility of combining more than one rule for prediction, thus defining weights akin to a score for each class. The authors of [8] have proposed to use the top K rules that match and weigh their vote with a weighted

[6] https://gitlab.com/ontic/bac.

χ^2-analysis. In [23], the top rule for each class is first determined, then the prediction is made on the one that maximizes the Laplace accuracy. Jorge and Azevedo [28] has proposed a weighted-voting based on some metrics, e.g. support, confidence and conviction of the rule. Wang and Karypis [24] sets as score the sum of the confidences for the matching rules. All these techniques have been used selecting as label the class that maximizes the defined score. The majority of associative classifiers, though, does not use a score and predicts the label with the first rule that matches the record [4, 9, 22, 25, 26].

Conclusion

In this work, we have proposed a technique to scale an associative classifier on very large datasets, namely a Distributed Associative Classifier (DAC). We considered an in-memory cluster-computing architecture, Apache Spark. In this architecture, the large availability of memory is heavily exploited to streamline the computation, avoiding disk access whenever possible, allowing an extremely faster sequential processing and caching of the intermediate results. This scheme, and the available memory, have of course their limits. In preliminary experiments, we identified the major issue of designing an associative classifier in this framework in the large number of extracted rules, which are only eventually pruned in the training phase. Therefore, we have anticipated all the pruning into a novel extraction algorithm, CAP-growth. DAC trains an ensemble model by means of bagging, which eases the distribution of the computation. Each model is generated by an instance of CAP-growth. A final consolidation phase for the models of the ensemble and a new voting strategy help further reduce the size of the model and improve the quality of the predictions.

To validate our approach, we have performed experiments in a real large-scale scenario, a binary-labeled dataset with more than 4 billion records, 800 million distinct values in its categorical features and larger than 1.2 TB in storage. The pruning done in CAP-growth has proved to be effective. When executing database coverage pruning as a final step, a negligible fraction of rules are pruned by this technique, always below 5%, without improvements in quality. DAC demonstrated better performance than a state-of-the-art technique, a Random Forest, both in terms of quality of the prediction and execution time. The best setting for DAC improves the AUROC upon the best for the Random Forest by 1% with a total training time that is 25 times smaller.

The DAC classifier, differently from a Random Forest, generates a readable model. The "hashing trick", which allows the Random Forest to deal with a large number of distinct values in the categoric fields, has the major drawback of making the model unintelligible by a human. This hampers the usability of the model for decision-making and makes also extremely difficult its debugging. DAC did not require hashing, though larger scales, i.e. billions or trillion of distinct values, might make it necessary. In this scenario, the model produced by DAC, without hashing, is made of rules containing the items exactly as they appear in the dataset, with all their semantics left intact. We believe this feature to be highly valuable for a classification model.

Future works will experiment different model generation strategies. For example, we will introduce a projection by column in the ensemble like the one implemented in Random Forests.

Authors' contributions
LV conceived the algorithm, carried out the implementation and the experiments and drafted the manuscript with input from all authors. EB and PG provided reviews on the manuscript. All authors read and approved the final manuscript.

Acknowledgements
The authors are thankful to Ernesto Valentino and Giulia Vasciaveo for having contributed to the code of DAC. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 700256 ("I-REACT" project).

Competing interests
The authors declare that they have no competing interests.

Availability of data and materials
The dataset supporting the conclusions of this article is available in the Criteo repository, http://labs.criteo.com/2013/12/download-terabyte-click-logs/.

Publisher's Note

Received: 14 September 2017 Accepted: 28 November 2017
Published online: 08 December 2017

References
1. El Houby EM, Hassan MS. Comparison between associative classification and decision tree for HCV treatment response prediction. World Acad Sci Eng Technol Int J Med Health Biomed Bioeng Pharm Eng. 2013;7(11):714–8.
2. Apiletti D, Baralis E, Cerquitelli T, Garza P, Pulvirenti F, Venturini L. Frequent itemsets mining for Big Data: a comparative analysis. Big Data Res. 2017;9:67–83.
3. Thabtah F. A review of associative classification mining. Knowl Eng Rev. 2007;22(01):37–65.
4. Bechini A, Marcelloni F, Segatori A. A MapReduce solution for associative classification of Big Data. Inf Sci. 2016;332:33–55.
5. Venturini L, Garza P, Apiletti D. BAC: a bagged associative classifier for Big Data frameworks. In: East European conference on advances in databases and information systems. Berlin: Springer; 2016. p. 137–46.
6. 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. San Francisco: AAAI Press; 1998. p. 80–6.
7. Breiman L. Some properties of splitting criteria. Mach Learn. 1996;24(1):41–7.
8. Li W, Han J, Pei J. CMAR: accurate and efficient classification based on multiple class-association rules. In: ICDM 2001, Proceedings IEEE international conference on, data mining, 2001. New York: IEEE; 2001. p. 369–76.
9. Baralis E, Chiusano S, Garza P. A lazy approach to associative classification. IEEE Trans Knowl Data Eng. 2008;20(2):156–71.
10. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, et al. MLlib: machine learning in Apache Spark. J Mach Learn Res. 2016;17(1):1235–41.
11. Landset S, Khoshgoftaar TM, Richter AN, Hasanin T. A survey of open source tools for machine learning with Big Data in the Hadoop ecosystem. J Big Data. 2015;2(1):24.
12. Singh D, Reddy CK. A survey on platforms for Big Data analytics. J Big Data. 2015;2(1):8.
13. Chapelle O, Manavoglu E, Rosales R. Simple and scalable response prediction for display advertising. ACM Trans Intell Syst Technol TIST. 2015;5(4):61.
14. Chen T, Guestrin C. Xgboost: a scalable tree boosting system. In: Proceedings of the 22Nd ACM SIGKDD international conference on knowledge discovery and data mining. New York: ACM; 2016. p. 785–94.
15. Weinberger K, Dasgupta A, Langford J, Smola A, Attenberg J. Feature hashing for large scale multitask learning. In: Proceedings of the 26th annual international conference on machine learning. New York: ACM; 2009. p. 1113–20.
16. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: machine learning in python. J Mach Learn Res. 2011;12:2825–30.
17. Liaw A, Wiener M. Classification and regression by randomForest. R News. 2002;2(3):18–22.
18. Witten IH, Frank E, Hall MA, Pal CJ. Data mining: practical machine learning tools and techniques. Burlington: Morgan Kaufmann; 2016.
19. Bradley AP. The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognit. 1997;30(7):1145–59.
20. Sebastiani F. Machine learning in automated text categorization. ACM Comput Surv. 2002;34(1):1–47.
21. Dong G, Zhang X, Wong L, Li J. CAEP: classification by aggregating emerging patterns. In: S Arikawa, Furukawa K, editors. Discovery science. Berlin: Springer; 1999. p. 30–42.
22. Chen G, Liu H, Yu L, Wei Q, Zhang X. A new approach to classification based on association rule mining. Decis Support Syst. 2006;42(2):674–89.
23. Yin X, Han J. CPAR: classification based on predictive association rules. In: Proceedings of the 2003 SIAM international conference on data mining. Philadelphia: SIAM; 2003. p. 331–5.
24. Wang J, Karypis G. HARMONY: efficiently mining the best rules for classification. In: Proceedings of the 2005 SIAM international conference on data mining. Philadelphia: SIAM; 2005. p. 205–16.

25. Thabtah F, Cowling P, Peng Y. MCAR: multi-class classification based on association rule. In: The 3rd ACS/IEEE international conference on computer systems and applications, 2005. New York: IEEE; 2005. p. 33.
26. Thabtah FA, Cowling P, Peng Y. MMAC: a new multi-class, multi-label associative classification approach. In: Fourth IEEE international conference on data mining, 2004. ICDM'04. New York: IEEE; 2004. p. 217–24.
27. Zaïane OR, Antonie ML. Classifying text documents by associating terms with text categories. In: Australian computer Science communications, vol. 24. Sydney: Australian Computer Society, Inc.; 2002. p. 215–22.
28. Jorge AM, Azevedo PJ. An experiment with association rules and classification: post-bagging and conviction. In: international conference on discovery science. Berlin: Springer; 2005. p. 137–49.
29. Xu X, Han G, Min H. A novel algorithm for associative classification of image blocks. In: The fourth international conference on computer and information technology, 2004. CIT'04. New York: IEEE; 2004. p. 46–51.
30. Li H, Wang Y, Zhang D, Zhang M, Chang EY. Pfp: parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM conference on Recommender systems. New York: ACM; 2008. p. 107–14.