Iterator-based Algorithms in Self-Tuning Discovery of Partial Implications

José L. Balcázar[1], Diego García-Sáiz[2], and Javier de la Dehesa[2]

[1] LSI Department, UPC, Campus Nord, Barcelona
jose.luis.balcazar@upc.edu
[2] Mathematics, Statistics and Computation Department, University of Cantabria
Avda. de los Castros s/n, Santander, Spain
garciasad@unican.es

Abstract. We describe the internal algorithmics of our recent implementation of a closure-based self-tuning associator: *yacaree*. This system is designed so as not to request the user to specify any threshold. In order to avoid the need of a support threshold, we introduce an algorithm that constructs closed sets in order of decreasing support; we are not aware of any similar previous algorithm. In order not to overwhelm the user with large quantities of partial implications, our system filters the output according to a recently studied lattice-closure-based notion of confidence boost, and self-adjusts the threshold for that rule quality measure as well. As a consequence, the necessary algorithmics interact in complicated ways. In order to control this interaction, we have resorted to a well-known, powerful conceptual tool, called Iterators: this notion allows us to distribute control among the various algorithms at play in a relatively simple manner, leading to a fully operative, open-source, efficient system for discovery of partial implications in relational data.

Keywords: Association mining, parameter-free mining, iterators, Python

.

1 Introduction

The task of identifying which implications hold in a given dataset has received already a great deal of attention [1]. Since [2], also the problem of identifying partial implications has been considered. Major impulse was received with the proposal of "mining association rules", a very closely related concept. A majority of existing association mining programs follow a well-established scheme [3], according to which the user provides a dataset, a support constraint, a confidence constraint, and, optionally, in most modern implementations, further constraints on other rule quality evaluation measures such as lift or leverage (a survey of quality evaluation measures for partial implications is [4]). A wealth of algorithms, of which the most famous is *apriori*, have been proposed to perform association mining.

Besides helping the algorithm to focus on hopefully useful partial implications, the support constraint has an additional role: by restricting the process to frequent (or frequent closed) itemsets, the antimonotonicity property of the support threshold defines a limited search space for exploration and avoids the often too wide space of the whole powerset of items.

Instead, however, the price becomes a burden on the user, who must supply thresholds on rule evaluation measures and on support. Rule measure thresholds may be difficult to set correctly, but at least they offer often a "semantic" interpretation that guides the choice; for instance, confidence is (the frequentist approximation to) the conditional probability of the consequent of the rule, given the antecedent, whereas lift and leverage refer to the (multiplicative or additive, respectively) deviation from independence of antecedent and consequent. But support thresholds are known to be very difficult to set right. Some smallish datasets are so dense that any exploration below 95% support, on our current technology, leads to a not always graceful breakdown of the associator program, whereas other, large but sparse datasets hardly yield any association rule unless the support is set at quantities as low as 0.1%, spanning a factor of almost one thousand; and, in order to set the "right" support threshold (whatever that means), no intuitive guidance is currently known, except for the rather trivial one of trying various supports and monitoring the number of resulting rules and the running time and memory needed.

The Weka *apriori* implementation automates partially the process, as follows: it explores repeatedly at several support levels, reducing the threshold from one run to the next by a "delta" parameter (to be set as well by the user), until a given number of rules has been gathered. Inspired by this idea, but keeping our focus in avoiding user-set parameters, we are developing an alternative association miner. It includes an algorithm that explores closed itemsets in order of decreasing support. This algorithm is similar in spirit to ChARM [5], except that some of the accelerations of that algorithm require ordering some itemsets by increasing support, which becomes inapplicable in our case. Additionally, our algorithm keeps adjusting automatically the support bound as necessary so as to be able to proceed with the exploration within the available memory resources. This is, of course, more expensive in computation time, compared to a traditional exploration with the "right" support threshold, as the number of closed frequent sets that can be filtered out right away is much smaller; on the other hand, no one can tell ahead of time which is the "right" support threshold, and our alternative spares the user the need of guessing it. To our knowledge, this is the first algorithm available for mining closed sets in order of descending support and without employing a user-fixed support threshold.

Similarly, in order to spare the user the choice of rule measure thresholds, we employ a somewhat complex (and slightly slower to evaluate) measure, the closure-based confidence boost, for which our previous work has led to useful, implementable bounds as well as to a specific form of self-tuning [6]. It can be proved that this quantity is bounded by a related, easy to compute quantity: namely, the closure-based confidence boost is always less than or equal to the

support ratio, introduced (with a less prononceable name) in [7], and defined below; this bound allows us to "push" into the closure mining process a constraint on the support ratio that spares computation of rules that will fail the rule measure threshold. We do this by postponing the consideration of the closed sets that, upon processing, would give rise only to partial implications below the confidence boost threshold.

As indicated, our algorithm self-tunes this threshold, which starts at a somewhat selective level, by lowering it in case the output rules show it appropriate. Then, the support ratio in the closure miner is to follow suit: the constraint is to be pushed into the closure mining process with the new value. This may mean that previously discarded closures are to be now considered. Therefore, we must reconcile four processes: one of mining closed frequent sets in order of decreasing support, filtering them according to their support ratio; two further ones that change, along the way, respectively, the support threshold and the support ratio threshold; and the one of obtaining the rules themselves from the closed itemsets. Unfortunately, these processes interfere very heavily with each other. Closed sets are the first objects to be mined from the dataset, and are to be processed in order of decreasing support to obtain rules from them, but they are to be processed only if they have both high enough support, and high enough support ratio. Closed sets of high support and low support ratio, however, cannot be simply discarded: a future decrease of the self-adjusting rule measure bound may require us to "fish" them back in, as a consequence of evaluations made "at the end" of the process upon evaluating rules; likewise, rules of low closure-based confidence boost need to be kept on hold instead of discarded, so as to be produced if, later, they turn out to clear the threshold after adjusting it to a lower level. The picture gains an additional complication from the fact that constructing partial implications requires not only the list of frequent closures, but also the Hasse edges that constitute the corresponding Formal Concept Lattice.

As a consequence, the varying thresholds make it difficult to organize the architecture of the software system in the traditional form of, first, mining the lattice of frequent closures and, then, extracting rules from them. We describe here how iterators offer a simple and efficient solution for the organization of our partial implication miner *yacaree*, available at SourceForge and shown at the demo track of a recent conference [8]. The details of the implementation are described here for the first time.

2 Concepts, Notation, and Overview

In our technological context (pure Python), "generators" constitute one of the ways of obtaining iterators. An iterator constructed in this way is a method (in the object-oriented sense) containing, anywere inside, the "yield" instruction; most often, this instruction is inside some loop. This instruction acts as a "return" instruction for the iterator, except that its whole status, including values of local variables and program counter, is stored, and put back into place at the next call to the method. Thus, we obtain a "lazy" method that gives us, one

by one, a sequence of values, but only computes one more value whenever it is called from the "consumer" that needs these values.

Generators as a comfortable way of constructing iterators are available only in a handful of platforms: several quite specialized lazy functional programming languages offer them, but, among the most common programming languages, only Python and C# include generators. Java or C++ offer a mere "iterator" interface that simply states that classes implementing iterators must offer, with specific names, the natural operations to iterate over them, but the notion of generators to program them easily is not available.

We move on to describe the essentials of our system, and the way iterators defined by means of generators allow us to organize, in a clear and simple way, the various processes involved.

A given set of available items U is assumed; its subsets are called itemsets. We will denote itemsets by capital letters from the end of the alphabet, and use juxtaposition to denote union of itemsets, as in XY. The inclusion sign as in $X \subset Y$ denotes proper subset, whereas improper inclusion is denoted $X \subseteq Y$. For a given dataset D, consisting of n transactions, each of which is an itemset labeled with a unique transaction identifier, we define the support $sup(X)$ of an itemset X as the cardinality of the set of transactions that contain X. Sometimes, the support is measured "normalized" by dividing by the dataset size; then, it is an empirical approximation to the probability of the event that the itemset appears in a "random" transaction. Except where explicitly indicated, all our uses of support will take the form of ratios, and, therefore, it does not matter at all whether they come absolute or normalized.

An association rule is an ordered pair of itemsets, often written $X \to Y$. The confidence $c(X \to Y)$ of rule $X \to Y$ is $sup(XY)/sup(X)$. We will refer occasionally below to a popular measure of deviation from independence, often named *lift*: assuming $X \cap Y = \emptyset$, the lift of $X \to Y$ is

$$\frac{sup(XY)}{sup(X)\,sup(Y)}$$

where all three supports are assumed normalized (if they are not, then the dataset size must of course appear as an extra factor in the numerator).

An itemset X is called frequent if its support is greater than or equal to some user-defined threshold: $sup(X) > \tau$. We often assume that τ is known; no support bound is implemented by setting $\tau = 0$. Our algorithms will attempt at self-tuning τ to an appropriate value without concourse of the user. Given an itemset $X \subseteq U$, its closure $\overline{X}$ of X is the maximal set (with respect to set inclusion) $Y \subseteq U$ such that $X \subseteq Y$ and $sup(X) = sup(Y)$. It is easy to see that $\overline{X}$ is unique. An itemset X is closed if $\overline{X} = X$. Closure operators are characterized by the three properties of monotonicity, idempotency, and extensivity.

The support ratio was essentially employed first, to our knowledge, in [7], where, together with other similar quotients, it was introduced with the aim of providing a faster algorithm for computing representative rules. The support

ratio of an association rule $X \to Y$ is that of the itemset XY, defined as follows:

$$\sigma(X \to Y) = \sigma(XY) = \frac{sup(XY)}{\max\{sup(Z) \mid sup(Z) > \tau,\, XY \subset Z\}}.$$

For many quality measures for partial implications, including support, confidence, and closure-based confidence boost (to be defined momentarily), the relevant supports turn out to be the support of the antecedent and the support of the union of antecedent and consequent. As these are captured by the corresponding closures, we deem inequivalent two rules $X \to Y$ and $X' \to Y'$ exactly when they are not "mutually redundant" with respect to the closure space defined by the dataset: either $\overline{X} \neq \overline{X'}$, or $\overline{XY} \neq \overline{X'Y'}$. We denote that fact as $(X \to Y) \not\equiv (X' \to Y')$.

We now assume $sup(XY) > \tau$. As indicated, our system keeps a varying threshold on the following rule evaluation measure: $\beta(X \to Y) =$

$$\frac{c(X \to Y)}{\max\{c(X' \to Y') \mid (X \to Y) \not\equiv (X' \to Y'),\, sup(X'Y') > \tau,\, X' \subseteq \overline{X},\, Y \subseteq \overline{X'Y'}}.$$

This notion, known as "closure-based confidence boost", as well as the "plain confidence boost", which is a simpler variant where the closure operator reduces to the identity, are studied in depth in [6]. Intuitively, this is a relative, instead of absolute, form of confidence: we are less interested in a partial implication having very similar confidence to that of a simpler one. A related formula measures relative confidence with respect to logically stronger partial implications (confidence width, see [6]); the formula just given seems to work better in practice. For the value of this measure to be nontrivial, XY must be a closed set; the following inequality holds:

Proposition 1. $\beta(X \to Y) \leq \sigma(X \to Y)$.

The threshold on $\beta(X \to Y)$ is self-adjusted along the mining process, on the basis of several properties such as coincidence with lift under certain conditions; all these details and properties are described in [6].

2.1 Architecture of *yacaree*

The diagram in Figure 1 shows the essentials of the class structure, for easier reference along the description of the iterators. For simplicity, a number of additional classes existing in the system are not shown. A couple of them, added recently, find minimal generators via standard means and implement a plain confidence boost version appropriate for full-confidence implications; their algorithmics are not novel, pose no challenge, and are omitted here. We are also omitting discussion of classes like the Dataset class, some heap-like auxiliary data structures, user interfaces, and a class capturing a few static values, as their role in our description is minor or easy to understand (or both).

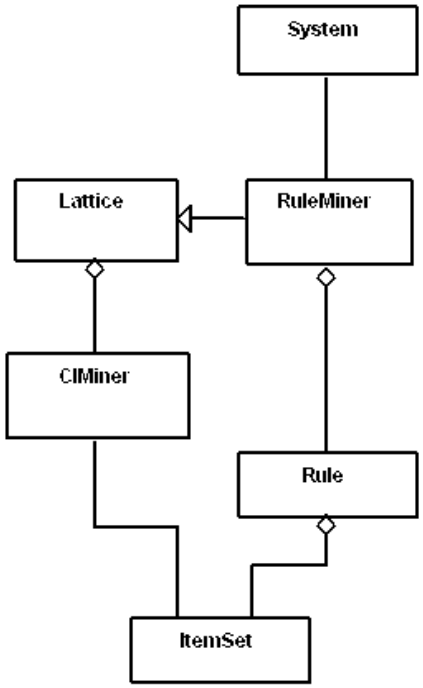

Fig. 1. Partial class diagram of the associator

2.2 Class Overview

We give a brief explanation of the roles of the classes given in the diagram. Details about their main methods (the corresponding iterators) come below.

Class ItemSet keeps the information and methods to prettyprint itemsets, including information such as support; it inherits from sets all set-theoretic operations. Class Rule keeps both antecedent and consequent (technically, it keeps the antecedent and the union of antecedent and consequent, as in this case the latter is always closed, which allows for more efficient processing), and is able to provide rule evaluation measures such as confidence or lift.

Class ClMiner runs the actual closure mining, with some auxiliary methods to handle all details. Its main method is the iterator mine_closures() (described below) which yields, one by one and upon being called, all closed sets having support above the threshold, in order of decreasing support. This "decreasing support" condition allows us to increase the support threshold, if necessary, to continue the exploration. As explained below, when the internal data structures of the closure miner are about to overflow, the support threshold is increased in such a way that half the closures found so far and still pending consideration are discarded.

Class Lattice runs its own iterator, candidate_closures(), which, in turn, calls mine_closures() as new closed sets become needed. Its main task is to call methods that implement the algorithms from [9] and actually build the lattice of closed sets, so that further iterators can expect to receive closures for which the immediate predecessors have been identified. Version 1.0 of *yacaree* employed the Border algorithm but in version 1.1 we have implemented the faster algorithm iPred and indeed obtained around a 10% acceleration. The fact that iPred could be employed in enumerations of closures by decreasing support was proved in [10]. See [11] for futher discussions.

Additionally, the support ratio of each closed set is also computed here, and the class offers yet another iterator that provides, for each closure, all the predecessor closures having support above a local, additional support threshold that can be specified at call time. In this way, we obtain all the candidate antecedents for a given closed set as potential consequent. This internal iterator amounts to a plain depth-first search, so that we do not discuss it further here.

Within class Lattice, two heap-structured lists keep, respectively, the closures that are ready to be passed on as they clear both the support and the support ratio thresholds (Lattice.ready) and the closures that clear the support threshold but fail the support ratio threshold (Lattice.freezer); these will be recovered in case a decrease of the confidence boost bound is to affect the support ratio pruning.

Finally, class RuleMiner is in charge of offering the system an iterator over all the association rules passing the current thresholds of support and closure-based confidence boost: mine_rules(). Its usage allows one to include easily further checks of confidence, lift, or any other such quantity.

3 Details

This section provides details of the main iterators and their combined use to attain our goals.

3.1 ClMiner.mine_closures()

The closure miner is the iterator that supports the whole scheme; it follows a "best-first" strategy, where here "best" means "highest support". We maintain a heap containing closures already generated, but which have not been expanded yet to generate further closures after them. The heap can provide the one of highest support in logarithmic time, as this is the closure that comes next. Then, as this closure is passed on to the lattice constructor, items (rather, closures of singletons) are added to it in all possible ways, and closure operations are applied in order to generate its closed successors, which are added to the heap unless they were already in it. The decreasing support condition ensures that they were never visited before. For simplicity, we omit discussing the particular case of the empty set, which, if closed, is to be traversed first, separately.

1: identify closures of singletons

```
2: organize them into a maxheap according to support
3: while heap nonempty do
4:   consider increasing the support threshold by monitoring the available
     memory
5:   if the support threshold must be raised then
6:     kill from the heap pending closures of support below new threshold,
       which is chosen so that the size of the heap halves
7:   end if
8:   pop from heap the max-support itemset
9:   yield it
10:  try to extend it with all singleton closures
11:  for such extensions with sufficient support do
12:    if their closure is new then
13:      add it to the heap
14:    end if
15:  end for
16: end while
```

In order to clarify how this algorithm works, we develop the following example. Consider a dataset with 24 transactions over universe $U = \{a, b, c, d, e\}$ of 5 items: $\{abcde, bcde \times 2, abe, cde, be, ae \times 3, ab \times 4, cd \times 6, b \times 2, a \times 3\}$. For this dataset, there are 12 closed sets, which we enumerate here with their corresponding supports: $\emptyset_{/24}$, $a_{/12}$, $b_{/11}$, $cd_{/10}$, $e_{/9}$, $ab_{/6}$, $ae_{/5}$, $be_{/5}$, $cde_{/4}$, $bcde_{/3}$, $abe_{/2}$, $abcde_{/1}$. The empty set is treated first, separately, as indicated. Then, the four next closures correspond to closures of singletons (the closures of c and d coincide) and form an initial heap, containing: $[a_{/12}, b_{/11}, cd_{/10}, e_{/9}]$.

The heap provides a as next closure in descending support; it is passed on to further processing at the "yield" instruction, and it is expanded with singleton closures in all possible ways, enlarging the heap into containing six pending closures: $[b_{/11}, cd_{/10}, e_{/9}, ab_{/6}, ae_{/5}, abcde_{/1}]$: each of the new sets in the heap is obtained by adding to a the closure of a singleton, and closing the result. The next closure is b, which goes into the "yield" instruction and, subsequently, generates two further closures to be added to the heap, which becomes: $[cd_{/10}, e_{/9}, ab_{/6}, ae_{/5}, be_{/5}, bcde_{/3}, abcde_{/1}]$. The closure ab generated from b is omitted, as it is repeated since it was already generated from a.

For illustration purposes, we assume now that the length of the heap, currently 7, is deemed too large. Of course, in a toy example like this one there is no need of moving up the support threshold, but let's do it anyway: assume that the test indicates that the heap is occupying too much memory, incurring in a risk of soon-coming overflow. Then, the support is raised as much as necessary so as to halve the length of the heap. Pending closures of support 5 or less would be discarded from the heap, the support threshold would be set at 6, and only three closures would remain in the heap: $[cd_{/10}, e_{/9}, ab_{/6}]$. Each of them would be processed in turn, given onwards by the "yield" instruction, and expanded with all closures of singletons; in all cases, we will find that expanding any of them with a singleton closure leads to a closure of support below 6, which is therefore

omitted as it does not clear the threshold. Eventually, these three closures in the heap are passed on, and the iterator will have traversed all closures of support 6 or higher.

As a different run, assume now that we accept the growth of the heap, so that it is not reduced. The traversal of closures would go on yielding cd, which would add cde to the heap; adding either a or b to cd leads to $abcde$ which is already in the heap. The next closure e adds nothing new to the heap, and the next is ab which adds abe; at this point the heap is $[ae_{/5}, be_{/5}, cde_{/4}, bcde_{/3}, abe_{/2}, abcde_{/1}]$. All further extensions only lead to repeated closures, hence nothing is further added to the heap, and, as it is emptied, the traversal of all the closures is completed.

The main property that has to be argued here is the following:

Proposition 2. *As the support threshold rises, all the closed sets delivered so far by the iterator to the next phase are still correct, that is, have support at least the new value of the threshold.*

Proof. We prove this statement by arguing the following invariants: first, the new threshold is bounded above by the highest support of a closure in the heap; second, all the closed sets provided so far up to any "yield" statement have support bounded below by all the supports currently in the heap. These invariants are maintained as we extract the closure C of maximum support in the heap, and also when we add to it extensions of C: indeed, C having been freshly taken off the heap, all previous deliveries have at least the same support, whereas all extensions that are to enter the heap are closed supersets of C and must have lower support, because C is closed.

Hence, all previous deliveries have support higher than the maximum support in the heap, which, in turn, is also higher than the new threshold; transitivity now proves the statement.

3.2 Lattice.candidate_closures()

In order to actually mine rules from the closures traversed by the loop described in the previous section, further information is necessary: data structures to allow for traversing predecessors, namely, the Hasse edges, that is, the immediate, nontransitive neighbors of each closure. These come from a second iterator that implements the iPred algorithm [9].

Additionally, we wish to push into the closure mining the confidence boost constraint. The way to do it is to compute the support ratio of each closure, and only pass it on to mine rules from it if this support ratio is above the confidence boost threshold; indeed, Proposition 1 tells us that, if the support ratio is below the threshold, the confidence boost will be too.

Due to the condition of decreasing support, we know that the closed superset that defines the support ratio is exactly the first successor to appear from the closure mining iterator. As soon as one successor of C appears, if the support ratio is high enough, we can yield C, as the Hasse edges to its own predecessors

are guaranteed to have been set before. If the support ratio is not enough, it is kept on a "freezer" (again a heap-like structure) from where it might be "fished back in" if the confidence boost threshold decreases later on.

One has to be careful that the same closed set, say C, may be the first successor of more than one predecessor. As we set the predecessors C' of C, we move to the "ready" heap those that have C as first successor, if their support ratio is high enough; then we yield them all. Additionally, as we shall see, it may happen that RuleMiner.mine_rules() moves closures from "freezer" to "ready". We explain this below.

```
1: for each closed set C yielded by ClMiner.mine_closures() do
2:    apply a Hasse edges algorithm (namely iPred) to set up the lattice edges
      connecting C to its predecessors
3:    for each unprocessed predecessor C' do
4:       compute the support ratio of C'
5:       if support ratio is over the rule evaluation threshold then
6:          add C' to the "ready" heap
7:       else
8:          add C' to the "freezer" heap
9:       end if
10:   end for
11:   for each closure in the "ready" heap do
12:      yield it
13:   end for
14: end for
```

We observe here that we are not guaranteeing decreasing support order in this iterator, as the changes to the support ratio threshold may swap closures with respect to the order in which they were obtained. What we do need is that the most basic iterator, ClMiner.mine_closures(), does provide them in decreasing support order, first, to ensure that the support threshold can be raised if necessary, and, second, to make sure that the support ratio is correctly computed from the first successor found for each closure.

Along the same example as before, consider, for instance, what happens when ClMiner.mine_closures() yields the closure ab to Lattice.candidate_closures(). The iPred algorithm identifies a and b as immediate predecessors of ab, and the corresponding Hasse edges are stored. Then, both a and b are identified as closures whose first successor (in decreasing support) has just appeared; indeed, other successors have less support than ab. The support ratios of a and b, namely, $12/6 = 2$ and $11/6$, are seen to be higher than the confidence boost threshold (which starts at 1.15 by default) and both a and b are moved to the "ready" heap and yielded to the subsequent rule mining phase. On the other hand, if the confidence boost threshold was, say, at 1.4, upon processing $bcde$ we would find $4/3 < 1.4$ as support ratio of cde, and this closure would wait in the freezer heap, until (if at all) a revised lower value of the confidence boost threshold would let it through, by moving it from the freezer queue to the ready queue.

3.3 RuleMiner.mine_rules()

In the class RuleMiner, which inherits from Lattice, the iterator mine_rules() relies on the closures provided by the previous iterator in the pipeline:

```
1: reserved_rules = [ ]
2: for each closure from candidate_closures() do
3:   for each predecessor having high enough support so as to reach the confidence threshold do
4:     form a rule r with the predecessor as antecedent and the closure as consequent
5:     use it to revise the closure-based confidence boost threshold
6:     if threshold decreased then
7:       move from Lattice.freezer to Lattice.ready those closures whose support ratio now passes the new threshold
8:       for each rule in reserved_rules do
9:         if its closure-based confidence boost threshold passes the threshold then
10:          yield it
11:        else
12:          keep it in reserved_rules
13:        end if
14:      end for
15:    end if
16:    if the closure-based confidence boost of r passes the threshold then
17:      yield r
18:    else
19:      keep it in reserved_rules
20:    end if
21:  end for
22: end for
```

Each closure makes available an iterator over its predecessors in the closures lattice (closed proper subsets), up to a given support level that we can specify upon calling it. For instance, at the closure *bcde*, of support 3, and assuming a confidence threshold of 0.6, we would explore predecessors *be* and *cde*, which lead to rules $be \to cd$ and $cde \to b$. The confidence boost has to be checked, but part of the task is already made since the very fact that the closure *bcde* arrived here implies that its support ratio is over the confidence boost threshold. In this case, the support ratio of closure *bcde* is 3. We must test confidences with smaller antecedents (see [6]). As the confidences of $b \to cd$ and $e \to cd$ are low enough, the rule $be \to cd$ becomes indeed reported; $cde \to b$ does as well, after checking how low the confidences of $cd \to b$ and $e \to b$ are.

The revision of the closure-based confidence boost threshold can be done in a number of ways. The current implementation keeps computing the lift of those rules whose antecedent is a singleton, as the condition on support ratio ensures that, in this case, it will coincide with the confidence boost [6]; these lift values enter a weighted average with the current threshold, and, if the average is

sufficiently smaller, the threshold is decreased. Only a partial justification exists so far for this choice.

When the threshold for confidence boost decreases, closures whose support ratio was too low may become now high enough; thus, the freezer is explored and closures whose support ratio is now above the new confidence boost threshold are moved into the ready queue (lines 6 and 7), to be processed subsequently.

3.4 System

The main program simply traverses all rules, as obtained from the iterator mine_rules(), in the class RuleMiner:

```
1: for each rule in RuleMiner.mine_rules() do
2:     account for it
3: end for
```

What is to be done with each rule depends on the instructions from the user interface, but usually we count how many of them are obtained and we write them all on disk, maybe up to a fixed limit on the number of rules (that can be modified by editing the source code). In this case, we report those of highest closure-based confidence boost.

4 A Second Implementation

With a view to offering this system in a more widespread manner, we have developed a joint project with KNIME GmbH, a small company that develops the open source data mining suite KNIME. This data mining suite is implemented in Java. Hence, we have constructed a second implementation in Java.

However, the issue is not fully trivial because of two main reasons. The first is that the notion of iterator in Java is different from that in Python, and is not obtained from generators: the "yield" instruction, which saves the state of an iteration at the time it is invoked, does not exist in Java, which simply declares that hasNext() and next() methods must be made available: respectively, to know whether there are more elements to process and to get the next element. A second significative change is that the memory control to ensure that the list of pending closures does not overflow has to be made in terms of the memory management API of KNIME, and requires one extra loop to check whether the decrease in memory usage was sufficient.

Therefore, we have to use the Iterator class to "copy", to the extent possible, the "yield" behavior, saving all necessary information to continue in queues and lists. The three most affected methods for this issue are, of course, mine_rules(), candidate_closures() and mine_closures(). We describe here only mine_rules().

In this case, a queue, called ready_rules, is needed in order to store the rules that are built from the current closure among the candidates and have achieved the support, confidence, and confidence boost requirements. Rules that do not clear these thresholds are stored in another queue, reserved_rules, as in the Python implementation. The code is shown next:

```
reserved_rules = empty queue of rules
ready_rules = empty queue of rules
ready_rules_iterator = iterator for ready_rules
while !ready_rules_iterator.hasNext() do
  for each closure from candidate_closures() do
    for each predecessor having high enough support so as to reach the
    confidence threshold do
      form a rule r with the predecessor as antecedent and the closure as
      consequent
      use it to revise the threshold for the rule evaluation measure
      if threshold decreased then
        move from Lattice.freezer to Lattice.ready those closures whose
        support ratio now passes the new threshold
        for each rule in reserved_rules do
          if its rule measure passes the new threshold then
            store it in ready_rules
          else
            keep it in reserved_rules
          end if
        end for
      end if
      if the rule measure of r passes the threshold then
        store it in ready_rules
      else
        keep it in reserved_rules
      end if
    end for
  end for
end while
return ready_rules
```

In Lattices.candidate_closures(), the candidate closures are likewise stored in a list called cadidate_closures_list in order that mine_rules method can obtain them. The program is constructed in the same way as the one just described, and is omitted here.

The last method that needs a change in the translation from Python to Java and KNIME is clminer.mine_closures(), and it consists of storing in a list called max-support_itemset_list the candidate itemsets that obey the max-support requirement, and of returning this list at the end of the method. In this case iterators aren't needed beacuse in this method is only required to store and return the list, so next() and hastNext() methods are not used.

```
max-support_itemset_list = empty list of itemset
identify closures of singletons
organize them into a maxheap according to support
while heap nonempty do
```

```
    consider increasing the support threshold by monitoring the available memory
    if the support threshold must be raised then
      kill from the heap pending closures of support below new threshold, which is chosen so that the size of the heap halves
    end if
    pop from heap the max-support itemset and store it in max-support_itemset_list
    try to extend it with all singleton closures
    for such extensions with sufficient support do
      if their closure is new then
        add it to the heap
      end if
    end for
    return max-support_itemset_list
end while
```

5 Conclusion

We have studied a variant of the basic association mining process. In our variant, we try to avoid burdening the user with requests to fix threshold parameters. We keep an internal support threshold and adjust it upwards whenever the computation process shows that the system will be unable to run down to the current threshold value. We tackle the problem of limiting the number of rules through one specific rule measure, closure-based confidence boost, for which the threshold is self-adjusted along the mining process. A minor detail is that for full-confidence implications it is not difficult to see that closure-based confidence boost is inappropriate, and plain confidence boost is to be used. Further details about this issue will be given in future work.

The confidence boost constraint is pushed into the mining process through its connection to the support ratio. Therefore, the closure miner has to coordinate with processes that move upwards the support threshold, or downwards the support ratio threshold.

Further study on the basis of our implementation is underway, and further versions of our association miner, with hopefully faster algorithmics, will be provided in the coming months. Another line of activity is as follows: granted that our approach offers partial implications without user-defined parameters, to what extent users that are *not* experts in data analysis are satisfied with the results? Our research group explores that topic in a separate paper [12].

Additionally, we are aware of two independent works where an algorithm is proposed to traverse the closure space in linear time [13], [14]; these algorithms do *not* follow an order of decreasing support, and we find nontrivial to modify them so that they fulfill this condition. Our research group is attempting at it, as, if successful, faster implementations could be designed.

References

1. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag (1999)
2. Luxenburger, M.: Implications partielles dans un contexte. Mathématiques et Sciences Humaines **29** (1991) 35–55
3. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining. AAAI/MIT Press (1996) 307–328
4. Geng, L., Hamilton, H.J.: Interestingness measures for data mining: A survey. ACM Comput. Surv. **38**(3) (2006)
5. Zaki, M.J., Hsiao, C.J.: Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Transactions on Knowledge and Data Engineering **17**(4) (2005) 462–478
6. Balcázar, J.L.: Formal and computational properties of the confidence boost in association rules. Available at: [http://personales.unican.es/balcazarjl]. Extended abstract appeared as "Objective novelty of association rules: Measuring the confidence boost. In Yahia, S.B., Petit, J.M., eds.: EGC. Volume RNTI-E-19 of Revue des Nouvelles Technologies de lInformation., Cepadu'es-Editions (2010) 297-302" (2010)
7. Kryszkiewicz, M.: Closed set based discovery of representative association rules. In Hoffmann, F., Hand, D.J., Adams, N.M., Fisher, D.H., Guimarães, G., eds.: Proc. of the 4th International Symposium on Intelligent Data Analysis (IDA). Volume 2189 of Lecture Notes in Computer Science., Springer-Verlag (2001) 350–359
8. Balcázar, J.L.: Parameter-free association rule mining with yacaree. In Khenchaf, A., Poncelet, P., eds.: EGC. Volume RNTI-E-20 of Revue des Nouvelles Technologies de l'Information., Hermann-Éditions (2011) 251–254
9. Baixeries, J., Szathmary, L., Valtchev, P., Godin, R.: Yet a faster algorithm for building the Hasse diagram of a concept lattice. In Ferré, S., Rudolph, S., eds.: Proc. of the 7th International Conference on Formal Concept Analysis (ICFCA). Volume 5548 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2009) 162–177
10. Balcázar, J.L., Tîrnăucă, C.: Border algorithms for computing Hasse diagrams of arbitrary lattices. In Valtchev, P., Jäschke, R., eds.: ICFCA. Volume 6628 of Lecture Notes in Computer Science., Springer (2011) 49–64
11. Kuznetsov, S.O., Obiedkov, S.A.: Algorithms for the construction of concept lattices and their diagram graphs. In Raedt, L.D., Siebes, A., eds.: Proc. of the 5th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD). Volume 2168 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2001) 289–300
12. García-Sáiz, D., Zorrilla, M., Balcázar, J.L.: Closures and partial implications in educational data mining. ICFCA, Supplementary proceedings (2012)
13. Ganter, B.: Two basic algorithms in concept analysis (preprint 1987). In Kwuida, L., Sertkaya, B., eds.: ICFCA. Volume 5986 of Lecture Notes in Computer Science., Springer (2010) 312–340
14. Uno, T., Asai, T., Uchida, Y., Arimura, H.: An efficient algorithm for enumerating closed patterns in transaction databases. In Suzuki, E., Arikawa, S., eds.: Discovery Science. Volume 3245 of Lecture Notes in Computer Science., Springer (2004) 16–31