Mining Fault-tolerant Item Sets using Subset Size Occurrence Distributions

Christian Borgelt[1] and Tobias Kötter[2]

[1] European Centre for Soft Computing
c/ Gonzalo Gutiérrez Quirós s/n, E-33600 Mieres (Asturias), Spain
christian.borgelt@softcomputing.es
[2] Dept. of Computer Science, University of Konstanz
Box 712, D-78457 Konstanz, Germany
tobias.koetter@uni-konstanz.de

Abstract. Mining fault-tolerant (or approximate or fuzzy) item sets means to allow for errors in the underlying transaction data in the sense that actually present items may not be recorded due to noise or measurement errors. In order to cope with such missing items, transactions that do not contain all items of a given set are still allowed to support it. However, either the number of missing items must be limited, or the transaction's contribution to the item set's support is reduced in proportion to the number of missing items, or both. In this paper we present an algorithm that efficiently computes the subset size occurrence distribution of item sets, evaluates this distribution to find fault-tolerant item sets, and exploits intermediate data to remove pseudo (or spurious) item sets. We demonstrate the usefulness of our algorithm by applying it to a concept detection task on the 2008/2009 Wikipedia Selection for schools.

1 Introduction and Motivation

In many applications of frequent item set mining one faces the problem that the transaction data to analyze is imperfect: items that are actually contained in a transaction are not recorded as such. The reasons can be manifold, ranging from noise through measurement errors to an underlying feature of the observed process. For instance, in gene expression analysis, where one may try to find co-expressed genes with frequent item set mining [9], binary transaction data is often obtained by thresholding originally continuous data, which are easily affected by noise in the experimental setup or limitations of the measurement devices. Analyzing alarm sequences in telecommunication data for frequent episodes can be affected by alarms being delayed or dropped due to the fault causing the alarm also affecting the transmission system [18]. In neurobiology, where one searches for assemblies of neurons in parallel spike trains with the help of frequent item set mining [11, 4], ensemble neurons are expected to participate in synchronous activity only with a certain probability. In this paper we present a new algorithm to cope with this problem, which efficiently computes the subset size occurrence distribution of item sets, evaluates this distribution to find fault-tolerant item sets, and uses intermediate data to remove pseudo (or spurious) item sets.

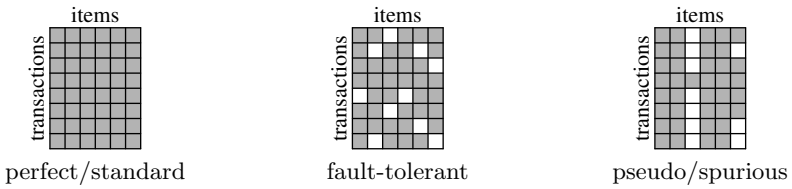

Fig. 1. Different types of item sets illustrated as binary matrices.

The rest of this paper is organized as follows: in Section 2 we review the task of fault-tolerant item set mining and some approaches to this task. In Section 3 we describe how our algorithm traverses the search space and how it efficiently computes the subset size occurrence distribution for each item set it visits. In Section 4 we discuss how the intermediate/auxiliary data that is available in our algorithm can be used to easily cull pseudo (or spurious) item sets. In Section 5 we compare our algorithm to two other algorithms that fall into the same category, and for certain specific cases can be made to find the exact same item sets. In addition, we apply it to a concept detection task on the 2008/2009 Wikipedia Selection for Schools to demonstrate its practical usefulness. Finally, in Section 6 we draw conclusions and point out possible future work.

2 Fault-Tolerant or Approximate Item Set Mining

In standard frequent item set mining only transactions that contain *all* of the items in a given set are counted as supporting this set. In contrast to this, in fault-tolerant item set mining transactions that contain only a subset of the items can still support an item set, though possibly to a lesser degree than transactions containing all items. Based on the illustration of these situations shown on the left and in the middle of Figure 1, fault-tolerant item set mining has also been described as finding almost pure (geometric or combinatorial) tiles in a binary matrix that indicates which items are contained in which transactions [10].

In order to cope with missing items in the transaction data to analyze, several fault-tolerant (or approximate or fuzzy) frequent item set mining approaches have been proposed. They can be categorized roughly into three classes: (1) error-based approaches, (2) density-based approaches, and (3) cost-based approaches.

Error-based Approaches. Examples of error-based approaches are [15] and [3]. In the former the standard support measure is replaced by a fault-tolerant support, which allows for a maximum number of missing items in the supporting transactions, thus ensuring that the measure is still anti-monotone. The search algorithm itself is derived from the famous Apriori algorithm [2]. In [3] constraints are placed on the number of missing items as well as on the number of (supporting) transactions that do not contain an item in the set. Hence it is related to the tile-finding approach in [10]. However, it uses an enumeration search scheme that traverses sub-lattices of items and transactions, thus ensuring a complete search, while [10] relies on a heuristic scheme.

Density-based Approaches. Rather than fixing a maximum *number* of missing items, density-based approaches allow a certain *fraction* of the items in a set to be missing from the transactions, thus requiring the corresponding binary matrix tile to have a minimum density. This means that for larger item sets more items are allowed to be missing than for smaller item sets. As a consequence, the measure is no longer anti-monotone if the density requirement is to be fulfilled by each individual transaction. To overcome this [19] require only that the average density over all supporting transaction must exceed a user-specified threshold, while [17] define a recursive measure for the density of an item set.

Cost-based Approaches. In error- or density-based approaches all transactions that satisfy the constraints contribute equally to the support of an item set, regardless of how many items of the set they contain. In contrast to this, cost-based approaches define the contribution of transactions in proportion to the number of missing items. In [18, 5] this is achieved by means of user-provided item-specific costs or penalties, with which missing items can be inserted. These costs are combined with each other and with the initial transaction weight of 1 with the help of a t-norm. In addition, a minimum weight for a transaction can be specified, by which the number of insertions can be limited.

Note that the cost-based approaches can be made to contain the error-based approaches as a limiting or extreme case, since one may set the cost/penalty of inserting an item in such a way that the transaction weight is not reduced. In this case limiting the number of insertions obviously has the same effect as allowing for a maximum number of missing items.

The approach presented in this paper falls into the category of cost-based approaches, since it reduces the support contribution of transactions that do not contain all items of a considered item set. How much the contribution is reduced and how many missing items are allowed can be controlled directly by a user. However, it treats all items the same, while the cost-based approaches reviewed above allow for item-specific penalties. Its advantages are that, depending on the data set, it can be faster, admits more sophisticated support/evaluation functions, and allows for a simple filtering of pseudo (or spurious) item sets.

In pseudo (or spurious) item sets a subset of the items co-occur in many transactions, while the remaining items do not occur in any (or only very few) of the (fault-tolerantly) supporting transactions (illustrated in Figure 1 on the right; note the regular pattern of missing items compared to the middle diagram). Despite the ensuing reduction of the weight of the transactions (due to the missing items), the item set support still exceeds the user-specified threshold. Obviously, such item sets are not useful and should be discarded by requiring, for instance, a minimum fraction of supporting transactions per item. This is easy in our algorithm, but difficult in the cost-based approaches reviewed above.

Finally note that a closely related setting is the case of uncertain transactional data, where each item is endowed with a transaction-specific weight or probability, which indicates the degree or chance with which it is a member of the transaction. Approaches to this related, but nevertheless fundamentally different problem, which we do not consider here, can be found in [8, 13, 7].

```
global variables:                          (may also be passed down in recursion)
lists : array of array of integer;         (* transaction identifier lists *)
cnts : array of integer;                   (* item counters, one per transaction *)
dist : array of integer;                   (* subset size occurrence distribution *)
iset : set of integer;                     (* current item set *)
e_min : real;                              (* minimum evaluation of an item set *)

procedure sodim (n: integer);              (* n: number of selectable items *)
var i : integer;                           (* loop variable *)
    t : array of integer;                  (* to access the transaction id lists *)
    e : real;                              (* item set evaluation result *)
begin
  while n > 0 do begin                     (* while there are items left *)
    n := n − 1; t := lists[n];             (* get the next item and its trans. ids *)
    for i := 0 upto length(t)-1 do begin   (* traverse the transaction ids *)
      inc(cnts[t[i]]);                     (* increment the item counter and *)
      inc(dist[cnts[t[i]]]);               (* the subset size occurrences, *)
    end;                                   (* i.e., update the distribution *)
    e := eval(dist, length(iset)+1);       (* evaluate subset size occurrence distrib. *)
    if e ≥ e_min then begin                (* if the current item set qualifies *)
      add(iset, n);                        (* add current item to the set *)
      ⟨ report the current item set iset ⟩;
      sodim(n);                            (* recursively check supersets *)
      remove(iset, n);                     (* remove current item from the set, *)
    end;                                   (* i.e., restore the original item set *)
    for i := 0 upto length(t)-1 do begin   (* traverse the transaction ids *)
      dec(dist[cnts[t[i]]]);               (* decrement the subset size occurrences *)
      dec(cnts[t[i]]);                     (* and then the item counter, *)
    end;                                   (* i.e., restore the original distribution *)
  end;
end;   (* end of sodim() *)
```

Fig. 2. Simplified pseudo-code of the recursive search procedure.

3 Subset Size Occurrence Distribution

The basic idea of our algorithm is to compute, for each visited item set, how many transactions contain subsets with 1, 2, ..., k items, where k is the size of the considered item set. We call this the *subset size occurrence distribution* of the item set, as it states how often subsets of different sizes occur. This distribution is evaluated by a function that combines, in a weighted manner, the entries which refer to subsets of a user-specified minimum size (and thus correspond to a maximum number of missing items). Item sets that reach a user-specified minimum value for the evaluation measure are reported.

Computing the subset size occurrence distribution is surprisingly easy with the help of an intermediate array that records for each transaction how many of the items in the currently considered set are contained in it. In the search, which is a standard depth-first search in the subset lattice that can also be seen as a

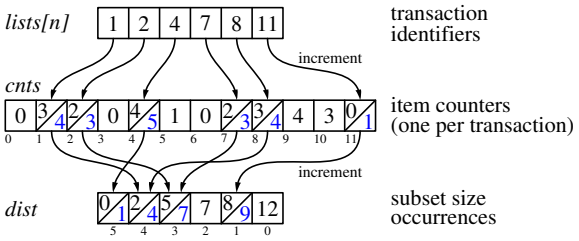

Fig. 3. Updating the subset size occurrence distribution with the help of an item counter array, which records the number of contained items per transaction.

divide-and-conquer approach (see, for example, [5] for a formal description), this intermediate array is updated every time an item is added to or removed from the current item set. The counter update is carried out with the help of transaction identifier lists, that is, our algorithm uses a vertical database representation and thus is closely related to the well-known Eclat algorithm [20]. The updated fields of the item counter array then give rise to updates of the subset size occurrence distribution, which records, for each subset size, how many transactions contain at least as many items of the current item set.

Pseudo-code of the (simplified) recursive search procedure is shown in Figure 2. Together with the recursion the main while-loop implements the depth-first/divide-and-conquer search by first including an item in the current set (first subproblem — handled by the recursive call) and then excluding it (second subproblem — handled by skipping the item in the while-loop).

The for-loop at the beginning of the outer while-loop increments first the item counters for each transaction containing the current item n, which thus is added to the current item set. Then the subset size occurrence distribution is updated by drawing on the new value of the updated item counters. Note that one could also remove a transaction from the counter for the original subset size (after adding the current item), so that the distribution array elements represent the number of transactions that contain *exactly* the number of items given by their indices. This could be achieved with an additional $\mathbf{dec}(dist[cnts[t[i]]])$ as the first statement of the for-loop. However, this operation is more costly than forming differences between neighboring elements in the evaluation function, which yields the same values (see Figure 4 — to be discussed later).

As an illustrative example, Figure 3 shows an example of the update. The top row shows the list of transaction identifiers for the current item n (held in the pseudo-code in the local variable t), which is traversed to select the item counters that have to be incremented. The second row shows these item counters, with old and unchanged counter values shown in black and updated values in blue. Using the new (blue) values as indices into the subset size distribution array, this distribution is updated. Again old and unchanged values are shown in black, new values in blue. Note that $dist[0]$ always holds the total number of transactions.

An important property of this update operation is that it is reversible. By traversing the transaction identifiers again, the increments can be retracted, thus restoring the original subset size occurrence distribution (before the current item n was added). This is exploited in the for-loop at the end of the outer while-

```
global variables:                          (may also be passed down in recursion)
wgts : array of real;                      (* weights per number of missing items *)

function eval (d: array of integer,        (* d: subset size occurrence distribution *)
               k: integer) : real;         (* k: number of items in the current set *)
var i: integer;                            (* loop variable *)
    e: real;                               (* evaluation result *)
begin
  e := d[k] · wgts[0];                     (* initialize the evaluation result *)
  for i := 1 upto min(k, length(wgts)) do  (* traverse the distribution *)
    e := e +(d[k − i] − d[k − i + 1]) · wgts[i];
  return e;                                (* weighted sum of transaction counters *)
end;   (* end of eval() *)
```

Fig. 4. Pseudo-code of a simple evaluation function.

loop in Figure 2, which restores the distribution, by first decrementing the subset size occurrence counter and then the item counter for the transaction (that is, the steps are reversed w.r.t. the update in the first for-loop).

Between the for-loops the subset size occurrence distribution is evaluated and if the evaluation result reaches a user-specified threshold, the extended item set is actually constructed and reported. Afterwards supersets of this item set are processed recursively and finally the current item is removed again. This is in perfect analogy to standard frequent item set algorithms like Eclat or FP-growth.

The advantage of this scheme is that the evaluation function has access to fairly rich information about the occurrences of subsets of the current item set. While standard frequent item set mining algorithms only compute (and evaluate) $dist[k]$ (which always contains the standard support) and the JIM algorithm [16] computes and evaluates only $dist[k]$, $dist[1]$ (number of transactions that contain at least one item in the set), and $dist[0]$ (total number of transactions), our algorithm knows (or can easily compute as a simple difference) how many transactions miss 1, 2, 3 etc. items. Of course, this additional information comes at a price, namely a higher processing time, but in return one obtains the possibility to compute much more sophisticated item set evaluations.

A very simple example of such an evaluation function is shown in Figure 4: it weights the numbers of transactions in proportion to the number of missing items. The weights can be specified by a user and are stored in a global weights array. We assume that $wgts[0] = 1$ and $wgts[i] \geq wgts[i+1]$. With this function fault-tolerant item sets can be found in a cost-based manner, where the costs are represented by the weights array. An obvious alternative—inspired by [16]—is to divide the final value of e by $dist[1]$ in order to obtain an extended Jaccard measure. In principle, all measures listed in [16] can be generalized in this way, by simply replacing the standard support (all items are contained) by the extended support computed in the function shown in Figure 4.

Note that the extended support computed by this function as well as the extended Jaccard measure that can be derived from it are obviously anti-monotone,

since each element of the subset size occurrence distribution is anti-monotone (if elements are paired from the number of items in the respective sets downwards), while *dist*[1] is monotone. This ensures the correctness of the algorithm.

4 Removing Pseudo/Spurious Item Sets

Pseudo (or spurious) item sets can result if there exists a set of items that is strongly correlated (no or almost no missing items) and supported by many transactions. Adding an item to this set may not reduce the support enough to let it fall below the user-specified threshold, even if this item is not contained in any of the transactions containing the correlated items. As an illustration consider the right diagram in Figure 1: the third item is contained in only one of the eight transactions. However, the total number of missing items in this binary matrix (and thus the extended support) is the same as in the middle diagram, which we consider as a representation of an acceptable fault-tolerant item set.

In order to cull such pseudo (or spurious) item sets from the output, we added to our algorithm a check whether all items of the set occur in a sufficiently large fraction of the supporting transactions. This check can be carried out in two forms: either the user specifies a minimum fraction of the support of an item set that must be produced from transactions containing the item (in this case the reduced weights of transactions with missing items are considered) or he/she specifies a minimum fraction of the number of supporting transactions that must contain the item (in this case all transactions have the same unit weight).

Both checks can fairly easily be carried out with the help of the vertical transaction representation (transaction identifier lists), the intermediate/auxiliary item counter array (with one counter per transaction) and the subset size occurrence distribution: One simply traverses the transaction identifier list for each item in the set to check and computes the number of supporting transactions that contain the tested item (or the support contribution derived from these transactions). The result is then compared with the total number of supporting transactions (which is available in $dist[m]$, where m is the number of weights (see Figure 4) or the extended support (the result of the evaluation function shown in Figure 4). If the result exceeds a user specified threshold (given as a fraction or percentage) for all items in the set, the item set is accepted, otherwise it is discarded (from the output, but still processed recursively, because these conditions are not anti-monotone and thus cannot be used for pruning).

In addition, it is often beneficial to filter the output for closed item sets (no superset has the same support/evaluation) or maximal item sets (no superset has a support/evaluation exceeding the user-specified threshold). In principle, this can be achieved with the same methods that are used in standard frequent item set mining. In our algorithm we consider closedness or maximality only w.r.t. the standard support (all items contained), but in principle, it could also be implemented w.r.t. the more sophisticated measures. Note, however, that this notion of closedness differs from the notion introduced and used in [6, 14], which is based on δ-free item sets and is a mathematically ore sophisticated approach.

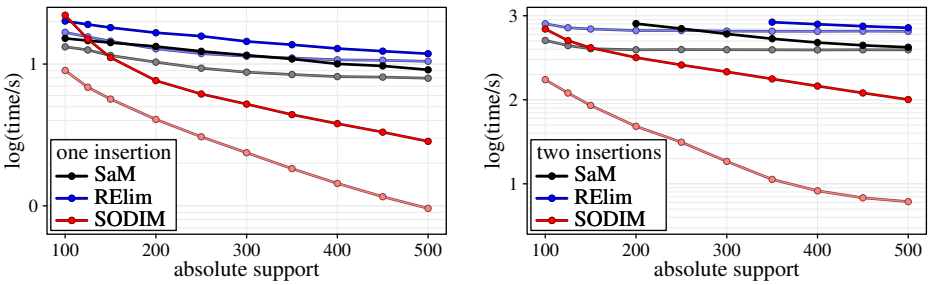

Fig. 5. Execution times on the BMS-Webview-1 data set. Light colors refer to an insertion penalty factor of 0.25, dark colors to an insertion penalty factor of 0.5.

5 Experiments

We implemented the described item set mining approach as a C program, called SODIM (**S**ubset size **O**ccurrence **D**istribution based **I**tem set **M**ining), that was essentially derived from an Eclat implementation (which provided the initial setup of the transaction identifier lists). We implemented all measures listed in [16], even though for these measures (in their original form) the JIM algorithm is better suited, because they do not require subset occurrence values beyond $dist[k]$, $dist[1]$, and $dist[0]$. However, we also implemented the extended support and the extended Jaccard measure (as well as generalizations of all other measures described in [16]), which JIM cannot compute. We also added optional culling of pseudo (or spurious) item sets, thus providing possibilities far surpassing the JIM implementation. This SODIM implementation has been made publicly available under the GNU Lesser (Library) Public License.[3]

In a first set of experiments we tested our implementation on artificially generated data. We created a transaction database with 100 items and 10000 transactions, in which each item occurs in a transaction with 5% probability (independent items, so co-occurrences are entirely random). Into this database we injected six groups of co-occurring items, which ranged in size from 6 to 10 items and which partially overlapped (some items were contained in two groups). For each group we injected between 20 and 30 co-occurrences (that is, in 20 to 30 transactions the items of the group actually co-occur). In order to compensate for the additional item occurrences due to this, we reduced (for the items in the groups) the occurrence probabilities in the remaining transactions (that is, the transactions in which they did not co-occur) accordingly, so that all items shared the same individual expected frequency. In a next step we removed from each co-occurrence of a group of items one group item, thus creating the noisy instances of item sets we try to find with the SODIM algorithm. Note that due to this deletion scheme *no* transaction contained all items in a given group and thus no standard frequent item set mining algorithm is able to detect the groups, regardless of the used minimum support threshold.

[3] `http://www.borgelt.net/sodim.html`

We then mined this database with SODIM, using a minimum standard support (all items contained) of 0, a minimum extended support of 10 (with a weight of 0.5 for transactions with one missing item) and a minimum fraction of transaction containing each item of 75%. In addition, we restricted the output to maximal item sets (based on standard support), in order to suppress the output of subsets of the injected groups. This experiment was repeated several times with different databases generated in the way described above. We observed that the injected groups were always perfectly detected, while only rarely a false positive result, usually with 4 items, was produced.

In a second set of experiments we compared SODIM to the two other cost-based methods reviewed in Section 2, namely RElim [18] and SaM [5]. As a test data set we chose the well-known BMS-Webview-1 data, which describes a web click stream from a leg-care company that no longer exists. This data set has been used in the KDD cup 2000 [12]. By properly parameterizing these methods, they can be made to find exactly the same item sets. We chose two insertion penalties (RElim and SaM) or downweighting factors for missing items (SODIM), namely 0.5 and 0.25, and tested with one and two insertions (RElim and SaM) or missing items (SODIM). The results, obtained on an Intel Core 2 Quad Q9650 (3GHz) Computer with 8 GB main memory running Ubuntu Lunix 10.04 (64 bit) and gcc version 4.4.3, are shown in Figure 5. Clearly, SODIM outperforms both SaM and RElim by a large margin, with the exception of the lowest support value for one insertion and a penalty of 0.5, where SODIM is slightly slower than both SaM and RElim. It should be noted, though, that this does not render SaM and RElim useless, because they offer options that SODIM does not, namely the possibility to define item-specific insertion penalties (SODIM treats all items the same). On the other hand, SODIM allows for more sophisticated evaluation measures and the removal of pseudo/spurious item sets. Hence all three algorithms are useful.

To demonstrate the usefulness of our method, we applied it also to the 2008/2009 Wikipedia Selection for schools[4], which is a subset of the English Wikipedia[5] with about 5500 articles and more than 200,000 hyperlinks. We used a subset of this data set that does not contain articles belonging to the subjects "Geography", "Countries" or "History", resulting in a subset of about 3,600 articles and more than 65,000 hyperlinks. The excluded subjects do not affect the chemical subject we focus on in our experiment, but contain articles that reference many articles or that are referenced by many articles (such as *United States* with 2,230 references). Including the mentioned subject areas would lead to an explosion of the number of discovered item sets and thus would make it much more difficult to demonstrate the effect we are interested in.

The 2008/2009 Wikipedia Selection for schools describes 118 chemical elements[6]. However, there are 157 articles that reference the *Chemical element* article or are referenced by it, so that simply collecting the referenced or referencing articles does not yield a good extensional representation of this concept.

[4] `http://schools-wikipedia.org/`
[5] `http://en.wikipedia.org`
[6] `http://schools-wikipedia.org/wp/l/List_of_elements_by_name.htm`

Table 1. Results for different numbers of missing items.

Missing items	Transactions	Chemical elements	Other elements	Not referenced
0	25	24	1	0
1	47	34	13	1
2	139	71	68	3
3	239	85	154	9

Searching for references to the *Chemical element* article thus results not only in articles describing chemical elements but also in other articles including *Albert Einstein*, *Extraterrestrial Life*, and *Universe*. Furthermore, there are 17 chemical elements (e.g. palladium) that do not reference the *Chemical element* article.

In order to better filter articles that are about a chemical element, one may try to extend the query with the titles of articles that are frequently co-referenced with the *Chemical element* article, but are more specific than a reference to/from this article alone. In order to find such co-references, we apply our SODIM algorithm. In order to do so, we converted each article into a transaction, such that each referenced article is an item in the transaction of the referring article. This resulted in a transaction database with 3,621 transactions.

We then ran our SODIM algorithm with a minimum item set size of 5 and a minimum support (all items contained) of 25 in order to find the co-references. 29 of the 81 found item sets contain the item *Chemical element*. From the 29 item sets we chose the following item set for the subsequent experiments: {*Oxygen*, *Electron*, *Hydrogen*, *Melting point*, *Chemical Element*}.

The first column of Table 1 shows the results for different settings for the allowed number of missing items. The second column contains the number of matching transactions. Column three and four contain the number of discovered chemical elements and the number of other articles. The last column contains the number of discovered chemical elements that do *not* reference the *Chemical Element* article. By allowing some missing items per transaction the algorithm was able to find considerably more chemical elements than the classical version.

6 Conclusions and Future Work

In this paper we presented a new cost-based algorithm for mining fault-tolerant frequent item sets that exploits subset size occurrence distributions. The algorithm efficiently computes these distributions while traversing the search space in the usual depth-first manner. As evaluation measures we suggested a simple extended support, by which transactions containing only some of the items of a given set can still contribute to the support of this set, as well as an extension of the generalized Jaccard index that is derived from the extended support. Since the algorithm records, in an intermediate array, for each transaction how many items of the currently considered set are contained, we could also add a simple and efficient check in order to cull pseudo and spurious item sets from the output. We demonstrated the usefulness of our algorithm by applying it, combined

with filtering for maximal item sets, to the 2008/2009 Wikipedia Selection for schools, where it proved beneficial to detect the concept of a chemical element despite the only limited standardization of pages on such substances.

We are currently trying to extend the method to incorporate item weights (weighted or uncertain transactional data, see Section 2), in order to obtain a method that can mine fault-tolerant item sets from uncertain or weighted data. A main problem of such an extension is that the item weights have to be combined over the items of a considered set (for instance, with the help of a t-norm). This naturally introduces a tendency that the weight of a transaction goes down even if the next added item is contained, simply because the added item is contained with a weight less than one. If we now follow the scheme of downweighting transactions that are missing an item with a user-specified factor, we have to make sure that a transaction that contains an item (though with a low weight) does not receive a lower weight than a transaction that does not contain the item (because the downweighting factor is relatively high).

Acknowledgements

This work was supported by the European Commission under the 7th Framework Program FP7-ICT-2007-C FET-Open, contract no. BISON-211898.

References

1. C.C. Aggarwal, Y. Lin, J. Wang and J. Wang. Frequent Pattern Mining with Uncertain Data. *Proc. 15th ACM SIGMOD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2009, Paris, France)*, 29–38. ACM Press, New York, NY, USA 2009
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *Proc. 20th Int. Conf. on Very Large Databases (VLDB 1994, Santiago de Chile)*, 487–499. Morgan Kaufmann, San Mateo, CA, USA 1994
3. J. Besson, C. Robardet, and J.-F. Boulicaut. Mining a New Fault-Tolerant Pattern Type as an Alternative to Formal Concept Discovery. *Proc. Int. Conference on Computational Science (ICCS 2006, Reading, United Kingdom)*, 144–157. Springer-Verlag, Berlin, Germany 2006
4. D. Berger, C. Borgelt, M. Diesmann, G. Gerstein, and S. Grün. An Accretion based Data Mining Algorithm for Identification of Sets of Correlated Neurons. *18th Ann. Computational Neuroscience Meeting (CNS*2009)*. Berlin, Germany 2009
5. C. Borgelt and X. Wang. SaM: A Split and Merge Algorithm for Fuzzy Frequent Item Set Mining. *Proc. 13th Int. Fuzzy Systems Association World Congress and 6th Conf. of the European Society for Fuzzy Logic and Technology (IFSA/EUSFLAT'09, Lisbon, Portugal)*, 968–973. IFSA/EUSFLAT Organization Committee, Lisbon, Portugal 2009
6. J.F. Boulicaut, A. Bykowski, and C. Rigotti. Approximation of Frequency Queries by Means of Free-sets. *Proc. 4th Europ. Conf. Principles and Practice of Knowledge Discovery in Databases (PKDD 2000, Lyon, France)*, LNCS 1910:75–85. Springer, Heidelberg, Germany 2000

7. T. Calders, C. Garboni, and B. Goethals. Efficient Pattern Mining of Uncertain Data with Sampling. *Proc. 14th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD 2010, Hyderabad, India)*, I:480–487. Springer-Verlag, Berlin, Germany 2010
8. C.-K. Chui, B. Kao, and E. Hung. Mining Frequent Itemsets from Uncertain Data. *Proc. 11th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD 2007, Nanjing, China)*, 47–58. Springer-Verlag, Berlin, Germany 2007
9. C. Creighton and S. Hanash. Mining Gene Expression Databases for Association Rules. *Bioinformatics* 19:79–86. Oxford University Press, Oxford, United Kingdom 2003
10. A. Gionis, H. Mannila, and J.K. Seppänen. Geometric and Combinatorial Tiles in 0-1 Data. *Proc. 8th Europ. Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD04, Pisa, Italy)*, LNAI 3202:173-184. Springer-Verlag, Berlin, Germany 2004
11. S. Grün and S. Rotter (eds.) *Analysis of Parallel Spike Trains*. Springer-Verlag, Berlin, Germany 2010
12. R. Kohavi, C.E. Bradley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Exploration 2(2):86–93. ACM Press, New York, NY, USA 2000
13. C.K.-S. Leung, C.L. Carmichael, and B. Hao. Efficient Mining of Frequent Patterns from Uncertain Data. *Proc. 7th IEEE Int. Conf. on Data Mining Workshops (ICDMW 2007, Omaha, NE)*, 489–494. IEEE Press, Piscataway, NJ, USA 2007
14. R.G. Pensa, C. Robardet, and J.F. Boulicaut. Supporting Bi-cluster Interpretation in 0/1 Data by Means of Local Patterns. *Intelligent Data Analysis* 10:457–472. IOS Press, Amsterdam, Netherlands 2006
15. J. Pei, A.K.H. Tung, and J. Han. Fault-Tolerant Frequent Pattern Mining: Problems and Challenges. *Proc. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMK'01, Santa Babara, CA)*. ACM Press, New York, NY, USA 2001
16. M. Segond and C. Borgelt. Item Set Mining Based on Cover Similarity. *Proc. 15th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD 2011, Shenzhen, China)*, to appear. Springer-Verlag, Berlin, Germany 2011
17. J.K. Seppänen and H. Mannila. Dense Itemsets. *Proc. 10th ACM SIGMOD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2004, Seattle, WA)*, 683–688. ACM Press, New York, NY, USA 2004
18. X. Wang, C. Borgelt, and R. Kruse. Mining Fuzzy Frequent Item Sets. *Proc. 11th Int. Fuzzy Systems Association World Congress (IFSA'05, Beijing, China)*, 528–533. Tsinghua University Press and Springer-Verlag, Beijing, China, and Heidelberg, Germany 2005
19. C. Yang, U. Fayyad, and P.S. Bradley. Efficient Discovery of Error-tolerant Frequent Itemsets in High Dimensions. *Proc. 7th ACM SIGMOD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2001, San Francisco, CA)*, 194–203. ACM Press, New York, NY, USA 2001
20. M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97, Newport Beach, CA)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997