A Scalable Data Analytics Algorithm for Mining Frequent Patterns from Uncertain Data

PAKDD-SDA 2014: Paper 11

Abstract. With advances in technology, massive amounts of valuable data can be collected and transmitted at high velocity in various scientific, biomedical or engineering applications. Hence, scalable data analytics tools are in demand for analyzing these data. For example, scalable tools for association analysis help reveal frequently occurring patterns and their relationships, which in turn lead to intelligent decisions. While a majority of existing frequent pattern mining algorithms—including FP-growth—handle only precise data, there are situations in which data are *uncertain*. In recent years, researchers have paid attention to frequent pattern mining from uncertain data. UF-growth and UFP-growth are examples of tree-based algorithms for mining uncertain data. However, their corresponding tree structures can be large. Other tree structures for handling uncertain data may achieve compactness at the expense of loose upper bounds on expected supports. To solve this problem, we propose (i) a compact tree structure that captures uncertain data with tighter upper bounds than aforementioned tree structures and (ii) a scalable data analytics algorithm that mines frequent patterns from our tree structure. Experimental results show the tightness of bounds to expected supports provided by our algorithm.

Keywords: Association analysis, frequent patterns, pattern discovery, tree structures, uncertain data

1 Introduction

Since the advent of frequent pattern mining [1], numerous studies have been conducted which explore mining *frequent patterns* (i.e., *frequent itemsets*) from *precise* data such as databases of supermarket transactions [11]. Mining for other interesting patterns such as skyline patterns [18] or permission request patterns [7] has also been studied. In cases where the source data is not static, frequent patterns have been mined from sliding windows in data streams [8, 13, 17]. In addition, frequent patterns can be used as input to other processes such as dimensionality reduction on databases [10].

Users definitely know whether an item is present in, or is absent from, a transaction in databases of precise data. However, there are situations in which users are uncertain about the presence or absence of items [3–5, 13, 15, 16, 20]. For example, a meteorologist may suspect (but cannot guarantee) that severe weather phenomena will develop during a thunderstorm. The uncertainty of such suspicions can be expressed in terms of existential probability. For instance, a thunderstorm may have a 60% likelihood of generating hail, and only a 15%

likelihood of generating a tornado, regardless of whether or not there is hail. With this notion, each item in a transaction t_j in databases containing precise data can be viewed as an item with a 100% likelihood of being present in t_j.

To handle uncertain data, the *U-Apriori algorithm* [6] was proposed in PAKDD 2007. As an Apriori-based algorithm, U-Apriori requires multiple scans of uncertain databases. To reduce the number of database scans (down to two), the tree-based *UF-growth algorithm* [14] was proposed in PAKDD 2008. In order to compute the *exact* expected support of each pattern, paths in the corresponding UF-tree are shared only if tree nodes on the paths have the same item and same existential probability. Hence, the UF-tree may be quite large when compared to the FP-tree. In an attempt to make the tree compact, the *UFP-growth algorithm* [2] groups *similar* nodes (with the same item x and similar existential probability values) into a cluster. However, depending on the clustering parameter, the corresponding UFP-tree may be as large as the UF-tree (i.e., no reduction in tree size). Moreover, because UFP-growth does not store every existential probability value for an item in a cluster, it returns not only the frequent patterns but also some infrequent patterns (i.e., false positives). In PAKDD 2013, the *PUF-growth algorithm* [16] was proposed to addresses some of the deficiencies of the UF-tree and UFP-tree. PUF-growth utilizes a concept of an upper bound to expected support together with much more aggressive path sharing to yield a smaller tree structure. Moreover, as alternatives to trees, hyperlinked array structures were used by the *UH-Mine algorithm* [2], which was reported [19] to outperform UFP-growth.

In this paper, we study the following questions: Can we further tighten the upper bound on expected support (e.g., lower than the PUF-tree)? Can the resulting tree be as compact as the FP-tree? How would frequent patterns be mined from such a tree? Our **key contributions** of this paper are as follows:

1. a **b**ranch-**l**evel **i**te**m** **p**refix-cap **tree (BLIMP-tree)**, which can be as compact as the original FP-tree; and
2. a scalable mining algorithm (namely, **BLIMP-growth**, which is guaranteed to find *all and only those* frequent patterns (i.e., *no* false negatives and *no* false positives) from uncertain data.

The remainder of this paper is organized as follows. The next section presents background and related works. We then propose our BLIMP-tree structure and BLIMP-growth algorithm in Sections 3 and 4, respectively. Experimental results are shown in Section 5, and conclusions are given in Section 6.

2 Background and Related Works

In this section, we first give some background information about frequent pattern mining of uncertain data (e.g., existential probability, expected support), and we then discuss some related works.

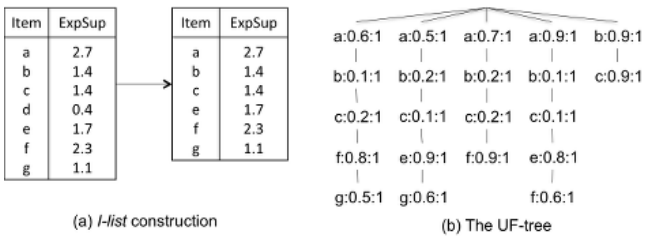

Fig. 1. The UF-tree for the database shown in Table 1 when *minsup*=1.1

2.1 Existential Probability and Expected Support

Let (i) `Item` be a set of m domain items and (ii) $X = \{x_1, x_2, \ldots, x_k\}$ be a k-itemset (i.e., a pattern consisting of k items), where $X \subseteq$ `Item` and $1 \leq k \leq m$. Then, a transactional database $= \{t_1, t_2, \ldots, t_n\}$ is the set of n transactions, where each transaction $t_j \subseteq$ `Item`. The projected database of X is the set of all transactions containing X.

Unlike precise databases, each item x_i in a transaction $t_j = \{x_1, x_2, \ldots, x_h\}$ in an uncertain database is associated with an ***existential probability value*** $\boldsymbol{P(x_i, t_j)}$, which represents the likelihood of the presence of x_i in t_j [12]. Note that $0 < P(x_i, t_j) \leq 1$. The ***existential probability*** $\boldsymbol{P(X, t_j)}$ **of a pattern** $\boldsymbol{X}$ **in** $\boldsymbol{t_j}$ is then the product of the corresponding existential probability values of items within X when these items are independent [12]: $P(X, t_j) = \prod_{x \in X} P(x, t_j)$. The ***expected support*** $\boldsymbol{expSup(X)}$ of X in the database is the sum of $P(X, t_j)$ over all n transactions in the database: $expSup(X) = \sum_{j=1}^{n} P(X, t_j)$. A pattern X is ***frequent*** in an uncertain database if $expSup(X) \geq$ a user-specified minimum support threshold *minsup*. Given a database and *minsup*, the research problem of **frequent pattern mining from uncertain data** is to discover from the database a complete set of frequent patterns having expected support $\geq$ *minsup*.

2.2 Tree-Based Frequent Pattern Mining Algorithms

Recall from Section 1 that the tree-based **UF-growth algorithm** [14] uses **UF-trees** to mine frequent patterns from uncertain databases in two scans of the database. Each node in a UF-tree captures (i) an item x, (ii) its existential probability, and (iii) its occurrence count. Tree paths are shared if the nodes on these paths share the same ⟨item, existential probability⟩-value. In general, when dealing with uncertain data, it is not uncommon that the existential probability values of the same item vary from one transaction to another. As such, the resulting UF-tree may not be as compact as the FP-tree. See Fig. 1, which shows a UF-tree for the database presented in Table 1 when *minsup*=0.5. The UF-tree contains four nodes for item a with different probability values as children of the root. Efficiency of the corresponding UF-growth algorithm, which finds all and *only those* frequent patterns, partially relies on the compactness of the UF-tree.

In an attempt to make the tree more compact, the **UFP-growth algorithm** [2] was proposed. Like UF-growth, the UFP-growth algorithm also scans

Table 1. A transactional database (*minsup*=1.1)

TID	Transactions	Sorted transactions (with infrequent items removed)
t_1	{a:0.6, b:0.1, c:0.2, f:0.8, g:0.5}	{a:0.6, b:0.1, c:0.2, f:0.8, g:0.5}
t_2	{a:0.5, b:0.2, c:0.1, e:0.9, g:0.6}	{a:0.5, b:0.2, c:0.1, e:0.9, g:0.6}
t_3	{a:0.7, b:0.2, c:0.2, f:0.9}	{a:0.7, b:0.2, c:0.2, f:0.9}
t_4	{a:0.9, b:0.1, c:0.1, e:0.8, f:0.6}	{a:0.9, b:0.1, c:0.1, e:0.8, f:0.6}
t_5	{b:0.9, c:0.9, d:0.4}	{b:0.9, c:0.9}

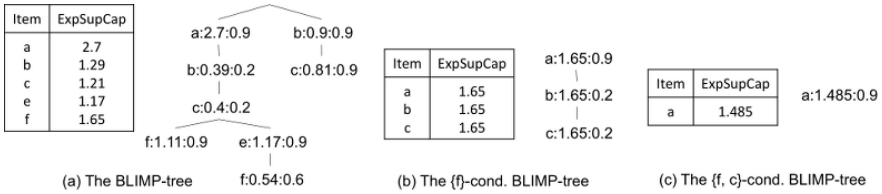

Fig. 2. The BLIMP-tree for the database shown in Table 1 when *minsup*=1.1

the database twice and builds a **UFP-tree**. As nodes for item x having similar existential probability values are clustered into a mega-node, the resulting mega-node in the UFP-tree captures (i) an item x, (ii) the *maximum* existential probability value (among all nodes within the cluster), and (iii) its occurrence count (i.e., the number of nodes within the cluster). Tree paths are shared if the nodes on these paths share the same item but *similar* existential probability values. In other words, the path sharing condition is less restrictive than that of the UF-tree. By extracting appropriate tree paths and constructing UFP-trees for subsequent projected databases, UFP-growth finds all frequent patterns and some *false positives* at the end of the second database scan. The third database scan is then required to remove those false positives.

A further attempt to improve the compactness of the tree was proposed with the **PUF-growth algorithm** [16]. The algorithm uses a PUF-tree to tighten the upper bound on the expected support of patterns. Each node in a PUF-tree captures (i) an item x and (ii) an upper bound. Instead of multiplying the existential probability of x with that of the highest existential probability of any other item in the same transaction as x. PUF-growth mines frequent patterns by taking advantage of the tree structure itself to restrict the multiplication to the highest existential probability of any other item in the *prefix* of x. A direct effect of lower numbers of false positives is a shorter runtime due to fewer projected databases needing to be extracted and less work required in the third database scan. For these reasons, PUF-growth was reported to be faster than UH-Mine [15].

3 Our BLIMP-tree Structure

In PUF-trees, the upper bound to expected support of an item x_r in a transaction t_j was calculated by the product of the existential probability of x_r and the highest existential probability among all items in the prefix of t_j (i.e., before x_r

in t_j). Such a bound also serves as an upper bound to the expected support for all the k-itemsets ($k \geq 2$) containing x_r and items in its prefix in t_j.

To further tighten the upper bound for all k-itemsets ($k > 2$), we propose the BLIMP-tree structure. The key idea is to keep track of a new value—the "blimp" value—calculated solely from the maximum of all existential probabilities for the single item represented in a node x_r. Intuitively, this "blimp" value is the maximum existential probability of x_r among all transactions containing x_r. Every time a frequent extension ($k > 2$) is added to the suffix item, the blimp value is used. Hence, each node in a BLIMP-tree contains: (i) an item x_r, (ii) an item cap $I^{Cap}(x_r, t_j)$ and (iii) a "blimp" value. See the following definitions.

Definition 1. The **item cap $I^{Cap}(x_r, t_j)$** of an item x_r in a transaction $t_j = \{x_1, \ldots, x_r, \ldots, x_h\}$, where $1 \leq r \leq h$, is defined as the product of $P(x_r, t_j)$ and the highest existential probability value M of items from x_1 to x_{r-1} in t_j (i.e., in the *proper prefix* of x_r in t_j):

$$I^{Cap}(x_r, t_j) = \begin{cases} P(x_r, t_j) \times M & \text{if } h > 1 \\ P(x_1, t_j) & \text{if } h = 1 \end{cases}, \text{ where } M = \max_{1 \leq q \leq r-1} P(x_q, t_j). \quad \square$$

Fig. 2(a) shows the contents of a BLIMP-tree for the database in Table 1, in which each node additionally maintains the highest probability value for items represented by that node in that tree path. With this information, BLIMP-trees give a *new upper bound* on the expected support of an itemset by the product of $I^{Cap}(x_r, t_j)$ and the blimp values in the prefix of x_r. This new compounded item cap of any k-itemset $X = \{y_1, y_2, \ldots, y_k\}$ in $t_j = \{x_1, \ldots, x_r, \ldots, x_h\} \subseteq$ `Item` (denoted as $I(\widehat{X, t_j})$ where $y_k = x_r$) can be defined as follows.

Definition 2. *Let $t_j = \{x_1, \ldots, x_r, \ldots, x_h\}$, where $h = |t_j|$ and $r \in [1, h]$; let $X = \{y_1, y_2, \ldots, y_k\}$ is a k-itemset in t_j such that $y_k = x_r$. Also, let M_{y_i} denote the maximum existential probability of the i-th item y_i in X among all transactions (including t_j) that contain y_k. Then,*

$$I(\widehat{X, t_j}) = \begin{cases} I^{Cap}(x_r, t_j) & \text{if } k \leq 2, \\ I^{Cap}(x_r, t_j) \times \prod_{i=1}^{k-2} M_{y_i} & \text{if } k \geq 3. \end{cases}$$

Example 1. Consider an uncertain database with five transactions as presented in the Transactions column in Table 1. $I(\widehat{X, t_j})$ for $X = \{a, b, c, f\}$ and $j = 1$ can be computed as $I^{Cap}(f, t_1) \times (\prod_{i=1}^{2} M_{x_i}) = 0.8 \times M \times M_{x_1} \times M_{x_2} = 0.8 \times 0.6 \times 0.6 \times 0.1 = 0.0288$. Similarly, $I(\widehat{X, t_j})$ for $X = \{b, c, g\}$ and $j = 1$ is $I^{Cap}(g, t_1) \times (\prod_{i=1}^{1} M_{x_i}) = 0.5 \times M \times M_{x_1} = 0.5 \times 0.8 \times 0.1 = 0.04$. $\square$

Note that $expSup^{Cap}(X) = \sum_{j=1}^{n} \{I(\widehat{X, t_j}) \mid X \subseteq t_j\}$, and it serves as an upper bound on the expected support of X. However, in BLIMP-trees, the *cap of expected support* does not satisfy the downward closure property because $expSup^{Cap}(Y)$ can be less than $expSup^{Cap}(X)$ for some proper subset Y of X. See Example 2.

Example 2. Let t_{21}={a:0.3, e:0.2, f:0.9, g:0.9} and t_{22} = {a:0.4, e:0.7, f:0.3, g:0.9} be the only transactions in the database containing X={a, e, f, g} and its subset Y = {a,

$e\}$. Then, $expSup^{Cap}(Y) = (P(e, t_{21}) \times M) + (P(e, t_{22}) \times M) = (0.2 \times 0.3) + (0.7 \times 0.4) = 0.06 + 0.28 = 0.34$. $expSup^{Cap}(X) = [(P(g, t_{21}) \times M) + (P(g, t_{22}) \times M)] \times M_{x_1} \times M_{x_2} = [(0.9 \times 0.9) + (0.9 \times 0.7)] \times 0.4 \times 0.7 = 0.4032$. This shows that $expSup^{Cap}(Y)$ can be $< expSup^{Cap}(X)$. □

However, for the same *specific* cases (e.g., when X & Y share the same suffix item x_r), the cap of expected support for BLIMP-trees satisfies the downward closure property and as such exhibits the *partial downward closure property*: For any non-empty subset Y of X such that both X & Y end with x_r, (i) $expSup^{Cap}(X) \geq minsup$ implies $expSup^{Cap}(Y) \geq minsup$ and (ii) $expSup^{Cap}(Y) < minsup$ implies $expSup^{Cap}(X) < minsup$.

Lemma 1. The cap of expected support of a pattern X satisfies the *partial* downward closure property.

The process of constructing a BLIMP-tree is identical to that of the UF-tree [14] for the first database scan, in which expected support of every domain item is computed. Any infrequent item x (i.e., having $expSup(x) < minsup$) is removed. During a second database scan, the BLIMP-tree is constructed in a fashion similar (but not identical) to that of the UF-tree. The key difference is that, when inserting a transaction item, in addition to computing the item cap, we also pass along the existential probability of that item. The item is then inserted into the BLIMP-tree according to the *I-list* order. If a node containing that item already exists in the tree path, we update both (i) its item cap (by taking the sum of the current item cap of the item with the existing item cap) and (ii) its blimp value (by taking the maximum of the current existential probability of the item with the existing blimp value). Otherwise, we create a new node with the item cap and existential probability of the item (i.e., the initial blimp value). For a better understanding of BLIMP-tree construction, see Example 3.

Example 3. Consider the database in Table 1, and let the user specified support threshold *minsup* be set to 1.1. Let the *I-list* follow the alphabetic ordering of item values. After the first database scan, the contents of the *I-list* after computing the expected supports of all items and after removing infrequent items (e.g., item d) are $\langle a$:2.7, b:1.4, c:1.4, e:1.7, f:2.3, g:1.1$\rangle$.

With the second database scan, we insert only the frequent items of each transaction (with their respective item cap values) in the *I-list* order. For instance, when inserting transaction t_1=$\{a$:0.6, b:0.1, c:0.2, f:0.8, g:0.5$\}$, items a, b, c, f and g (with their respective item cap and existential probability values $\langle 0.6, 0.6\rangle$, $\langle 0.1 \times 0.6 = 0.06, 0.1\rangle$, $\langle 0.2 \times 0.6 = 0.12, 0.2\rangle$, $\langle 0.8 \times 0.6 = 0.48, 0.8\rangle$, $\langle 0.5 \times 0.8 = 0.4, 0.5\rangle$) are inserted in the BLIMP-tree as shown in Fig. 2(a). As t_2 shares a common prefix $\langle a, b, c\rangle$ with an existing path in the BLIMP-tree created when t_1 was inserted, (i) the item cap values of those items in the common prefix (i.e., a, b and c) are added to their corresponding nodes, (ii) the existential probability values of those items are checked against the blimp values for their corresponding nodes, with only the maximum saved for each node, and (iii) the remainder of the transaction (i.e., a new branch for items e and g) is inserted as a child of the last node of the prefix (i.e., as a child of c). Fig. 2(a) shows the status of the BLIMP-tree after inserting all of the remaining transactions and pruning those items with infrequent extensions (i.e., item g, since its total item cap is less than the

user specified *minsup*. Similar to the FP-tree, our BLIMP-tree maintains horizontal node traversal pointers, which are not shown in the figures for simplicity. □

The number of tree nodes in a BLIMP-tree (i) can be equal to that of an FP-tree [9] (when the BLIMP-tree is constructed using the frequency-descending order of items) and (ii) is bounded above by $\sum_{t_j \in \text{ DB}} |F(t_j)|$. In addition, the complete set of mining results can be generated because a BLIMP-tree contains $F(t_j)$ for all transactions as it stores the total item cap for a node. Mining based on this item cap value ensures that no frequent k-itemset ($k > 1$) will be missed.

Furthermore, the compounded item cap for any pattern X computed based on (i) the existential probability value of x_k, (ii) the highest existential probability value in its prefix and (iii) the blimp values of its prefix items provides a *tighter* upper bound than that based on the upper bound of PUF-trees because the former tightens the bound as candidates are generated during the mining process with increasing cardinality of X, whereas the latter has no such compounding effect.

4 The BLIMP-growth Algorithm

Here, we propose a pattern-growth mining algorithm called **B**LIMP-growth, which mines frequent patterns from our BLIMP-tree structure. In general, the construction of a BLIMP-tree is similar to that of UF-tree, except that BLIMP values are additionally stored as the third component in each tree node. (Recall that each node in a UF-tree contains only two components: an item and its expected support.) Thus, the basic operation in BLIMP-growth for mining frequent patterns is to construct a projected database for each potential frequent pattern and recursively mine its potentially frequent extensions.

Once an item x is found to be potentially frequent, the existential probability of x must contribute to the expected support computation for every pattern constructed from the $\{x\}$-projected database (denoted as DB_x). Hence, the cap of expected support of x is guaranteed to be the upper bound of the expected support of the pattern, since it is calculated from the existential probability of x together with the two highest existential probability values in the prefix of x, whereas the expected support of x uses every existential probability value in the prefix of x. This implies that the complete set of patterns with suffix x can be mined based on the partial downward closure property stated in Lemma 1. Note that $expSup^{Cap}(X)$ is the upper bound of $expSup(X)$, and it satisfies the partial downward closure property. So, we can directly proceed to generate all potential frequent patterns from the BLIMP-tree based on the following corollary.

Corollary 1. Let (i) X be a k-itemset (where $k > 1$) with $expSup^{Cap}(X) \geq minsup$ in the database and (ii) Y be an itemset in the X-projected database (denoted as DB_X). Then, $expSup^{Cap}(Y \cup X)$ in the original database $\geq minsup$ if and only if $expSup^{Cap}(Y)$ in all the transactions in $DB_X \geq minsup$.

Based on Lemma 1 and Corollary 1, which apply to BLIMP-trees without change, we apply the BLIMP-growth algorithm to our BLIMP-tree for generating

only those k-itemsets (where $k > 1$) with caps of expected support $\geq$ *minsup*. Although this mining process may also lead to some false positives (i.e., those itemsets that appear to be frequent but truly infrequent) in the resulting set of frequent patterns at the end of the second database scan, all these false positives will be easily filtered out with the third database scan. Our BLIMP-growth is guaranteed to return to the user the *exact* collection of frequent patterns (i.e., *all* and *only those* frequent patterns with *neither* false positives *nor* false negatives).

Example 4. The BLIMP-growth algorithm mines extensions of every item in the *I-list*. For example, the $\{f\}$-conditional tree, as shown in Fig. 2(b) when *minsup* = 1.1, is constructed by accumulating the tree path $\langle a$:2.7, b:0.39, c:0.4, e:1.17 $\rangle$. When projecting this path, BLIMP-growth updates the cap of expected support for each item in the projected database (as shown in the *I-list* in the figure) using the caps of expected support from all f nodes in the original tree. Since e appears only in the path for f:0.54, it is pruned in DB_f for having $expSup^{Cap} < minsup$. Since every other item in the prefix of f has $expSup^{Cap} \geq minsup$, they are represented in the $\{f\}$-conditional tree.

This $\{f\}$-conditional tree is then used to generate (i) all 2-itemsets containing item f and (ii) their further extensions by recursively constructing projected databases from them. For all k-itemsets, $k \geq 3$, that are generated, the cap of expected support is multiplied by the blimp value in each node. Consequently, at the $k = 2$ level in the $\{f\}$-conditional tree, patterns $\{f, c\}$:1.65, $\{f, b\}$:1.65 and $\{f, a\}$:1.65 are generated. BLIMP-growth then continues to mine the $\{f, b\}$-conditional tree and the $\{f, c\}$-conditional tree, as shown in Fig. 2(c). Notice that b does not appear in the $\{f, c\}$-conditional tree since the multiplication of the $expSup^{Cap}(c)$ with the blimp value of b in the $\{f\}$-conditional tree generates a new $expSup^{Cap}(b)$ in $DB_{f,c} < minsup$. Consequently, at the $k = 3$ level in the $\{f, c\}$-conditional tree, only the pattern $\{f, c, a\}$:1.485 is generated. Because of lower blimp values we are able to prune candidates $\{f, c, b\}$ and $\{f, c, b, a\}$ from ever being generated in BLIMP-growth.

The complete set of candidates generated by BLIMP-growth includes: $\{f, c, a\}$:1.485, $\{f, b, a\}$:1.485, $\{f, c\}$:1.65, $\{f, b\}$:1.65, $\{f, a\}$:1.65, $\{e, c\}$:1.17, $\{e, b\}$:1.17, $\{e, a\}$:1.17 and $\{c, b\}$:1.21 □

As shown in Example 4, BLIMP-growth finds a complete set of patterns from a BLIMP-tree without any false negatives.

5 Experimental Results

We compared the performances of our BLIMP-growth algorithm with the existing PUF-growth [16] algorithm. Recall from a previous work by Leung et al. [16] on the evaluation of the performance of PUF-growth that PUF-growth was shown to be outperformed UF-growth [14], UFP-growth [2] and UH-Mine [2]. We used both real life and synthetic datasets for our tests. The synthetic datasets, which are generally sparse, are generated within a domain of 1000 items by the data generator developed at IBM Almaden Research Center [1]. We also considered several real life datasets such as mushroom, retail and kosarack. We assigned a (randomly generated) existential probability value from the range (0,1] to each item in every transaction in the dataset. The name of each dataset indicates some

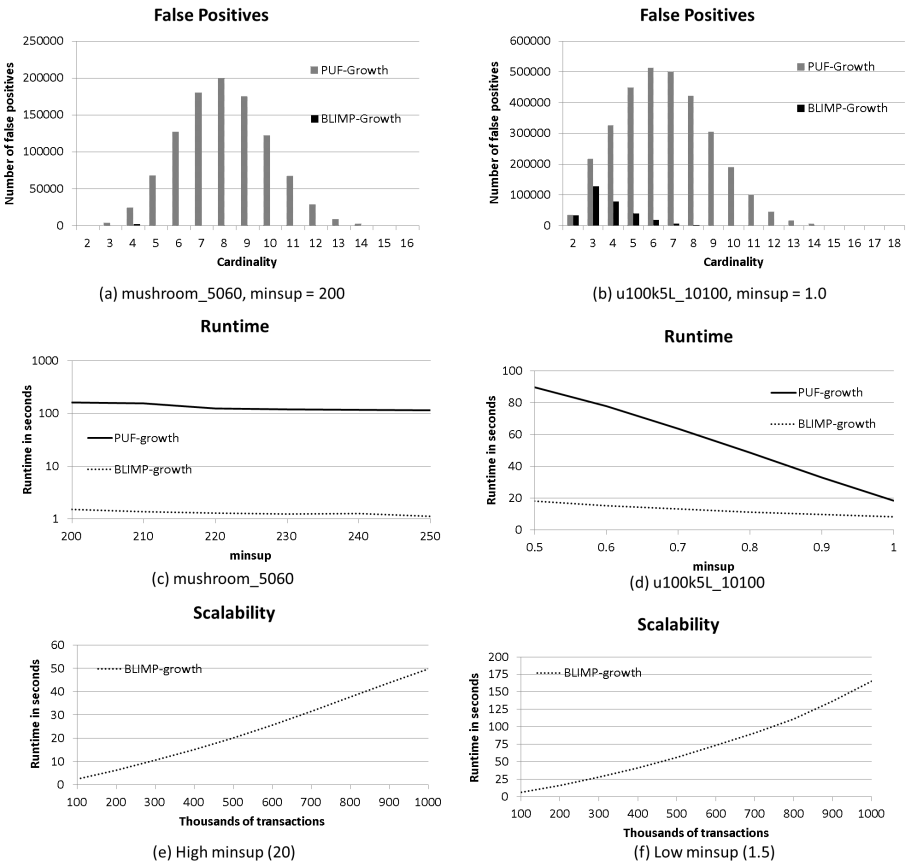

Fig. 3. Experimental results

characteristics of the dataset. For example, the dataset u100k5L_10100 contains 100K transactions with average transaction length of 5, and each item in a transaction is associated with an existential probability value that lies within a range of [10%, 100%]. Due to space constraints, we present here only some results on the above datasets.

All programs were written in C++ and ran in a Linux environment on an Intel Core i5-661 CPU with 3.33GHz and 7.5GB ram. Unless otherwise specified, runtime includes CPU and I/Os for *I-list* construction, tree construction, mining, and false-positive removal. While the number of false positives generated at the end of the second database scan may vary, all algorithms (ours and others) produce the same set of truly frequent patters at the end of the mining process. The results shown in this section are based on the average of multiple runs for each case. In all experiments, *minsup* was expressed in terms of the absolute support value, and all trees were constructed using the ascending order of item value.

5.1 False Positives

Both the existing PUF-growth algorithm and our BLIMP-growth algorithm generate some false positives. Their overall performances depend on the number of false positives generated. In this experiment, we measured the number of false positives generated by all three algorithms for fixed values of *minsup* with different datasets. Due to space constraints, we present results using one *minsup* value for each of the two datasets (i.e., mushroom_5060 and u100k5L_10100) in Figs. 3(a)–(b). In general, BLIMP-growth was observed to remarkably reduce the number of false positives when compared with PUF-growth. The primary reason of this improvement is that the upper bounds for the BLIMP-growth algorithm are much tighter than PUF-growth for higher cardinality itemsets ($k > 2$), hence less total candidates are generated and subsequently less false positives. If fact, when existential probability values were distributed over a narrow range with a higher *minsup* as shown in Fig. 3(a), BLIMP-growth generated fewer than 1% of the total false positives of PUF-growth. Also note in Fig. 3(b) that for $k = 3$ itemsets, the total number of false positives in BLIMP-growth was much less (fewer than 20% in total, with fewer than 60% for $k = 4$ itemsets and fewer than 20% for higher cardinality levels) than that of PUF-growth. This happens because for lower cardinality (typically around $k = 3$) itemsets, the blimp value may actually be higher than the silver value. When different items from different transactions end up contributing to the silver value, the maximum value of a single item over all the transactions containing X is more likely to be higher than the maximum of all the second highest values (regardless of item) in those transactions. The probability of this happening decreases significantly with higher cardinality itemsets. As a reulst, BLIMP-growth had a runtime less than or equal to that of PUF-growth in every single experiment we ran.

5.2 Runtime

Recall that PUF-growth was shown to outperform UH-Mine [16] and subsequently UFP-growth [2, 19]. Hence, we compared our BLIMP-growth algorithm with PUF-growth. Figs. 3(c)–(d) show that BLIMP-growth had shorter runtimes than PUF-growth for datasets mushroom_5060 and u100k5L_10100. The primary reason is that, even though PUF-growth finds the exact set of frequent patterns when mining an extension of X, it may suffer from the high computation cost of generating unnecessarily large numbers of candidates due to only using two values in its item cap calculation: the existential probability of the suffix item and the single highest existential probability value in the prefix of x_r in t_j. This allows large amounts of high cardinality candidates to be generated with similar expected support cap values as low cardinality candidates with the same suffix item. The use of the blimp values in BLIMP-growth ensures that those high cardinality candidates are never generated due to their expected support caps being much closer to the actual expected support. Fig. 3(d) also shows that for low values of *minsup* BLIMP-growth had shorter runtimes. The primary reason is that for lower values of *minsup*, the number of high cardinality candidates

being generated increases. In this situation, the probability is higher that the blimp values in each node will actually be low, tightening the upper bound even further.

5.3 Scalability

Nowadays, high volume of high-variety and high-veracity vailable data can be collected and transmitted at high velocity. It becomes important to have a scalable algorithm to analyze these data. To test the scalability of BLIMP-growth, we applied the algorithm to mine frequent patterns from datasets with increasing size. The experimental results presented in Figs. 3(e)–(f) indicate that our algorithm (i) is scalable with respect to the number of transactions and (ii) can mine large volumes of uncertain data within a reasonable amount of time.

The experimental results show that our algorithms effectively mine frequent patterns from uncertain data irrespective of distribution of existential probability values (whether most of them have low or high values and whether they are distributed into a narrow or wide range of values).

6 Conclusions

In this paper, we proposed a scalable data analytics algorithm called *BLIMP-growth* to discover frequent patterns from uncertain data. The algorithm first constructs the *BLIMP-tree structure* to capture important information from uncertain data. The algorithm then finds all potentially frequent patterns (i.e., patterns with upper bounds to expected support $\geq$ user-defined *minsup* threshold) from this tree structure. As such a collection of potentially frequent patterns contains all truly frequent patterns as well as some false positives (i.e., patterns with upper bounds to expected support $\geq$ *minsup* but with expected support $<$ *minsup*). To ensure the scalability of the algorithm, BLIMP-growth reduces the number of false positives by obtaining tight upper bounds to expected supports. It does so by accumulating item caps with a blimp value (computed based on the maximum existential probability of a particular item) in the BLIMP-tree structures during the mining process. Hence, they further tighten the upper bound on expected supports of frequent patterns when compared to existing algorithms like PUF-growth. BLIMP-growth has both been shown to generate significantly fewer false positives than PUF-growth (in the range of 1% to 55% of the total value). In addition, with low values of *minsup* BLIMP-growth has been shown to generate fewer than 20% of the total false positives of PUF-growth. Our algorithms are guaranteed to find *all* frequent patterns (with *no* false negatives). Experimental results show the effectiveness of our BLIMP-growth algorithms in mining frequent patterns from the respective tree structures.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: VLDB 1994, pp. 487–499.

2. Aggarwal, C.C., Li, Y., Wang, J., Wang, J.: Frequent pattern mining with uncertain data. In: ACM KDD 2009, pp. 29–37.
3. Bernecker, T., Kriegel, H.-P., Renz, M., Verhein, F., Zuefle, A.: Probabilistic frequent itemset mining in uncertain databases. In: ACM KDD 2009, pp. 119–127.
4. Calders, T., Garboni, C., Goethals, B.: Approximation of frequentness probability of itemsets in uncertain data. In: IEEE ICDM 2010, pp. 749–754.
5. Calders, T., Garboni, C., Goethals, B.: Efficient pattern mining of uncertain data with sampling. In: PAKDD 2010, Part I. LNAI 6118, pp. 480–487.
6. Chui, C.-K., Kao, B., Hung, E.: Mining frequent itemsets from uncertain data. In: PAKDD 2007. LNAI 4426, pp. 47–58.
7. Frank, M., Dong, B., Felt, A.P., Song, D.: Mining permission request patterns from android and facebook applications. In: IEEE ICDM 2012, pp. 870-875.
8. Gao, C., Wang, J., Yang, Q.: Efficient mining of closed sequential patterns on stream sliding window. In: IEEE ICDM, pp. 1044-1049.
9. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: ACM SIGMOD 2000, pp. 1–12.
10. Krajca, P., Outrata, J., Vychodil, V.: Using frequent closed itemsets for data dimensionality reduction. In: IEEE ICDM 2011, pp. 1128-1133.
11. Lakshmanan, L.V.S., Leung, C.K.-S., Ng, R.T.: Efficient dynamic mining of constrained frequent sets. ACM TODS 28(4), 337–389 (2003)
12. Leung, C.K.-S.: Mining uncertain data. WIREs Data Mining and Knowledge Discovery 1(4), 316–329 (2011)
13. Leung, C.K.-S., Hao, B.: Mining of frequent itemsets from streams of uncertain data. In: IEEE ICDE 2009, pp. 1663–1670.
14. Leung, C.K.-S., Mateo, M.A.F., Brajczuk, D.A.: A tree-based approach for frequent pattern mining from uncertain data. In: PAKDD 2008. LNAI 5012, pp. 653–661.
15. Leung, C.K.-S., Tanbeer, S.K.: Fast tree-based mining of frequent itemsets from uncertain data. In: DASFAA 2012. LNCS 7238, pp. 272–287.
16. Leung, C.K.-S., Tanbeer, S.K.: PUF-Tree: A compact tree structure for frequent pattern mining of uncertain data. In: PAKDD 2013. LNCS 7818, pp. 13–25.
17. Patnaik, D., Laxman, S., Chandramouli, B., Ramakrishnan, N.: Efficient episode mining of dynamic event streams. In: IEEE ICDM 2012, pp. 605-614.
18. Soulet, A., Rassi, C., Plantevit, M., Cremilleux, B.: Mining dominant patterns in the sky. IEEE ICDM 2011, pp. 655-664.
19. Tong, Y., Chen, L., Cheng, Y., Yu, P.S.: Mining frequent itemsets over uncertain databases. PVLDB 5(11), 1650–1661 (2012)
20. Zhang, Q., Li, F., Yi, K.: Finding frequent items in probabilistic data. In: ACM SIGMOD 2008, pp. 819–832.