Efficient Incremental Mining of Contrast Patterns in Changing Data

James Bailey and Elsa Loekito

Department of Computer Science and Software Engineering
The University of Melbourne, Australia

Key words: Data mining, Contrast patterns, Databases

1. Introduction

A contrast pattern, also known as an emerging pattern [7], is an itemset whose frequency differs significantly between two classes of data. Such patterns describe differences between datasets and have been shown to be useful for building powerful classifiers [11, 9, 2, 8] [1]. Incrementally mining them in changing data is very important, where transactions can be inserted and deleted and mining needs to be repeated after changes occur. When the changes are small, the previously mined contrast patterns should be reused where possible, to compute the new patterns. A primary example of changing data is a data stream - a sequence of continuously arriving transactions (or itemsets). Mining of contrast patterns in a data stream is useful for stream classification [2] and network traffic change detection [4]. Work in [10] presented an algorithm to incrementally mine contrast patterns, but is oriented to updates of a single type. When a dataset changes due to insertion and deletion together, the efficiency of [10]'s approach is reduced, due to redundant computations. *In this paper, we present a new algorithm that addresses the scenario of incrementally mining contrast patterns in response to simultaneous insertion and deletion.* Our ap-

Email addresses: (jbailey,eloekito)@cs.mu.oz.au (James Bailey and Elsa Loekito)
[1] See http://www.cs.wright.edu/ gdong/EPC.html for a comprehensive bibliography

proach can be applied to any evolving dataset, but we particularly focus on data streams, a popular type of dataset for data mining (e.g. [6, 1]). The patterns of contrast correspond to itemsets which appear in the more recent transactions and not in the less recent ones.

A sliding window model [5] is a natural choice for determining which data to include for stream contrasts. Here, efficient incremental maintenance of the patterns in the sliding window is important. Arrival of new transactions deletes some of the oldest transactions from the window. This simultaneous transaction insertion and deletion may result in some interactions between these updates. The existing incremental technique for mining contrast patterns [10], performs well when changes of a single type occur in the input data, but has drawbacks when changes of multiple types simultaneously occur.

Contributions: We propose a new efficient technique for incrementally mining contrast patterns in scenarios where simultaneous insertions and deletions occur, such as in a data stream. We experimentally show it can deliver substantial speedups over the previous approach of [10].

2. Preliminary Definitions

Assume a database D, with attributes $\{A_1, A_2, \ldots, A_n\}$. Attribute A_i is defined by some values $domain(A_i)$. The set of all items I is the aggregate of all such domain values across attributes. $I = \bigcup_{i=1..n} domain(A_i)$. An *itemset* is a set of items in I. A *dataset* is a set of transactions, where each transaction is an itemset. For itemsets, P and Q, Q is a *superset* of P, or Q *contains* P, iff every item in P is also in Q. Given two sets of itemsets, X and Y, $X \cap_B Y$ denotes the border intersection of X and Y, namely all elements of X which are contained in at least one element of Y. The *support* of an itemset S in D, $support_D(S)$, is the number of transactions which contain S, divided by the number of transactions

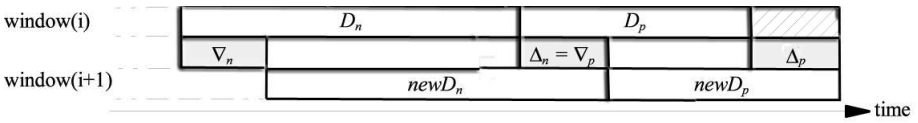

Figure 1: Sliding Window Model ($|\nabla_n| = |\Delta_n| = |\nabla_p| = |\Delta_p| = \delta$)

in D. A principal type of contrast pattern is the emerging pattern (EP) [7], which is an itemset whose support increases significantly from one dataset, labeled as the negative class, to another, labeled as the positive class. Let D_p and D_n be the positive and negative classes. The *growth rate* of S is the ratio of its support in D_p over that in D_n, i.e. $\frac{support_{D_p}(S)}{support_{D_n}(S)}$. We will focus on a particular type of pattern, known as the *Jumping Emerging Pattern (JEP)*[7], which is an EP with an infinite growth rate (i.e. $support_{D_n}(S) = 0$) and can capture sharp contrasts. A JEP is minimal if it is not a superset of other JEPs. Focusing on mining only the minimal JEPs helps reduce overlap in a set of patterns. Assume an operator JEP(D_p,D_n), which mines the minimal JEPs between D_p and D_n. It can be computed by existing algorithms such as [7, 3, 9], which can be treated as a "black box" [2]. The set of patterns which is output from mining may be used for either domain understanding or for the construction of models and (ensemble) classifiers. Although the set of mined patterns can be large, any specific test instance (transaction) is unlikely to match with many patterns from the ensemble. So with respect to any individual classification decision, the amount of redundancy amongst the matched patterns is likely to be small.

A *data stream* is a sequence of transactions. We define a sliding window of length k, which can be viewed as an evolving dataset. The window contains two sub-windows. The more recent sub-window is referred to as the *positive class* (i.e. D_p), and the less recent sub-window as the *negative class* (i.e. D_n), where $|D_p| + |D_n| = k$. See Fig. 1 for an illustration. In the current window

[2] In our experiments, we use an implementation of [3].

(window(i)), as JEP(D_p, D_n) is being computed, newly arriving transactions are *buffered* (labeled as Δ_p). When mining completes, the window is updated to include the buffered transactions. It now contains sub-windows $newD_p$ and $newD_n$, with the oldest transactions (labeled ∇_n) being removed. At the same time, some transactions of D_p are deleted from D_p (labeled as ∇_p) and inserted to $newD_n$ (labeled as Δ_n). Thus, $\nabla_p \equiv \Delta_n$, $newD_p = D_p^- \cup \Delta_p$, and $newD_n = D_n^- \cup \Delta_n$, where $D_p^- = D_p - \nabla_p$ and $D_n^- = D_n - \nabla_n$. The increment size, δ, is the number of deleted and inserted transactions in each class, i.e. $|\nabla_n| = |\Delta_n| = |\nabla_p| = |\Delta_p| = \delta$, and $\delta << k$.

3. Efficient Incremental JEP Maintenance Algorithms

Our objective is to (incrementally) compute JEP($newD_p, newD_n$), given the inputs: JEP(D_p, D_n) and updates $\nabla_p, \nabla_n, \Delta_p, \Delta_n$. We will shortly describe our algorithm **ExclusiveIncremental**. First we provide some intuition about the limitations of the existing incremental technique from [10], known as LMDR.

LMDR processes the updates sequentially in four steps: It incrementally mines the patterns after each of the updates 1) ∇_n, followed by 2) Δ_n, then 3) ∇_p and then 4) Δ_p. Patterns may be added at Steps 1) and 4), while existing patterns may be deleted at Steps 2) and 3). The drawback is that the effect of each update is evaluated independently of the other updates. This means that redundant computations may occur, whereby certain patterns appear due to an earlier update, only to be eliminated due to a later update (or vice versa).

For example, ∇_n and Δ_n may contain transactions which intersect. Consider two transactions $\{a, b, c, d\} \in \nabla_n$ and $\{c, d, e, f\} \in \Delta_n$. These intersect on the itemset $\{c, d\}$. Now $\{c, d\}$ might be a new pattern which is inserted after processing ∇_n, but which then gets deleted after subsequently processing update Δ_n. Similar interactions can occur with respect to the pairs of updates ∇_n plus

Algorithm 1 ExclusiveIncremental: mine contrasts w.r.t. $newD_p$ and $newD_n$

$J_0 = \text{JEP}(D_p, D_n)$, $D_p^- = D_p - \nabla_p$, $D_n^- = D_n - \nabla_n$, $newD_p = D_p^- \cup \Delta_p$, $newD_n = D_n^- \cup \Delta_n$

***LMDR*$^+$ vs *ExclusiveIncremental*$^+$**$(J_0, newD_p, D_n)$**:** compute $\text{JEP}(newD_p, D_n)$

LMDR$^+$:	*ExclusiveIncremental*$^+$:
1: $J_{retain} \leftarrow J_0 \cap_B D_p^-$	1: $J'_{retain} \leftarrow J_0 \cap_B newD_p$
2: $J_{appear} \leftarrow \text{JEP}(\Delta_p, D_n)$	2: $J'_{appear} \leftarrow \text{JEP}(\Delta_p, (D_p \cup D_n))$
3: $\text{JEP}(newD_p, D_n) \leftarrow J_{retain} \cup J_{appear}$	3: $\text{JEP}(newD_p, D_n) \leftarrow J'_{retain} \cup J'_{appear}$

***LMDR*$^-$ vs *ExclusiveIncremental*$^-$**$(J_0, D_p, newD_n)$**:** compute $\text{JEP}(D_p, newD_n)$

LMDR$^-$:	*ExclusiveIncremental*$^-$:
1: $J_n \leftarrow J_0 \cup (\text{JEP}(\nabla_n, D_n^-) \cap_B D_p))$	1: $J_{invalid} \leftarrow J_0 \cap_B \Delta_n$
2: $J_{disappear} \leftarrow \text{JEP}(D_p, \Delta_n)$	2: $J_{retain} \leftarrow J_0 - J_{invalid}$
3: $\text{JEP}(D_p, newD_n) \leftarrow$	3: $J_{new} \leftarrow \text{JEP}(\nabla_n, newD_n) \cap_B D_p$
$(J_n \times J_{disappear}) \cap_B D_p$	4: $J_{disappear} \leftarrow \text{JEP}(D_p, \Delta_n)$
	5: $\text{JEP}(D_p, newD_n) \leftarrow J_{retain} \cup J_{new}$
	$\cup ((J_{invalid} \times J_{disappear}) \cap_B D_p)$

***ExclusiveIncremental*$(J_0, newD_p, newD_n)$:** compute $\text{JEP}(newD_p, newD_n)$

1: $J'_0 \leftarrow$ *ExclusiveIncremental*$^-(J_0, D_p^-, newD_n)$

2: $\text{JEP}(newD_p, newD_n) \leftarrow$ *ExclusiveIncremental*$^+(J'_0, newD_p, newD_n)$

∇_p, and ∇_p plus Δ_p. In contrast, our new algorithm avoids unnecessary work by taking into account the interactions between updates.

Our algorithm (ExclusiveIncremental) and a comparison to LMDR are shown in Algorithm 1. The logic is divided into two subalgorithms, one processing updates to D_n and the other processing updates to D_p. ExclusiveIncremental is both correct (only mines valid patterns) and complete (mines all valid patterns).

Updating the Positive Class (*ExclusiveIncremental*$^+$). Upon deletion of ∇_p, *LMDR*$^+$ keeps the old JEPs occurring in D_p^- (represented by J_{retain}, line 1). Subsequently, insertion of Δ_p may cause new JEPs to appear, which are JEPs for Δ_p versus D_n (represented by J_{appear}, line 2). However, J_{retain} and J_{appear} may overlap, and so J_{appear} may contain some old JEPs which were removed from J_{retain}. To remove this cancellation, our algorithm finds both J'_{retain}, which includes the old JEPs which occur in Δ_p (line 1), and J'_{appear} which excludes new JEPs which occur in D_p (line 2); $(J'_{retain} \cap J'_{appear}) = \emptyset$.

Updating the Negative Class (*ExclusiveIncremental*$^-$). Upon deletion of ∇_n, *LMDR*$^-$ finds the newly occurring JEPs by finding itemsets of $\text{JEP}(\nabla_n, D_n^-)$

which occur in D_p (represented by J_n, line 1). Such itemsets do not exist in J_0. Then, the insertion of Δ_n is processed by finding the pair-wise union between J_n and $J_{disappear} = \text{JEP}(D_p, \Delta_n)$, to find all the minimal JEPs which occur in D_p, but not in D_n^- nor Δ_n (line 3). However, some itemsets in J_n may remain in the output. They are itemsets which occur in D_p but not in ∇_n. Some of the old JEPs which were removed from J_n, moreover, may re-appear. This cancellation is removed in our algorithm. Firstly, all of the old JEPs which do not occur in Δ_n are kept (line 1-2). Then, the new JEPs are found from the deleted transactions in ∇_n, which do not occur in D_n^- nor Δ_n (line 3). Finally, only old JEPs which occur in Δ_n (represented by $J_{invalid}$) are involved in the pair-wise union (line 5). $J_{invalid}$ is a smaller set than J_n.

Updating Both the Positive and Negative Classes (*ExclusiveIncremental*). When incremental changes occur to both classes, the two algorithms can be combined. We begin with *ExclusiveIncremental*$^-$() for handling the changes in D_n, then pass its output to *ExclusiveIncremental*$^+$(). Since $\nabla_p = \Delta_n$ in a sliding window, references to D_p in the first routine can be substituted by D_p^-. This avoids generating patterns which occur in ∇_p. Line 1 in *ExclusiveIncremental*$^+$() can be replaced by $J'_{retain} = J_0$ for a sliding window.

4. Performance Study and Discussion

We compare the performance of our *ExclusiveIncremental* algorithm, against *LMDR* [10], and also a "*Naive*" algorithm which mines the JEPs from scratch in every window. We use three data stream datasets: DATA 1: KDD-CUP 1999, DATA 2: KDD-CUP 2000 and DATA 3: a synthetic dataset generated by the IBM data generator[3]. Tables 1 and 2 show the characteristics of each

[3] `http://www.almaden.ibm.com`. DATA 3 was generated with parameter '-g0 0.3 -p 10 -shake 5'; *-shake* models the fuzzy boundary between the classes; *-g0* and *-p* determine the class distribution. All continuous attributes are discretized by equi-width binning.

Dataset	DATA 1	DATA 2	DATA 3
Source	KDD'99	KDD'00	Synthetic
#attributes	20	30	10
#items	200	600	50

Table 1: Data Characteristics

S	A	B	C	D	E
$\|D_n\|$	1000	1000	5000	5000	10000
$\|D_p\|$	200	200	500	1000	1000
δ	10	20	100	100	100

Table 2: Scenarios (S = Scenario)

Dataset	S	$\|J_0\|$	Incremental Mining Time (seconds)					
			PosIncremental		NegIncremental		PosNegIncremental	
			Naive$^+$	*ExInc*$^+$ (speed-up)	*Naive*$^-$	*ExInc*$^-$ (speed-up)	*Naive*	*ExInc* (speed-up)
DATA 1	A	10	0.05	**0.01** (5.00)	0.12	**0.04** (3.00)	0.17	**0.01** (17.00)
	B	2	**0.02**	**0.02** (1.00)	**0.12**	0.17 (0.71)	0.14	**0.03** (4.67)
	C	2	0.42	**0.08** (5.25)	0.50	**0.36** (1.39)	0.92	**0.15** (6.13)
	D	5	0.28	**0.07** (4.00)	1.12	**0.53** (2.11)	1.40	**0.21** (6.67)
	E	8	0.68	**0.14** (4.86)	2.45	**1.02** (2.40)	3.13	**0.26** (12.04)
DATA 2	A	85	1.61	**0.11** (14.64)	3.03	**0.54** (5.61)	4.64	**0.19** (24.42)
	B	145	1.36	**0.43** (3.16)	3.85	**1.06** (3.63)	5.21	**0.59** (8.83)
	C	362	29.97	**7.05** (4.25)	59.60	**28.81** (2.07)	88.57	**18.62** (4.76)
	D	576	78.18	**8.90** (8.78)	**45.95**	53.39 (0.86)	124.13	**27.11** (4.58)
	E	1005	63.18	**15.47** (4.08)	258.25	**86.42** (2.99)	321.43	**37.64** (8.54)
DATA 3	A	1333	**0.30**	1.47 (0.20)	1.15	**0.49** (2.35)	1.45	**1.91** (0.76)
	B	1357	**0.32**	2.69 (0.12)	1.17	**1.00** (1.17)	**1.49**	3.06 (0.49)
	C	1680	**3.07**	7.73 (0.40)	9.87	**8.31** (1.19)	12.94	**11.20** (1.16)
	D	3149	**11.41**	12.38 (0.92)	20.24	**16.66** (1.21)	31.65	**22.22** (1.42)
	E	2182	**11.64**	11.80 (0.99)	29.36	**20.37** (1.44)	41.00	**15.96** (2.57)

Table 3: Runtime Comparison Against the *Naive* Algorithm; *ExInc* = *ExclusiveIncremental*; speed-up = $\frac{\text{runtime of } Naive}{\text{runtime of } ExclusiveIncremental}$

data stream, and the scenarios used. All experiments used a 2.4GHz CPU with 3GB RAM running Solaris. Tables 3 and 4 show the runtimes of each algorithm, averaged over 100 windows. *ExInc* refers to our *ExclusiveIncremental* algorithm. *PosIncremental* shows the runtimes for processing the positive class changes, *NegIncremental* for processing the negative class changes, and *PosNegIncremental* for processing simultaneous changes to both classes.

Compared to *Naive*, our algorithm is usually faster, depending (as expected) on the degree of change undergone by the window. For handling the changes in the positive class, *ExInc*$^+$ is up to 8 times faster than *Naive*$^+$, for which D_p and D_n have a large size and the increment is small, and it is up to 3 times faster than *LMDR*$^+$ for DATA 1 and DATA 2. For DATA 3 however, since it is dense (i.e. contains a small number of items but a large number of patterns), *ExInc*$^+$ is slower than both *Naive*$^+$ and *LMDR*$^+$, except for scenario D where the number

Dataset	S	Incremental Mining Time (seconds)					
		PosIncremental		NegIncremental		PosNegIncremental	
		$LMDR^+$	$ExInc^+$ (speed-up)	$LMDR^-$	$ExInc^-$ (speed-up)	$LMDR$	$ExInc$ (speed-up)
DATA 1	A	**0.01**	**0.01** (1.00)	0.06	**0.04** (1.50)	0.07	**0.01** (7.00)
	B	**0.02**	**0.02** (1.00)	**0.01**	0.17 (0.06)	**0.03**	**0.03** (1.00)
	C	0.19	**0.08** (2.38)	**0.23**	0.36 (0.64)	0.42	**0.15** (2.80)
	D	0.19	**0.07** (2.71)	**0.20**	0.53 (0.38)	0.39	**0.21** (1.86)
	E	0.44	**0.14** (3.14)	**0.33**	1.02 (0.32)	0.77	**0.26** (2.96)
DATA 2	A	0.27	**0.11** (2.45)	20.61	**0.54** (38.17)	20.88	**0.19** (109.89)
	B	0.82	**0.43** (1.91)	78.42	**1.06** (73.98)	79.24	**0.59** (134.31)
	C	24.34	**7.05** (3.45)	871.13	**28.81** (30.24)	895.47	**18.62** (43.26)
	D	28.55	**8.90** (3.21)	1482.7	**53.39** (27.77)	1511.3	**27.11** (55.75)
	E	52.01	**15.47** (3.36)	3217.8	**86.42** (37.23)	3269.8	**37.64** (86.87)
DATA 3	A	**1.42**	1.47 (0.97)	1255.3	**0.49** (2561.86)	1256.7	**1.91** (657.97)
	B	**2.18**	2.69 (0.81)	5512.7	**1.00** (5512.70)	5514.9	**3.06** (1802.25)
	C	**5.78**	7.73 (0.75)	22441	**8.31** (2700.48)	22447	**11.20** (2004.11)
	D	17.89	**12.38** (1.45)	40225	**16.66** (2414.47)	40243	**22.22** (1811.12)
	E	**11.18**	11.80 (0.95)	49010	**20.37** (2405.99)	49021	**15.96** (3071.49)

Table 4: Runtime Comparison Against the *LMDR* [10] algorithm; *ExInc* = *ExclusiveIncremental*; speed-up = $\frac{\text{runtime of } LMDR}{\text{runtime of } ExclusiveIncremental}$

of patterns is the largest. The results also show that $LMDR^+$ performs better than $Naive^+$ in all scenarios for DATA 1 and DATA 2.

For processing changes in the negative class, $ExInc^-$ is up to 5 times faster than $Naive^-$, but $LMDR^-$ is the fastest in most scenarios for DATA 1, which is sparse and contains less patterns. $ExInc^-$ is less efficient in this scenario. When the number of patterns is large, such as in DATA 2 and DATA 3, $LMDR^-$ is slower than $Naive^-$, and $ExInc^-$ is up to 5000 times faster than $LMDR^-$, especially for the dense DATA 3. In those scenarios, the pair-wise union operation performed by $LMDR^-$ (line 3 in Algorithm 1) is computationally expensive, due to the large number of patterns involved.

Finally, for handling changes in both classes, *ExInc* has the fastest runtimes, followed by *LMDR*, and *Naive*, for DATA 1. For DATA 2 and DATA 3, where *LMDR* performs poorly because of the slow runtime of $LMDR^-$, *ExInc* is able to outperform LMDR by up to 5000 times.

References

[1] Aggarwal, C. C., 2003. A framework for diagnosing changes in evolving data streams. In: Int'l Conf. on Management of Data. San Diego, CA, pp. 575–586.

[2] Alhammady, H., Ramamohanarao, K., 2005. Mining emerging patterns and classification in data streams. In: Int'l Conference on Web Intelligence. pp. 272–275.

[3] Bailey, J., Manoukian, T., Ramamohanarao, K., 2003. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In: Proc. 3rd ICDM. Melbourne, Florida, pp. 485–488.

[4] Ben-David, S., Gehrke, J., Kifer, D., 2004. Detecting change in data streams. In: Proc. 30th VLDB Conference. Toronto, Canada, pp. 180–191.

[5] Chang, J. H., Lee, W. S., 2004. A sliding window method for finding recently frequent itemsets over online data streams. J. of Information Science and Engineering 20, 753–762.

[6] Dong, G., et al., 2003. Online mining of changes from data streams. In: Workshop on Management and Processing of Data Streams.

[7] Dong, G., Li, J., 1999. Efficient mining of emerging patterns: Discovering trends and differences. In: Proc. of KDD. San Diego, CA, pp. 43 – 52.

[8] Dong, G., Zhang, X., Wong, L., Li, J., 1999. Caep: Classification by aggregating emerging patterns. In: Discovery Science. pp. 30–42.

[9] Li, J., Liu, G., Wong, L., 2007. Mining statistically important equivalence classes and delta-discriminative emerging patterns. In: Proc. of KDD. pp. 430–439.

[10] Li, J., Manoukian, T., Dong, G., Ramamohanarao, K., 2004. Incremental maintenance on the border of the space of emerging patterns. DMKD 9 (1), 89–116.

[11] Li, J., Ramamohanarao, K., Dong, G., 2000. Making use of the most expressive jumping emerging patterns for classification. In: Proc. 4th PAKDD. Kyoto, Japan, pp. 220–232.