Automatic Assignment of Item Weights for Pattern Mining on Data Streams

Yun Sing Koh[1], Russel Pears[2], and Gillian Dobbie[1]

[1] Department of Computer Science, University of Auckland, New Zealand
{ykoh,gill}@cs.auckland.ac.nz
[2] School of Computing and Mathematical Sciences, AUT University, New Zealand
rpears@aut.ac.nz

Abstract. Research in Weighted Association Rule Mining (WARM) has largely concentrated on mining traditional static transactional datasets. Whilst there have been a few attempts at researching WARM in a data stream environment, none have addressed the problem of assigning and adapting weights in the presence of concept drift, which often occurs in a data stream environment. In this research we experiment with two methods of adapting weights; firstly, a simplistic method that recomputes the entire set of weights at fixed intervals, and secondly a method that relies on a distance function that assesses the extent of change in the stream and only updates those items that have had significant change in their patterns of interaction. We show that the latter method is able to maintain good accuracy whilst being several times faster than the former.

Keywords: Weighted Items, Valency Model, Data Stream Mining.

1 Introduction

Ever since its inception, data stream mining has remained one of the more challenging problems within the data mining discipline. Although extensively researched, many unsolved problems remain. Yet streams are an increasingly important and rich source of data that can yield valuable knowledge when mined effectively. Data from a wide variety of application areas ranging from online retail applications such as online auctions and online bookstores, telecommunications call data, credit card transactions, sensor data and climate data are but a few examples of applications that generate vast quantities of data on a continuous basis. Furthermore, data produced by such applications are highly volatile with new patterns and trends emerging on a continuous basis.

Association Rule Mining (ARM) is one approach to mining such data streams. However a well known limitation of ARM is that it has the potential to generating a large number of rules, most of which are trivial or of no interest to the decision maker. One method of overcoming this problem is to weight items in terms of importance so that rules that only contain high weight items are presented to the user. The crucial factor in ensuring the success of this approach is the

J.Z. Huang, L. Cao, and J. Srivastava (Eds.): PAKDD 2011, Part I, LNAI 6634, pp. 387–398, 2011.

assignment of weights to items. The typical approach is for users to supply the weights through a subjective process based on their specialized knowledge of the domain involved. While this subjective weight assignment may be feasible in an environment where data is stable, we believe that it will not be effective in the context of data stream mining where data is subject to constant change. Such change or concept drift is bound to invalidate any weight assignment over a period of time. In this research we thus propose an algorithm to adapt item weights to reflect changes that take place in the data stream.

Recent research by Koh et al. [1] has shown that automatic inference of weights from a transaction graph is an effective method of ranking items according to their interest measure, or Valency. The Valency model assigns a weight to an item depending on how strongly it interacts with other items. However the Valency model was not designed to operate in a dynamic environment and in this research we focus on extending the Valency model to adapt weights by capturing the changing patterns of interaction over time.

The rest of the paper is organized as follows. In the next section we review work that has been done on item weighting and briefly cover work done on incremental methods employed in a data stream environment. In Section 3 we describe the Valency model in greater detail, while in Section 4 we describe our methodology for incremental update of weights. Our experimental results are presented in Section 5. The paper concludes in Section 6 where we assess to what extent we have achieved our goal of evolving weights over a stream as well as presenting our thoughts for future research in this area.

2 Background and Related Work

There has been much work in the area of pattern mining in data streams [2,3,4]. However, there has been very little research specifically directed at mining weighted patterns in a data stream environment. The few attempts to address this problem have not employed automatic methods for item weight assignment and maintenance.

Ahmed et al. [5] proposed a sliding window based technique WFPMDS (Weighted Frequent Pattern Mining over Data Streams) which employs a single pass through the data stream. They utilized an FP tree to keep track of the weighted support of items and hence their mining scheme was essentially based on the FP tree algorithm with the major difference being that weighted support was used in place of raw support. Kim et al. [6] proposed a weighted mining scheme based on two user defined thresholds $t1$ and $t2$ to divide items into infrequent, latent and frequent categories. Items with *weighted support* $< t1$ were categorized into the infrequent category and those with *weighted support* $> t2$ were grouped into the frequent category; all others were taken to be latent. Items in the infrequent category were pruned while items in the other two categories were retained in a tree structure based on an extended version of the FP tree. However in common with Ahmed et al., their weight assignment scheme was both subjective and static, and as such would degrade in the face of concept

drift that occurs in many real world situations. When concept drift occurs such mining schemes essentially take on the character of frequent pattern mining as items are not re-ranked in terms of importance, and changes in their weighted support only reflect changes taking place in support, rather than item rank.

The main challenge to be overcome in weighted association rule mining over a stream is to be able to adapt item weights according to the changes that take place in the environment. Such changes force adjustments to previously assigned weights in order to make them better reflect the new data distribution patterns. We argue that it is not possible for domain experts to play the role that they normally do in static data environments. Firstly, due to the open ended nature of a data stream, it will be very difficult for a human to constantly keep updating the weights on a regular basis. Secondly, even if they regularly invest the time to keep the weights up to date, it will be very hard, if not impossible to keep track of changes in the stream for applications such as click stream mining, fraud detection, telecommunications and online bookstore applications, where the number of items is very large and the degree of volatility in the data can be high. Our weight assignment scheme is based on the Valency model due to its success in generating high quality rules as reported in [1]. However, as connectivity between items plays a critical role in the Valency model, it is necessary to monitor the stream and identify which items have had significant changes in their interactions with other items in order to efficiently adapt the weights. In a stream containing N items, a naive method of performing such an estimation would be to check the interactions between all pairs of items which has a worst case time complexity of $O(N^2)$. With N having values in the tens of thousands or even hundreds of thousands, such an approach would be extremely inefficient or even prohibitive in the case of high speed data streams. Thus the challenge boils down to finding an efficient and accurate estimation method that has a worst case time complexity closer to the ideal value of $O(N)$. We describe such a method in Section 4.

3 Valency Model

The Valency model, as proposed by Koh et al. is based on the intuitive notion that an item should be weighted based on the strength of its connections to other items as well as the number of items that it is connected with. Two items are said to be connected if they have occurred together in at least one transaction. Items that appear often together relative to their individual support have a high degree of connectivity and are thus weighted higher. The total connectivity c_k of item k which is linked to n items in its neighborhood is defined as:

$$c_k = \sum_{i}^{n} \frac{count(ki)}{count(k)} \tag{1}$$

The higher the connectivity c_k of a given item k, the higher its weight should be, and vice versa. While high connectivity is a necessary condition for a high weight,

it is not considered to be sufficient by itself, as the weighting scheme would then be too dependent on item support which would bias weighting too much towards the classical (un-weighted) association rule mining approach. With this in mind, a purity measure was defined that encapsulated the degree to which an item could be said to be distinctive. The smaller the number of items that a given item interacted with, the higher the purity and vice versa. The reasoning here is that an item should not be allowed to acquire a high weight unless it also has high purity, regardless of its connectivity. The role of purity was thus to ensure that only items with high discriminative power could be assigned a high weight, thus reducing the chances of cross support patterns from manifesting.

Formally, the purity of a given item k was defined as:

$$p_k = 1 - \frac{\log_2(|I_k|) + \log_2(|I_k|)^2}{\log_2(|U|)^3} \tag{2}$$

where $|U|$ represents the number of unique items in the dataset and $|I_k|$ represents the number of unique items which are co-occurring with item k. Purity as defined in Equation 2 ensures that the maximum purity of 1 is obtained when the number of items linked with the given item is 1, whereas the purity converges to the minimum value of 0 as the number of linkages increases and becomes close to the number of items in the universal set of items. The logarithmic terms ensure that the purity decreases sharply with the number of linkages in a non linear fashion.

The Valency contained by an item k, denoted by v_k as the combination of both the purity and the connectivity components is defined below:

$$v_k = \beta.p_k + (1 - \beta).\sum_{i}^{n} \frac{count(ki)}{count(k)}.p_i \tag{3}$$

where β is a parameter that measures the relative contribution of the item k over the items that it is connected with in the dataset and is

$$\beta = \frac{1}{n}\sum_{i}^{n} \frac{count(ik)}{count(k)}$$

We use the Valency contained by an item as its weight.

4 Our Weight Adaptation Methodology

We use a sliding window mechanism to keep track of the current state of the stream. Our weight adaptation scheme uses a distance function to assess the extent of change in the stream. In order to keep overheads within reasonable bounds the distance computation is only applied periodically, or after every b number of instances (i.e. a block of data of size b) are processed. Thus a block consists of a number of overlapping windows.

We now present a 2-phased approach to capturing interactions between items and detecting change points in the data stream. In phase 1 an initial block of

data is read and item neighborhoods are built in the form of 1-level trees in order for an initial assignment of weights to be made. In the second phase, weights are adapted through the use of a distance function as subsequent blocks of data arrive in the data stream.

In phase 1 (Read in initial block): Each transaction is read and items are sorted in their arrival order in the stream. Whenever a new item is detected, a 1-level tree is created with that item as its root. Figure 1 shows the trees created for a transaction containing 4 items, A B, C, and D. The 1-level trees so formed enable easy computation of the connectivity and purity values for each item. Once the purity and connectivity is calculated for an item its weight is then computed by applying Equation 3.

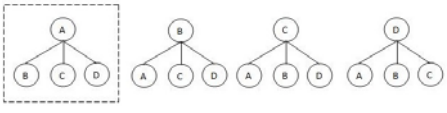

Fig. 1. Graph Node

In phase 2 (Read in subsequent blocks): The oldest transaction in the window is deleted from the inverted index matrix and the new transaction is inserted. At the conclusion of the current block a distance function is applied to each item to determine whether or not its joint support with neighboring items (i.e. items that it occurs with) has changed significantly enough, in which case it is recomputed from a buffer containing transactions in the current block. The update to the joint support of an item A with another item B triggers an update to the connectivities of both items, A and B. The purity of an item is only updated if a new item is introduced or if an item disappears from the current block. We recalculate weights for any items that have had their connectivity or purity values changed.

4.1 Data Structure: Inverted Index Matrix

We use an inverted index to buffer transactions in the current block. Transactions are represented by a prefix tree structure. The index key consists of the item id. The support of the item is also stored with the index key in order to speed up retrieval. Each index value consists of a triple $< i, j, k >$ where i is a pointer to the next key, j is the tree prefix that the item occurs in, and k is the support of the item with respect to tree j. Given an initial transaction (A, B, C, D, E), arriving in this order, Table 1(a) shows the initial entry, whereby the initial tree id is 1. Given the arrival of another transaction (B, A, C, F), we now sort this transaction into the order: (A, B, C, F). Here the entry for $A(2, 1, 1)$ is updated to (2,1,2); this is repeated for each subsequent entry until item C where an entry (6, 1, 1) is added to slot 2 of the index value array to point to its successor F. A new slot is required for entry (6,1,1) as this represents a new branch in the prefix tree structure. Table 1(b) shows the updated index after the arrival of the second transaction.

Table 1. Inserting into the modified Inverted Matrix

(a) Before Insertion of transaction (B, A, C, F)

loc	Index	Transactional Array	
		1	2
1	(A,1)	(2,1,1)	
2	(B,1)	(3,1,1)	
3	(C,1)	(4,1,1)	
4	(D,1)	(5,1,1)	
5	(E,1)	(ϕ,1,1)	

(b) After Insertion

loc	Index	Transactional Array	
		1	2
1	(A,2)	(2,1,2)	
2	(B,2)	(3,1,2)	
3	(C,2)	(4,1,1)	(6,1,1)
4	(D,1)	(5,1,1)	
5	(E,1)	(ϕ,1,1)	
6	(F,1)	(ϕ,1,1)	

Table 2. Deleting from the modified Inverted Matrix

(a) Before Deletion

loc	Index	Transactional Array	
		1	2
1	(A,4)	(2,1,3)	(4,2,1)
2	(B,4)	(3,1,3)	(6,3,1)
3	(C,3)	(4,1,2)	(6,1,1)
4	(D,2)	(5,1,1)	(5,2,1)
5	(E,2)	(ϕ,1,1)	(ϕ,2,1)
6	(F,2)	(ϕ,1,1)	(ϕ,3,1)

(b) After Deletion

loc	Index	Transactional Array	
		1	2
1	(A,3)	(2,1,2)	(4,2,1)
2	(B,3)	(3,1,2)	(6,3,1)
3	(C,2)	(4,1,2)	
4	(D,2)	(5,1,1)	(5,2,1)
5	(E,2)	(ϕ,1,1)	(ϕ,2,1)
6	(F,1)	(ϕ,3,1)	

The inverted matrix structure supports efficient deletion of transactions as well. Table 2 shows the state of the index after the arrival of a number of additional transactions. Given the oldest transaction (A, B, C, F) in the current window, with tree id 1, we find the position of A with tree id 1 and reduce the count corresponding to this position by one. Here the entry for $A(2, 1, 2)$ is updated to (2,1,1); this is repeated for each subsequent entry until item F is encountered. As the count for the F(ϕ,1,1) is updated to (ϕ,1,0), this entry is removed from the matrix.

The inverted matrix structure is used to buffer transactions across only two blocks, since our distance and estimation functions only require the comparison of the state of the current block with that of the previous block. Another advantage of using this data structure is the efficiency in finding the joint support between two items. Given two items, A and C, the joint support AC, is obtained by carrying out an intersection operation between the items based on their tree ids. For each tree that this pair occurs in we extract the minimum of their tree count values and accumulate this minimum across all trees that this pair participates in. As A and C appear in only one tree with tree id 1, the minimum count across tree 1 is taken, which happens to be 2.

The inverted matrix while being efficient at computing the joint support of item pairs does not efficiently support the maintenance of an item's connectivity. To update the connectivity of an item, its neighborhood needs to be enumerated. This would require excessive traversal of the inverted matrix. Hence we also maintain one level trees for each item, as described earlier. For every new item which appears in the stream, an entry is added to the inverted matrix and a link is established to its one level tree where that item appears as a root node.

4.2 Distance Function

To calculate the weights for an item using the Valency model mentioned in Section 3, we track the joint support of an item with other items in its neighborhood. In order to prevent expensive and unnecessary updates, we use a distance function $d(X, Y)$ to assess the extent of change in joint support between a given item X with another item Y. Let $S_n(X)$ be the actual support of item X in block n and $\hat{S}(X)_n$ be its support conditional on no change taking place in the connectivity between it and item Y in consecutive windows $n-1$ and n. The distance function is then given by:

$$d(X, Y) = |\hat{S}(X)_n - \mathrm{S}(X)_n|$$

where $\hat{S}(X)_n$ is given by:

$$\hat{S}(X)_n = \mathrm{S}(XY)_{n-1} \frac{\mathrm{S}(Y)_n}{\mathrm{S}(X)_n} \frac{\mathrm{S}(X)_{n-1}}{\mathrm{S}(Y)_{n-1}} \frac{1}{\mathrm{C}(X, Y)_{n-1}}$$

where n is the current block number.

Rationale. Under the assumption of no pattern drift in the connectivity between X and Y we have:

$$\mathrm{C}(X, Y)_n = \mathrm{C}(X, Y)_{n-1}$$

where $\mathrm{C}(X, Y)_{n-1}$, $\mathrm{C}(X, Y)_n$ are the connectivities between X and Y in blocks n-1 and n respectively. We estimate:

$$\hat{S}(X)_n = \frac{\hat{S}(XY)_n}{\mathrm{C}(X, Y)_n} = \frac{\hat{S}(XY)_n}{\mathrm{C}(X, Y)_{n-1}}$$

In the absence of concept drift the joint support between X and Y will remain stable only if the factor

$$\frac{S(Y)_n}{S(Y)_{n-1}} \frac{S(X)_{n-1}}{S(X)_n}$$

remains stable between blocks. Our intention is to trap any significant changes to the join support arising out of significant changes to this factor. Under the assumption of stability in this factor and no concept drift between X and Y we have:

$$\hat{S}(XY)_n = S(XY)_{n-1} \frac{S(Y)_n}{S(Y)_{n-1}} \frac{S(X)_{n-1}}{S(X)_n}$$

and so we reformulate $\hat{S}(X)_n$ as:

$$\hat{S}(X)_n = \mathrm{S}(XY)_{n-1} \frac{\mathrm{S}(Y)_n}{\mathrm{S}(X)_n} \frac{\mathrm{S}(X)_{n-1}}{\mathrm{S}(Y)_{n-1}} \frac{1}{\mathrm{C}(X, Y)_{n-1}}$$

We now have an estimation of the support of X that is sensitive to changes in the stream. Any significant departure from our assumption of no drift will now

be trapped by the distance function $d(X,Y) = |\hat{S}(X)_n - \mathrm{S}(X)_n|$ since it includes the drift sensitive term $\hat{S}(X)_n$.

In order to assess which values of $d(X,Y)$ are significant we note that the function is a difference between the sample means of two random variables which are drawn from unknown distributions. In order to assess significance between the means we make use of the Hoeffding bound [7]. The Hoeffding bound is attractive as it is independent of the probability distribution generating the observations. The Hoeffding bound states that, with probability $1-\delta$, the true mean of a random variable, r, is at least $\overline{r} - \epsilon$ when the mean is estimated over t samples, where

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2t}}$$

R is the range of r. In our case the variable r is denoted by $\hat{S}(X)_n - \mathrm{S}_n(X)$, which has a range value R of 1, and the number of samples $t = b$, the block size. Thus for any value of $d(X,Y) > \epsilon$ our assumption of no concept drift is violated and an update of the join support $S_n(XY)$ is required from the transactions in our buffer for block n.

However we were also conscious of the fact that even small changes in the joint support $S_n(XY)$ not signalled by the distance function can accumulate over all items Y in item X's neighborhood. Although each of the deviations are small individually they could become significant when added over all the links between X and its neighbors. We thus decided to add a correction factor (but not an update) to the joint support even in the event that $d(X,Y) \leq \epsilon$. Our experimentation showed the importance of adding this correction factor, with it in place the precision of identifying high weight items increased quite significantly.

The correction factor is given by:

$$S(XY)_n = \mathrm{S}(XY)_{n-1} + \mathrm{S}(XY)_{n-1} * \frac{(\mathrm{S}(X)_n - \hat{S}(X)_n) * \mathrm{C}(X,Y)_{n-1}}{2}$$

The correction factor basically adds or subtracts, as the case may be, a factor to the joint support that is equal to the product of the connectivity with the median value of the deviation between $\mathrm{S}(X)_n$ and $\hat{S}(X)_n$. We carry out a complete update of the joint support of every pair of items after p number of blocks. This is to ensure that the estimation remains within a reasonable range. The parameter p has to be set at a reasonable value to achieve a good balance between precision and efficiency. In all our experimentation we set p to 10.

5 Evaluation

We report on a comparative analysis of our incremental approach, referred to as *WeightIncrementer*, with the simple approach of updating all item weights periodically at each block of data, referred to as *Recompute*. Our experimentation used both synthetic and real world datasets. We compared both approaches

on execution time. In addition, we tracked the precision of WeightIncrementer relative to Recompute. Using Recompute we tracked the identity of items having weights in the top k% and used the identity of these items to establish the precision of WeightIncrementer. The precision is given by: $\frac{|WI \bigcap R|}{|R|}$ where WI is the set of items returned by WeightIncrementer as its top $k\%$ of items; R is the set of items returned by Recompute as its top $k\%$ of items.

Our experimentation on the synthetic data was conducted with the IBM dataset generator proposed by Agrawal and Srikant [8]. To create a dataset D, our synthetic data generation program takes the following parameters: number of transactions $|D|$, average size of transactions $|T|$, average size of large itemsets $|I|$, and number of large itemsets $|L|$. We chose not to compare our methods with other existing item weight schemes for data streams, as they require explicit user defined weights, thus making any comparison inappropriate.

5.1 Precision

In this set of experiments, we used the following parameters $|T|20|I|4|D|500K$ and the number of unique item as 1000. Figure 2 shows the variation of Precision with the number of large $|L|$ items which were varied in the range from 5 to 25, in increments of 5. The top k% parameter was varied from 10% to 60%. The overall precision of WeightIncrementer vis-a-vis Recompute across all such experiments (i.e. over all possible combinations of L and k parameters) was 0.9. To measure the effect of the ϵ parameter in the Hoeffding bound we varied the reliability parameter δ. We used values of 0.00001, 0.001, 0.01, 0.1, and 0.2 for δ and tracked the precision for the top 20% of the items using a block size of 50K. Table 3 displays the results of precision based on the varying values. Overall there is a general trend where Precision increases with δ (i.e. decreasing ϵ).

Table 3. Variation of Precision with δ

Large Itemset	δ value				
	0.2	0.1	0.01	0.001	0.00001
5	0.95	0.95	0.95	0.95	0.90
10	0.81	0.81	0.80	0.80	0.78
15	0.91	0.91	0.80	0.80	0.80
20	0.94	0.94	0.94	0.94	0.89
25	0.97	0.89	0.88	0.88	0.88

5.2 Execution Time

We next compared the two methods on execution time. Both algorithms used the same data structure as described in the previous section. In this set of experiments, we used the following parameters $|T|20|I|4|L|20$ and the number of unique items as 1000. We varied the number of transactions ($|D|$) from 500K to 5M, with a block size of 100K. Figure 3 shows the execution time for the two methods for varying values of δ.

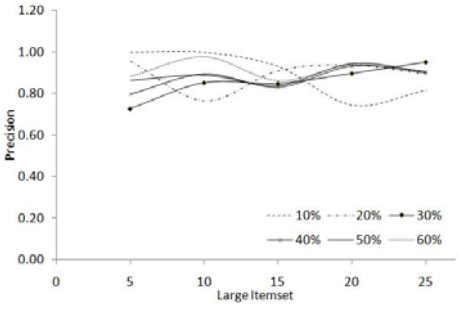

Fig. 2. Precision based on datasets L5 to L25

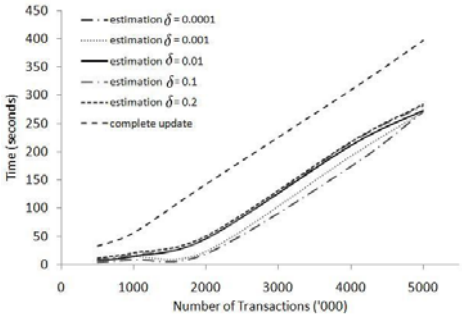

Fig. 3. Execution Time

In the experiments above, with a δ value of 0.00001 the percentage of updates to joint support (considering all possible pairs of items that interact with each other in a given block) was 27% whereas when the δ value was increased to 0.2 the percentage of updates increased to 37%. We also tracked the speedup achieved by WeightIncrementer with respect to Recompute. The speedup, defined as the ratio of the speed of WeightIncrementer to the speed of Recompute, ranged from a minimum of 1.4 to a maximum of 8.3.

5.3 Evaluating Drift

To evaluate the performance of WeightIncrementer in capturing drift we modified the IBM data generator to inject known drift patterns to track the sensitivity of WeightIncrementer to drift detection. We select x random points in time which:

- Introduce large itemsets: The main motivation is to simulate emerging patterns. Given the original set of large itemsets introduced, we hold off introducing a particular large itemset until we reach a predetermined point. Once we reach that point we introduce a large itemset into the stream with a probability of occurrence that increases incrementally over time.
- Remove large itemsets: The main motivation is to simulate disappearing patterns. When we reach a predetermined point, we remove a large itemset from the stream by incrementally decreasing its probability of occurrence over a period of time.

To evaluate whether our approach captures drift, we compare the results of the two methods, WeightIncrementer, with Recompute. We track the success rate of Recompute in detecting the large itemsets in the top 50% that were deliberately injected/removed and use this as a baseline to measure the performance of WeightIncrementer. We denote the ratio of the success rate of WeightIncrementer to Recompute as the *Hit Ratio*. If WeightIncrementer is able to capture all the items that were deliberately injected/removed relative to Recompute, then the hit ratio would be 1. In this experiment we modified the $|T|20|I|4|L|40|D|500K$

dataset; we use a block size of 50K and a δ value of 0.1. We injected various drift points into the dataset. Table 4 displays the results for the various drift points.

Table 4 shows that WeightIncrementer was able to detect both emerging as well as disappearing patterns with a minimum hit ratio of 74%, demonstrating that it can effectively adapt item weights and re-rank items with a high degree of precision in the presence of concept drift.

Table 4. Drift Analysis based on Hit Ratio

Drift Points	Large Items Injected	Large Items Removed	Hit Ratio
10	5	5	0.88
10	10	0	0.74
10	0	10	1.00
20	10	10	0.95
20	20	0	0.80
20	0	20	0.95

5.4 Real World Dataset: Accident

To evaluate our algorithm on a real dataset, we choose to evalute our algorithm on the accident dataset [9,10]. The accident dataset was provided by Karolien Geurts and contains (anonymized) traffic accident data. In total, 340,184 traffic accident records are included in the data set. In total, 572 different attribute values are represented in the dataset. On average, 45 attributes are filled out for each accident in the data set.

We ran experiments using multiple block size at 25K, 40K, and 60K. We then looked at the precision of the items based on the top 50%. We also compared the speedup of WeightIncrementer against Recompute. As Table 5 shows, the speedup ranged from 1.5 to 1.7. From Table 5 we note that the precision based on the top 50% of items was around the 87% mark. We also observed that the precision was not sensitive to changes in the δ value, remaining fairly constant across the δ range.

Table 5. Results based on the Accident Dataset

Block Size	WeightIncrementer			Recompute
	Precision (Top 50%)	Percentage of Updates	Time (s)	Time (s)
25K	0.86	0.39	776	1187
40K	0.87	0.36	736	1377
60K	0.90	0.38	575	955

Table 6. Hit Ratio vs δ (Block 60K)

Delta	0.2	0.1	0.01	0.00001
Precision	0.90	0.90	0.89	0.88

Overall we notice that the results from the accident dataset is consistent with that of synthetic data. The distance function works well to detect drift and estimate joint support. However we do acknowledge that under certain situations the run time performance of WeightIncrementer approaches that of Recompute; this happens when there are rapid changes, or fluctuations in the data stream. This will cause a substantial drift amongst all items in the dataset. When this

occurs, most of the joint support values need to be updated. However this only occurs in unusual circumstances with certain types of datasets, and we would not normally expect to see this kind of trend in a typical retail dataset displaying seasonal variations.

6 Conclusion

In this paper we proposed an algorithm to adaptively vary the weights of items in the presence of drift in a data stream. Item weights are assigned to items on the basis of the Valency model, and we formulated a novel scheme for detecting and adapting the weights to drift. Our approach reduces the number of updates required substantially and increases efficiency without compromising on the quality of the weights. We tested our drift detection mechanism on both synthetic and real datasets. Our evaluation criteria focussed on precision and efficiency in runtime. We were able to achieve Precision rates ranging from 86% to the 95% mark whilst achieving substantial speedup in runtime for both synthetic and real-world data.

References

1. Koh, Y.S., Pears, R., Yeap, W.: Valency based weighted association rule mining. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010. LNCS, vol. 6118, pp. 274–285. Springer, Heidelberg (2010)
2. Chang, J.H., Lee, W.S.: Finding recent frequent itemsets adaptively over online data streams. In: KDD 2003, pp. 487–492. ACM, New York (2003)
3. Chi, Y., Wang, H., Yu, P., Muntz, R.: Moment: maintaining closed frequent itemsets over a stream sliding window. In: ICDM 2004, pp. 59 –66 (November 2004)
4. Leung, C.S., Khan, Q.: Dstree: A tree structure for the mining of frequent sets from data streams. In: ICDM 2006, pp. 928 –932 (December 2006)
5. Ahmed, C.F., Tanbeer, S.K., Jeong, B.S.: Efficient mining of weighted frequent patterns over data streams. In: 10th IEEE International Conference on High Performance Computing and Communications, pp. 400–406 (2009)
6. Kim, Y., Kim, W., Kim, U.: Mining frequent itemsets with normalized weight in continuous data streams. JIPS 6(1), 79–90 (2010)
7. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58(301), 13–30 (1963)
8. IBM Almaden Research Center: Synthetic data generation code for associations and sequential patterns (1997), `http://www.almaden.ibm.com/softwarequest`
9. Goethals, B.: Fimi dataset repository, `http://fimi.cs.helsinki.fi/data/`
10. Geurts, K., Wets, G., Brijs, T., Vanhoof, K.: Profiling high frequency accident locations using association rules. Transportation Research Record: Journal of the Transportation Research Board 1840, 123–130 (2003)