Framework for Storing and Processing Relational Entities in Stream Mining*

Pawel Matuszyk and Myra Spiliopoulou

Otto-von-Guericke-University Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
{pawel.matuszyk,myra}@iti.cs.uni-magdeburg.de

Abstract. Relational stream mining involves learning a model on relational entities, which are enriched with information from further streams that reference them. To incorporate such information into the entities in an efficient incremental way, we propose a multi-threaded framework with a weighting function that prioritizes the entities delivered to the learner for learning and adaption to drift. We further propose a generator for drifting relational streams, and use it to show that our framework reaches substantial reduction of computation time.

1 Introduction

Stream mining algorithms are gaining in importance on many research and application fields. However, traditional stream mining makes the limiting assumption that a data instance is seen only once. In relational learning, the same entity (a customer, a patient, etc.) is observed many times, each time associated with additional data that should be taken into account during learning. This raises new challenges: (a) how to enrich a relational entity with all information known about it, although space and time are limited? (b) how to choose the entities that are most important for learning, since we can never consider all of them? In this study, we propose a multi-threaded framework that prepares and prioritizes relational entities for stream mining.

Relational learning is a mature field, but has only recently been perceived in the stream learning context [SS09, FCAM09, SS10, SS11, IDDG11], although learning on a drifting stream of relational entities is demanded in many applications. Consider, for example, churn prediction as experienced e.g. in the telecommunications industry or in banking. Companies attempt to predict the likelihood that a customer will continue or discontinue a contract. Obviously, this prediction is regularly done for each customer, each time considering the new activities of the customer *and* combining them to past activities. Making the right prediction for a customer x allows to use the predictor on customers, who *later* behave similarly to x. We generalize this example to the *Stream Learning Problem for Relational Entities* as follows:

* Part of this work was funded by the German Research Foundation project SP 572/11-1 IMPRINT: Incremental Mining for Perennial Objects.

> Let T be a stream of relational entities, which are in 1-to-n relation to instances from further streams $T_1, \ldots, T_m$. At timepoint t, learn/adapt a model $\zeta_T(t)$ on stream T, the *target stream*, exploiting the related instances from the other streams that have been seen thus far.

In the churn example, the customers constitute the target stream T; their service invocations, hotline requests, money transfers etc. constitute further streams of activities. These activities are exploited by $\zeta_T(t)$ to predict whether the label of customer x observed at t is CONTINUE or DISCONTINUE (i.e. quit the contract).

The stream learning problem for relational entities requires a tight and efficient coupling of stream preparation with stream mining. As mentioned at the beginning, space is limited and execution speed is an issue: how should a relational entity be expanded with all information seen on it thus far, so that it is made available to the learner? We propose a four-layer architecture with parallelizable components. Two layers of the architecture are responsible for (a) accommodating and (b) fetching relational entities from secondary storage, whenever activities referencing them are encountered; the other two layers are responsible for (c) queuing activities and (d) extracting information from the activities before discarding them. A core element of the framework is the weighting scheme responsible for the prioritization of relational entities that are expanded and delivered to the learner at each moment. The prioritization is motivated by the fact that the stream learner can never process all entities that have ever been seen, and by the fact that the model to be learned at a given moment should not consider entities that have not been referenced for a long while. Hence, our framework can be coupled with an arbitrary stream learner, whereby it handles the management and prioritization of past information for the relational stream.

In the next section we explain the context of this publication and related works. The new framework and its architecture are presented in section 3. Section 4 encompasses evaluation of the new framework and a description of a data generator, which has been developed for this purpose. In the last section we conclude and summarize the work and discuss a possible, further development.

2 Related Work

Learning on multiple interrelated streams is a very new problem, mostly called 'relational stream mining' or 'multi-relational stream mining'. In [SS09], Siddiqui et al. stress the fact that the relational entities re-appear, by calling them *perennial*, as opposed to the *ephemeral* transactions that reference these entities. We adopt this terminology, and use the terms 'perennial entity', 'perennial' and 'relational entity (of the target stream)' interchangeably hereafter; the terms 'instance' or 'ephemeral instance' we use for elements of the other streams.

First solutions to the Stream Learning Problem for Relational Entities, as we described it in the introduction, have been proposed by Siddiqui et al. in [SS09], where they studied stream clustering, and in [SS10], where they proposed a decision tree classifier for multi-relational streams. More recently, Ikonomovska et al. proposed following task: 'For each stream, determine the amount of facts that a relational incremental learner needs to observe at any point in time in order to be able to infer a correct model of the target function' [IDDG11, page 698]. Their goal is to minimize the information delivered to the learner, while our goal is to deliver only the entities relevant at some moment - but keep all information on these entities intact for the learner to exploit. Accordingly, we concentrate on lossless reconstruction of the relational entities.

However, the demand for recalling past entities does not always appear. For example, the method of Fumarola et al. on association rules discovery over multi-relational streams [FCAM09] only deals with the entities that are inside the window. The demand of recovering past entities emerges through the need to adapt the model on instances that reference entities seen earlier. For example, consider a decision tree based stream classifier that distinguishes between low-risk and high-risk customers. The customers constitute the *target stream*, the entities of which should be associated to the fastest transaction stream(s) of the customers[1] Whenever a new transaction for this customer appears, the customer's entity must be reconstructed and and its label must be predicted again, given the entire information known on this customer - the classifier may then predict for this customer a different label than it has predicted earlier. Multi-stream classification algorithms [IDDG11, SS10] either operate on the most recent data (at the cost of information loss, especially for entities that re-appear at very slow pace) or require an entity reconstruction algorithm.

Incremental entity reconstruction algorithms have been proposed in [SS09, SS11]. The former is an incremental version of a join-like operation called *propositionalization* [KRv$^+$03], which essentially extends the schema by turning values into columns/features.

The incremental propositionalization method of [SS09] ensures that the schema does not grow in an unbounded way by fixing the size of the feature space and clustering the values of similar entities into the fixed features. The CRMPES method of [SS11] rather identifies values that predict the label with classification rule mining, and uses these values as features. While this method requires much less space than the former, it is by nature sensitive to drift, and may lose important information on rarely re-appearing entities. In this study, we focus on entity reconstruction for multi-stream classification *without information loss.* We therefore build upon the former method, by establishing a database architecture that prepares the interrelated streams for adaptive learning and ranks entities on their expected importance for adaption.

In studies on database querying over streams, we find heuristics that prioritize the entities to compute approximate joins [DGR03, XYC05]. These heuristics predict which of the entities seen thus far will be needed for the join computation and which will not: the former are retained in the cache, the latter are kept in secondary storage. Their objective is to minimize the number of accesses to the secondary storage. Das et al. propose several heuristics that rank entities on the likelihood of being needed for the join computation [DGR03]: PROB assigns

[1] There may be more than one transaction streams: bank account transactions, transactions on loans, transactions with a credit card etc.

higher rank to those entities of the one stream that are frequently referenced from the other stream. LIFE [DGR03]computes for each entity the time it will stay inside the window[2]; LIFE chooses entities whose expected remaining 'lifetime' in the window is longer. The 'Heuristic of Estimated Expected Benefit' HEEB [XYC05] exploits past knowledge to compute the likelihood that an entity being observed now will be needed in near future. HEEB comes closest to the demands of our stream classification scenario: if an entity x of the target stream has been often referenced from the other streams in the past, it is likely that many instances refering to it will arrive in the future.

One pitfall of the HEEB weighting scheme is that it does not account for drift. In this study, we abstract drift as the *ageing* of some part of the model, until this part becomes completely obsolete. The anytime stream learner ClusTree uses an exponentially non-increasing function $\omega(\Delta t) = \beta^{-\lambda \Delta t}$ to assign an age-based weight to micro-clusters [KABS09]. One-stream classifiers based on VFDT [DH00] discard subtrees that have not received instances for a while [HSD01, GRM03], while the multi-stream classifier TrIP computes the age of a subtree on the basis of the age of the entities in it, since entities may re-appear [SS10]. We similarly take account of an entity's age with help of an exponentially non-increasing decay function, and re-juvenate entities as they re-appear.

3 Framework

Conventional stream mining algorithms learn a model by sliding a window over the arriving data instances. For the learning problem introduced in Section 1, the model is learned on the *permanent* relational entities and is updated as new instances referencing these entities arrive. Hence, instead of sliding a window over the arriving instances, we need to fetch the referenced entities for learning and adaption. We use a *cache* to accommodate the referenced entities: if a relational entity is referenced by an arrived instance, then this entity is fetched from the database to the cache, and is extended with the information of the arrived instance. This aggregation of new information is implemented as an incremental propositionalization mechanism [KRv$^+$03, SS09], which is part of our Four-Layer Architecture for relational entity preparation. This architecture and the functionalities of its components are depicted in Figure 1 and described below. In the following, we use the term 'perennial' for the relational entities of the target stream, to stress their permanent nature, and 'ephemeral' for the arriving instances, to stress that they are seen and forgotten, as is the case for conventional stream data [GMM$^+$03].

3.1 Four-Layer-Architecture

The Ephemeral layer consists of streams of ephemeral instances. It is responsible for supplying the framework with data. This layer can be physically distributed over many computers in a network. In the churn prediction scenario,

[2] As in stream classification, stream join is computed on the content of a window sliding over the streams.

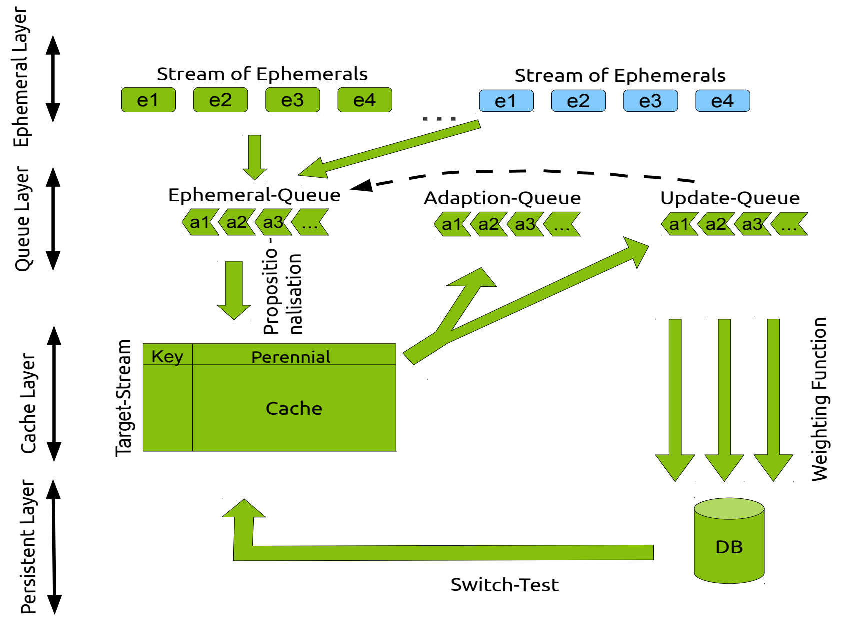


Fig. 1. Four-Layer-Architecture: Ephemerals (new data instances) are stored in the Ephemeral-Queue and propositionalised into perennials (entities). A subset of perennials is stored in the cache for faster access. It also serves as adaption mechanism. The Database as persistent storage is located in the persistent layer. The Queue layer is an intermediate layer that is responsible for paralellyzation and for decoupling the work of single components from each other.

for instance, the ephemerals represent money transfers, transactions or hotline requests recorded at different servers.

The Queue layer consists of three queues. The ephemerals delivered by the previous layer are stored in the Ephemeral-Queue, from where they are extracted by further threads of the framework. The extracted ephemerals are then propositionalised into the corresponding perennials using the mechanism of [SS09]. A selection of perennials is stored in the cache and used for model learning; all other perennials remain stored in the database, and fetched through an update action (carried out by a further thread group). The Ephemeral-Queue guarantees the separation of the mining algorithm from the stream management process. We have further queues, used for *actions for adaption and for updating*, which are performed by other threads:

– *Actions for adaption* encompass learning and forgetting. They process a perennial that should be presented to the mining algorithm and thus incorporated into a mining model, or one that should be forgotten by the algorithm and thus, excluded from the model. Such actions are launched every time a perennial enters or leaves the cache.

– *Update actions* serve the purpose of updating perennials stored in a database (cf. persistent layer). Therefore, they consist of a reference to a perennial and of an ephemeral that has to be aggregated into the referenced perennial.

The Queue Layer is responsible for the parallelization of processes within the framework. Each queue is served by its own thread group. The distribution of the tasks among many thread groups leads to the reduction of the computation time on multi-core processors.

The Cache layer serves following purposes. First, it speeds up the access to the perennials used for learning; these are selected with help of the weighting function presented thereafter. By delivering perennials to the learning algorithm, the cache layer is responsible for the process of model adaption to concept drift.

Second, the cache implements the operation of sliding a window over arriving stream instances. Since we study a learning problem upon relational entities, we cannot use a conventional window. Rather, as stream instances, i.e. ephemerals, arrive, the entities referenced by them (in the churn prediction example: already seen customers, or new ones) are presented to the mining algorithm. To prevent cache overflow and stick to the most important entities for learning, we define a weighting function that decides which perennials should enter the cache and which ones should be moved to the database.

Our framework allows to define an arbitrary weighting function. For example, in the churn prediction scenario, the human expert may decide that the most important customers for learning are not those observed most recently but those most active during the whole observation period. In this study, we propose following exponential function to compute the weight of a perennial p:

$$f_W(p) = (1 - \beta) \cdot e^{-p_a} + \beta \cdot (-e^{-p_s} + 1) \tag{1}$$

where p_a is the age of p, i.e. the elapsed time since the perennial was referenced for the last time, while p_s is the *support* of p, defined as the number of ephemerals referencing p thus far. The parameter β expresses the preferences of the analyst (or the decision maker) regarding relative importance of age versus support.

The Persistent layer is responsible for the management of perennials. The relational entities seen thus far may not be deleted, but it is not possible to keep all of them in main memory. Therefore, we use a database as a persistent storage. When a perennial is referenced by an ephemeral that has just arrived in the stream, then an "update action" is created and stored in the Update-Queue. From there the update actions are extracted by a thread group that is responsible for aggregating new information to the perennials, using the incremental propositionalization of [SS09]. After such an update, the weight of a perennial may change. If the new weight is larger than the lowest weight in the cache, then the perennial with the highest weight in the database replaces the perennial with the lowest weight in the cache. To avoid too frequent switches between database and cache, we perform a 'switch-test', where we check whether the difference between the two weights is significant.

Example 1. Consider the aforementioned churn prediction scenario, where the label of a customer has to be predicted. The challenge is here to combine all data that comes from multiple streams and relations belonging to this customer efficiently and learn upon those data in real time.

The process starts at the Ephemeral Layer, i.e. as a new transaction by the given customer arrives. Subsequently, this ephemeral object is stored in the Queue Layer, where it awaits the propositionalization. Hence, the corresponding perennial (the customer) has to be retrieved from the cache or from the database (Persistent Layer) and updated using the new transaction in course of the propositionalization. After that, the customer object can be stored back into the cache or into the database. The data mining model is kept consistent with the data in the cache, which plays the role of sliding window : every change of the data in the cache has to be followed by an update of the model. Therefore, an update action is created and stored in the Queue Layer. From there it is retrieved and performed by a different, parallel thread. Then, if the weight of the customer object (cf. Eq. 1) is high, the update action concludes by storing the object into the cache and incorporating it into the model. Finally, the object's label is derived.

3.2 Data Flow within the Framework

After explaining the internal structure and architecture of our framework, we now focus on the data flow and processes within the framework, as depicted in Figure 2. The work of the framework starts with the arrival of a new ephemeral instance, which is subsequently stored in the Ephemeral-Queue. From there it is extracted by another thread that carries out the propositionalisation. First, the location of the referenced perennial has to be determined. The thread checks whether the perennial is in the cache. If this is the case, the propositionalisation is performed immediately, the weight of the perennial is updated, and the perennial is saved back into the cache. If the referenced perennial is not in the cache, then it must be in the database. In this case, an update action has to be created and stored in the Update-Queue. Another thread group extracts the update actions from the queue and picks the referenced perennial(s) from the database.

Following case may occur: the perennial has been already moved into the cache, while the update action was waiting in the Update-Queue. Therefore, a further check is necessary. Thereafter, the perennial is updated using propositionalisation, and a switch-test is performed to check whether the perennial should be moved to the cache.

Cache overflow prevention: If the perennial has high weight (Eq. 1) and must be kept in the cache, we check whether the cache is overfilled. In such a case, we move the perennial with the lowest weight back to the database. Additionally, we launch a forgetting action (stored in the Adaption-Queue), so that this perennial is no more considered in the current model.

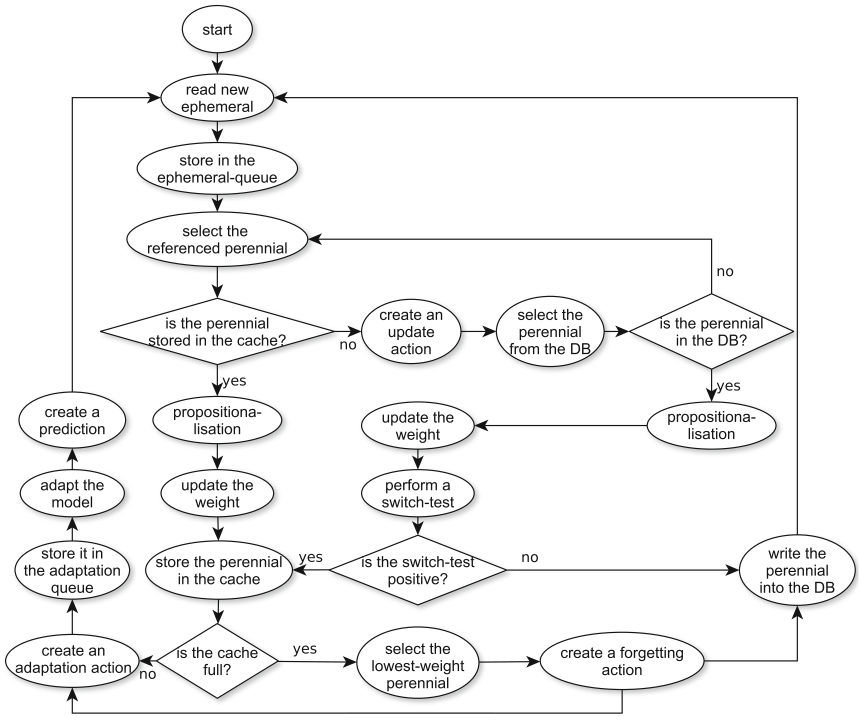

start
read new ephemeral
store in the ephemeral-queue
select the referenced perennial
is the perennial stored in the cache?
no
create an update action
select the perennial from the DB
is the perennial in the DB?
no
yes
yes
propositionalisation
update the weight
perform a switch-test
is the switch-test positive?
yes
no
write the perennial into the DB
propositionalisation
update the weight
store the perennial in the cache
is the cache full?
yes
no
select the lowest-weight perennial
create a forgetting action
create an adaptation action
store it in the adaptation queue
adapt the model
create a prediction

Fig. 2. Data flow within the architecture; processes can be parallelized (e.g. propositionalisation of perennials and database accesses can be performed simultaneously)

If a new perennial is saved in the cache, then also an adaption action is put into the Adaption-Queue. The last group of threads executes the actions in the Adaption-Queue and predicts the perennials' labels. This is repeated as long as there are new unprocessed ephemerals, whereby the threads run in parallel.

4 Experiments

To evaluate the performance of our framework in a controlled way, we developed a data generator that creates streams that change their speed and exhibit drift. The generator is presented first. We then present the experiments on execution speed and result quality, for which we coupled our framework with the relational stream classifier TrIP [SS10]. We used an Intel i5 with 2.4 GHz (2 cores and 4 parallel threads) and 4 GB RAM.

4.1 Data Generator

Our data generator takes as input a number of perennials and generates a stream of ephemerals which reference them. Although the streams are by definition infinite, an obviously finite number of ephemerals is specified in order to ensure that the generator terminates its work and results can be seen.

The number of classes of perennials and the number of feature space dimensions are also input parameters. The classes follow the Gaussian distribution. Furthermore, the generator simulates concept drift by shifting the μ parameter of the class distribution by a given value. We introduce several parameters to govern concept drift, and use a simple notation, which we call 'drift string', to set them. The parameters are: the time point s when the concept drift starts; the time point e when the concept drift ends; the velocity of the drift v (number of units a class center is shifted at each time point); the class c affected by the drift, and the attribute a affected by the drift. Hence, the drift string has the form: $s < s > e < e > v < v > c < c > a < a >$.

4.2 Reducing Computation Time through Multi-threading

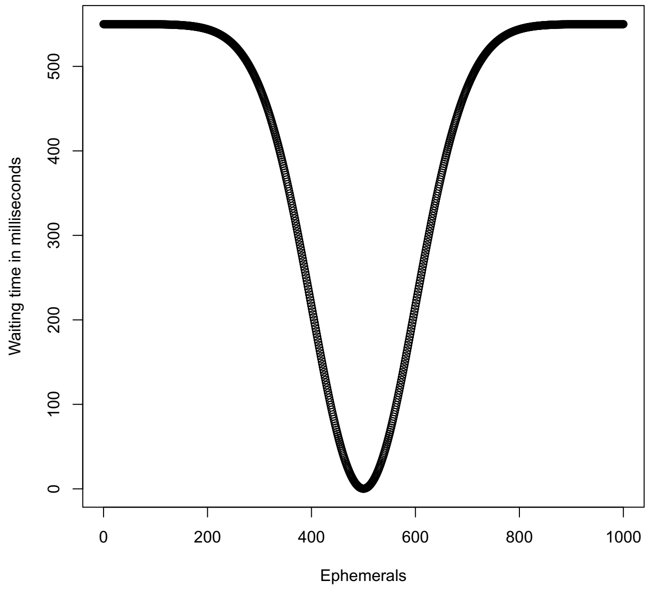
To measure how our four-layer-architecture (4LA) reduces computation time, we implemented a baseline one-thread-architecture (1TA) with the same functionality as 4LA.

The first parameter affecting the computation time of the framework is the cache size. When the cache is small, then many perennials have to be moved to the database, thus increasing computation time. The other important parameter is the number of perennials: if they are only few, then it is more likely that a new ephemeral will reference a perennial that is already stored in the cache.

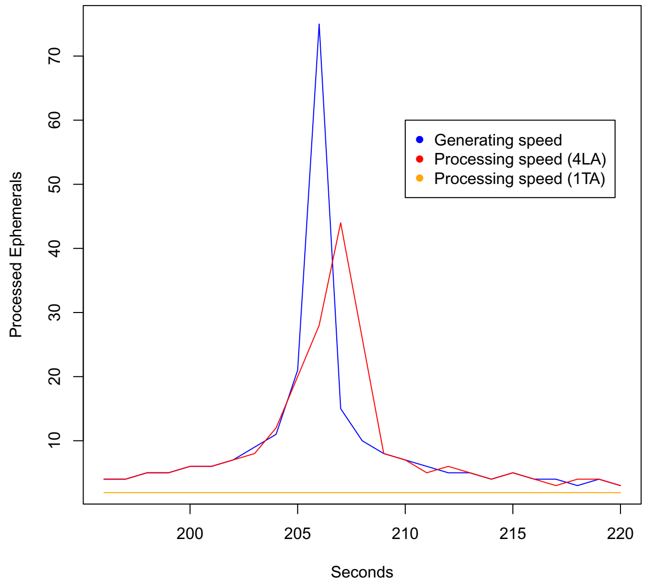
In Table 1, we show the impact of the parameters on the computation time of 4LA and 1TA. Two cases have to be distinguished. In the first case, the cache is so large that it accommodates all perennials (black numbers, below the diagonal). In the second case, the number of perennials exceeds cache size (blue numbers, above the diagonal), so some perennials had to be moved to the database, increasing computation time. Red numbers denote the best relative

Table 1. Computation time of 4LA and 1TA in milliseconds: lower values are better, best improvements marked in red. When the cache is too small (above the diagonal), computation time includes data swapping.

Number of Perennials		2	10	100	1000	10 000
Cache Size (as number of perennials)	3 (1TA)	15803	525368	689051	666646	543933
	3 (4LA)	1253	26479	258546	605514	499776
	10 (1TA)	24298	75260	616953	676923	514007
	10 (4LA)	1510	9148	58591	569834	460868
	100 (1TA)	24537	25193	34744	579670	494098
	100 (4LA)	1340	1137	1602	249130	439244
	1000 (1TA)	18516	33919	29059	32481	382275
	1000 (4LA)	1080	972	1135	1124	308691
	10 000 (1TA)	20921	25042	26350	26307	38591
	10 000 (4LA)	887	888	1086	1103	1342



(a) Simulation of a peak in the stream speed of ephemerals using Gaussian distribution; the curve represents waiting time in milliseconds. Therefore, the peak points downwards.



(b) Generation and processing speed of 4LA (red) and 1TA (blue); the red curve has a lower peak and exhibits a time lag over the peak in the stream speed; the peak of the blue curve is at the same time point as the stream peak.

Fig. 3. Effect of smoothing a peak in stream speed

improvement of 4LA over 1TA. In the first case (below the diagonal), the improvement on computation time reached 97.13%. In the second, more realistic case, the relative improvement reached 94.96%, i.e. 4LA needed only 5.04% of 1TA's computation time.

4.3 Dealing with Speed-Up of the Stream

A further advantage of our 4LA framework is that it smooths temporal speedups of the ephemeral streams. We have simulated such a speedup by reducing the elapsed time between the arrival of two ephemerals, and compared the computation time of 4LA to 1TA. When the stream speed becomes higher than the processing speed of the miner, then further actions that cannot be carried out immediately, are cumulated in the queues. Following experiment shows that our new architecture can cope with a temporal speed-up of the stream.

A peak in the speed of the stream of ephemerals was simulated using normal distribution (cf. Figure 3a) over the waiting time between generating two ephemerals. Thus, the peak of the distribution points to the bottom of the page. Thereafter, the processing speed of the 4LA and 1TA were measured. The stream speed is represented by the blue curve in Figure 3b. It is apparent that the processing speed of the 4LA (red curve) is not as high as the stream speed. An advantage of the new framework shows at this stage - the framework does not collapse under the high load of ephemerals, but it rather starts to cumulate the ephemerals in queues. When the speed of the stream becomes lower again, the framework processes the ephemerals from the queues, what is apparent from the shift of the red curve to the right of the blue curve. For the contrast, the maximal processing speed of the 1TA has been depicted by the orange line in Figure 3b.

5 Conclusions

We described a novel framework for storing and processing relational entities in stream mining, based on a four-layer-architecture. Due to a partial parallelization of processes within the framework an essential reduction of computation time was possible. The computation time was reduced by up to 97.13%. Thanks to the usage of the queue layer the framework gained the ability to cope with streams witch changing speed - an issue of particular relevance for real-world scenarios. Furthermore, we used a new weighting function that prioritizes the entities to be used for learning, giving preference to those most recently referenced.

Our framework is scalable and appropriate for multi-core processors. As future work, we want to extend it so that it runs on many computers in a network. This will not only reduce computation time, but it also will allow us also to delegate part of the stream classification itself to a network of computers. While distributed data mining has been widely investigated for static data, stream mining and the incremental propositionalization step in the preprocessing phase incur additional coordination overhead that has yet to be investigated.

References

[DGR03] Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: SIGMOD Conference, pp. 40–51. ACM (2003)

[DH00] Domingos, P., Hulten, G.: Mining high-speed data streams. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 71–80. ACM (2000)

[FCAM09] Fumarola, F., Ciampi, A., Appice, A., Malerba, D.: A sliding window algorithm for relational frequent patterns mining from data streams. In: Gama, J., Costa, V.S., Jorge, A.M., Brazdil, P.B. (eds.) DS 2009. LNCS, vol. 5808, pp. 385–392. Springer, Heidelberg (2009)

[GMM$^+$03] Guha, S., Meyerson, A., Mishra, N., Motwani, R., O'Callaghan, L.: Clustering data streams: Theory and practice. IEEE Transactions on Knowledge and Data Engineering, 515–528 (2003)

[GRM03] Gama, J., Rocha, R., Medas, P.: Accurate decision trees for mining high-speed data streams. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 523–528. ACM (2003)

[HSD01] Hulten, G., Spencer, L., Domingos, P.: Mining time-changing data streams. In: Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 97–106. ACM Press (2001)

[IDDG11] Ikonomovska, E., Driessens, K., Dzeroski, S., Gama, J.: Adaptive windowing for online learning from multiple inter-related data streams. In: Proc. of Int. Workshop on Learning and Data Mining for Robots (LEMIR 2011) at the 11th IEEE Int. Conf. on Data Mining Workshops Volume, Vancouver, Canada, pp. 697–704 (December 2011)

[KABS09] Kranen, P., Assent, I., Baldauf, C., Seidl, T.: Self-adaptive anytime stream clustering. In: Ninth IEEE International Conference on Data Mining, ICDM 2009, pp. 249–258. IEEE (2009)

[KRv$^+$03] Krogel, M.-A., Rawles, S., Železný, F., Flach, P.A., Lavrač, N., Wrobel, S.: Comparative Evaluation of Approaches to Propositionalization. In: Horváth, T., Yamamoto, A. (eds.) ILP 2003. LNCS (LNAI), vol. 2835, pp. 197–214. Springer, Heidelberg (2003)

[SS09] Siddiqui, Z.F., Spiliopoulou, M.: Combining multiple interrelated streams for incremental clustering. In: Winslett, M. (ed.) SSDBM 2009. LNCS, vol. 5566, pp. 535–552. Springer, Heidelberg (2009)

[SS10] Siddiqui, Z.F., Spiliopoulou, M.: Tree induction over perennial objects. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 640–657. Springer, Heidelberg (2010)

[SS11] Siddiqui, Z.F., Spiliopoulou, M.: Classification rule mining for a stream of perennial objects. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) RuleML 2011 - Europe. LNCS, vol. 6826, pp. 281–296. Springer, Heidelberg (2011)

[XYC05] Xie, J., Yang, J., Chen, Y.: On joining and caching stochastic streams. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 359–370. ACM, New York (2005)