Distributed Adaptive Rules with Apache Flink for Data Stream Mining – Preliminary Work

Lukasz Korycki, Virginia Commonwealth University, USA

***Abstract*—Data stream mining is among the most important contemporary machine learning issues. The potentially infinite sources give us many opportunities, but also pose additional challenges. To properly handle streaming data we need to adjust our solutions, so they can work with dynamic data and under strict constraints. Adaptive rule-based classifiers are one of the algorithms that can be utilized in this domain. They can work in the online setting, incorporating incoming instances and providing a human-readable form of their models. On the other hand, since they are often characterized by a relatively high computational complexity, parallelization of such methods may be necessary if we want to use them in the streaming environment. Distributed systems are one of the possible solutions that have been already successfully used in this task. In this work, we present an implementation of adaptive rules distributed using Apache Flink framework. The obtained results indicate that the horizontal parallelization is able to provide significant speed-up without impeding accuracy**

***Keywords*-IEEE, IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

DATA streams are infinite sources of knowledge and challenges. Sensors, web engines and monitoring systems generate countless data points every day and every second. On the one hand, we have unprecedented opportunities to analyze different phenomenas in a continuous and more adequate way – not only from scare and incomplete snapshots, as it is in the case of off-line batch processing. On the other hand, we face several problems coming with the nature of rapid, dynamic and massive sources, that have to be properly addressed.

In machine learning domain we have to take into consideration two substantial observations. Firstly, distributions of data streams are very often non-stationary. Concepts are evolving as time passes, unseen or rare data samples appears. We refer it as a concept drift. Secondly, streams are continuous and potentially infinite. There is no possibility of gathering the whole data and using it for a single batch-mode training. Processing of data streams is restricted to limited rationality, which means that quality and complexity of decisions made by streaming algorithms highly depend on finite computational resources. The algorithms are able to operate only on a partial and temporary knowledge. As a result, previously created models should be regularly adjusted to a dynamically changing environment, to evolving data streams.

The problem of keeping models up-to-date while working under strict constraints is the main concern of adaptive learning methods. In general, we can distinguish two groups of such algorithms: batch-based and on-line. The former may occur in a form of retraining or incremental updates. The latter is a special case of the incremental cases, with the restriction that each incoming instance has to be processed on-by-one, immediately and only once. It seems to be the most suitable to streaming data, since such methods tend to provide minimal latency and low memory occupations. In fact, low latency is one of the most important requirements while designing algorithms for data streams. Distributed streaming frameworks are possible solution to this problem

In this work, we introduce an implementation of adaptive rules for data stream mining. We use delineate basic concepts, as well as we present related works. In Section III a sequential algorithm along with proposed implementations are described. Section IV is focused on conducted experiments, while Section V concludes the whole work.

II. Distributed adaptive rules

While there are plenty of machine learning algorithms dedicated for data stream mining, like Hoeffding Trees [x] and Oza's ensembles [x], only few of them provide human-readable forms of models. This may be a significant problem in such domains as medicine or economy, in which automatic decisions should be properly explained.

A. Rule-based models

Rule-based algorithms and decision tress are the most suitable in such cases. While the latter are known to be very efficient in terms of computing time, the former is considered the most flexible due to modularity of decision rules.

One of the most popular classifiers based on rules is Adaptive Model Rules (AMR) proposed in [x]. It is an online model that incorporate knowledge from new instances by expanding its rules based on the widely used Hoeffding bound criterion [x]:

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

where R is a value range, n_l is a number of examples seen on a node l and δ is a confidence parameter. If a difference between two best attributes on the node is greater than ϵ, then there is a $1-\delta$ confidence that the better attribute introduces superior information gain and it should be used to create a split.

While the flexibility and transparency of rule-based systems are their advantages, a high computational complexity is considered as one of the main drawbacks. Fortunately, since the models are modular, there is a lot of possibilities to improve their performance using parallelization techniques.

B. Distributed computing

One of the possible approaches is distribution of computations across several machines. The domain of distributed computing is currently occupied mainly by algorithms implemented in one of the three frameworks: Apache Spark, Flink or Storm. While all of them aim for increasing throughput of algorithms and decreasing latency in general, they are not suitable for all specific settings that may be distinguished in data stream mining.

- **Spark** – it is highly oriented on batch processing and treats online processing as a special case of the former. The main idea behind this framework is to perform massive parallel computations on huge batches of data, so the overhead of scheduling tasks becomes minor. Originally, the main use case was to simply ensure computational feasibility for large data sets without a strong focus on the rate of incoming data. Currently, Spark has been evolving into a streaming framework with low latency of processing. Structured Streaming and Continuous Processing are two main features for such cases. The former is based on micro-batches of high-performing Data Frames, which decreases latency. However, since computation are still organized in batches, it may be prohibitive, for example, in cases with limited amount of labeled data Continuous processing enables strictly streaming processing, but since it is still in experimental mode, a lot of operators are not available yet. It is probably the main reason, why modeling online streaming processing is not very convenient in Spark at the moment.

- **Flink** – this framework can be somehow considered as opposite to Spark – it is build around the idea of online processing, while batches are special cases of it. All operators in Flink are considered *long-running*. It means that their instances (including parallel ones) are kept on tasks partitions constantly, so data literally *flows* through them in a continuous stream from sources. Such approach naturally creates an environment for statefull computations in which parts of models are maintained in operator instances. It is especially important in the case of adaptive distributed machine learning algorithms.

- **Storm** – it can be treated as a predecessor of Flink. Dedicated for online streaming processing, Storm organizes processing using spouts (sources) and bolts (operators), which handle streaming data. Currently, it seems that Storm is being gradually replaced with Spark Streaming and Flink. In addition, a low rate of contributions to its repository is an another factor that should be taken into consideration while choosing a framework.

C. Distributed decision rules

Due to the presented characteristics, it is not surprising that most of the streaming machine learning algorithms, even if few, have been implemented in Storm or Flink, mainly in SAMOA [x]. The already mentioned AMR algorithm is one of them [x].

There two main approaches to implementing this algorithm in a distributed manner. The first one is **vertical parallelization**, in which decision rules are distributed across several partitions, so all necessary updates are performed separately for each rule. Since there is still no efficient way to distribute prediction process without creating a synchronization barrier for each instance, in vertical architecture all conditions are maintained in a single rules aggregator, so only statistics for splits are distributed. One of the most important properties of this method, that should be emphasized, is that incoming instances does not wait for updates that arrive asynchronously.

The second approach is **horizontal parallelization**. In this architecture the algorithm maintains several independent rule models. It allows for processing several instances at the same time. Such approach definitely increases a throughput, however, since for each model a kind of subsampling is performed (depending on its availability), it also introduces a potential problem of undersampling. There is a third option (hybrid) that combines both approaches by introducing a common default rule, which propagates new rules to all models.

The AMR algorithm has already been implemented in Storm [x]. There is also an implementation of the Hoeffding Tree in Flink. However, to the best of our knowledge, AMR has not been integrated with the latter framework.

III. Proposed implementations

The proposed implementations of distributed adaptive rules in Flink are based on the original AMR and the two architectures mentioned in the previous section.

A. Sequential

The sequential version (`SAMR`) of the algorithm's learning is almost the same as the original one [x]. For each incoming instance statistics are updated for all rules that covers it. If a sufficient number N_{min} of instances was registered for a rule, the algorithm tries to expand the rule by looking for an attribute for which an effective split of classes can be established. For this purpose an entropy is calculated. If the value for the best possible split is significantly greater (Hoeffding bound) than a current rule's entropy, a new condition based on the split value is added to the rule and statistics are released.

If no rule covers and incoming instance, statistics for the default rule are updated and under the same circumstances, an expansion is attempted. If the rule received a condition, it is added to the set of maintained rules and a new default rule is created.

One of the crucial parts of the learning process is an approach to looking for the best split. The naive method that keeps the whole information about instances and analyzes possible bins sequentially, while calculating entropies, consumes a lot of time and memory. Some improvements are possible, like for example, utilizing a binary tree to optimize the searching [x]. However, to the best of our knowledge, the best solution, which can be also easily integrated with distributed architectures, is estimating the required class counters, using

Gaussian distributions [x]. In such approach only a mean, standard deviation, counters, minimal and maximal attributes values have to be maintained. All of them can be calculated incrementally in an efficient way [x].

In the presented implementations, due to time limitations, we do not apply anomaly test nor removal of underperforming rules. This is the only different compared with the original algorithm, which, however, may be significant in the case of drifting data streams.

B. Vertical parallelization

In our vertical implementation (`VAMR`) we used the following long-running operators, connections and data. We present only the most important elements. For more details, please, check the source code attached.

- `Rules Aggregator` – it is a *ProcessFunction* that receives instances as a main input. It performs all the operations (checking coverage, classification, maintaining the default rule) except for updates of rules statistics. Requests for the updates and initialization of new rules' statistics are sent, using a *SideOutput*. The operator is placed in an iteration of *IterativeStream* which allows for receiving a feedback from statistics partitions.

- `Partial Rules Processor` – this parallel operator is responsible for updating statistics for given rules and expanding them. If a new condition is created, it is sent to the `Rules Aggregator`. The number of these operators depends on a given level of parallelism (partitions).

- `Modulo Partitioner` – it is responsible for distributing rule statistics between partitions. We simply use a modulo formula based on an rule's id.

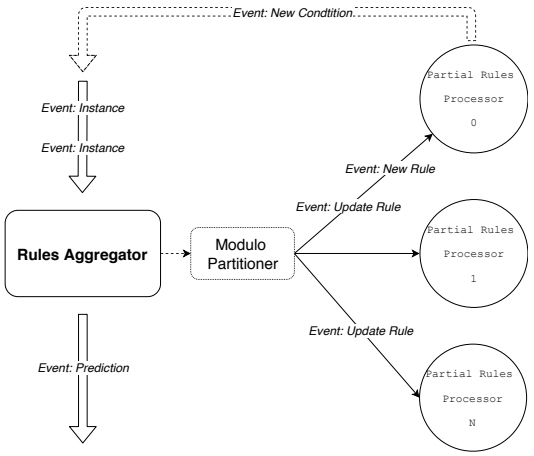

Fig. 1: The vertical parallelization.

Since different operators may receive different types od data, we needed to unify the data points into a common type, which is `Event`.

C. Horizontal parallelization

To implement the horizontal parallelization (`HAMR`) we simply created a pipeline connecting the input stream with `Single Predictor`, which is a *MapFunction* containing a single sequential model of ARM, and increased the number of partitions for it. As a result, we achieved our goal by utilizing basic Flink's capabilities.

IV. Experiments

In the experimental section, we evaluated our implementations of the distributed AMR, using different data streams and levels of parallelism. We compared them with the sequential version of the algorithm.

A. Data streams

During the experiments, we used 10 different data streams (7 real and 3 synthetic). Most of them are supposed to be non-stationary. They are summarized in the Tab. I.

Tab I: Summary of the used data streams.

Name	Instances	Attributes	Classes	Type
ActivityRaw	1 048 570	3	6	Real
Connect4	67 557	42	3	Real
Covertype	581 012	54	7	Real
Electricity	45 312	8	2	Real
Hyperplane	1 000 000	15	5	Synthetic
Lymph	1 000 000	18	5	Synthetic
Olympic	271 116	7	3	Real
Poker	829 201	10	10	Real
Weather	18 159	8	2	Real
Wine	1 000 000	13	3	Synthetic

B. Set-up

The experiments were conducted using 1, 2, 4 or 8 partitions on a local machine. We adjusted the minimum number N_{min} of instances for expanding a rule, depending on a size of a stream (100 or 1000). The confidence parameter δ was equal to 0.05. For each setting we measured accuracy over a whole data stream, as well as time spent on computations.

C. Results

SAMR. In general, the AMR algorithm exhibited a mediocre performance (Tab. II) with an exception for the Olympic stream (0.85) and Activity (0.32). The computing time (Tab. V) varied significantly (standard deviation equal to about 42), which was caused by both data stream size and complexity of generated models.

VAMR. One can see that the vertical parallelization leads to some drops in accuracy compared with SAMR, usually up to 0.1. The average drop was oscillating around 0.06 with a relatively high standard deviation (about 0.10), however, it was caused mainly by two data streams: Electricity (drop between 0.3-0.4) and Lymph (improvements about 0.07). At the same time, VAMR was able to provide a more less stable speedup

Tab II: Accuracy of SAMR and **relative accuracy** when running VAMR with different numbers of partitions P.

Stream	SAMR	P=1	P=2	P=4	P=8
Electricity	0.7187	0.7154	0.6943	0.7180	0.6274
Covertype	0.6220	0.9590	0.8640	0.8524	0.9214
Weather	0.7008	0.9288	0.9793	0.9654	0.9793
Activity	0.3213	1.0459	1.0841	0.9470	1.1316
Poker	0.5548	0.8940	0.8694	0.9234	0.9355
Olympic	0.8527	0.9699	1.0007	0.9966	0.9924
Hyperplane	0.5822	1.0199	1.0096	1.0137	1.0228
Wine	0.6033	0.9106	0.9151	0.9036	0.9226
Connect4	0.5618	0.9606	0.9310	1.0028	0.9976
Lymph	0.4900	1.0795	1.0637	1.0717	1.0853
Avg	**0.6007**	**0.9484**	**0.9411**	**0.9394**	**0.9616**
Std	0.1423	0.1009	0.1145	0.0995	0.1358

Tab III: Computing time [s] for SAMR and **speed-up** when running VAMR with different numbers of partitions P.

Stream	SAMR	P=1	P=2	P=4	P=8
Electricity	7.093	2.9070	3.9449	4.1946	4.0741
Covertype	24.453	1.0529	1.7989	2.3445	1.1923
Weather	1.854	1.6052	1.2493	1.0779	1.5725
Activity	57.101	2.3569	2.0596	2.6571	2.8897
Poker	29.076	2.7821	2.5237	0.9990	0.7239
Olympic	154.433	8.0104	11.9957	15.6103	18.7760
Hyperplane	56.423	2.5391	3.3036	5.2219	4.9093
Wine	26.612	1.0858	1.1218	0.9124	1.0133
Connect4	7.37	0.7683	0.9750	0.9833	1.0824
Lymph	37.565	1.5611	1.6192	0.8845	1.4584
Avg	**40.198**	**2.4669**	**3.0592**	**3.4886**	**3.7692**
Std	42.1967	2.0922	3.2834	4.5233	5.4596

between 3 and 3.5 for more than one partitions. However, it was elevated mainly by results for the Olympic stream (about 8-18), which also significantly increased variability of results. In most cases, the algorithm was not scaling up with the number of partitions. It was expected, since independent updates do not influence prediction in terms of the computing time. On the other hand, increasing the level of parallelization should positively influence accuracy (faster updates) – it can be barely seen for P=8. It is possible that evaluation using a higher number of cores would provide more information about this relation. The speed-up observed for P=1 was a result of splitting prediction and updates into separate channels.

HAMR. Much more encouraging results have been obtained for the horizontal parallelization. One can clearly see that HAMR was able to provide significant speed-ups (increasing more less directly proportionally to number of partitions for P=2 and P=4) while keeping accuracy on a similar level as SAMR. The average loss was about 0.01-0.02 with standard deviation between 0.02-0.08 that was increasing with the number of partitions, being elevated mainly by results for one data stream: Activity (1.05-1.18).

The algorithm was also able to slightly improve results in 3 other cases. It is an important observation, since it indicates that the ensemble of completely independent classifiers (regarding both training and prediction) may be sufficient enough

Tab IV: Accuracy of SAMR and **relative accuracy** when running HAMR with different numbers of partitions P.

Stream	SAMR	P=2	P=4	P=8
Electricity	0.7187	0.9607	0.9248	0.8935
Covertype	0.6220	0.9702	0.9542	0.9034
Weather	0.7008	1.0013	1.0065	0.9891
Activity	0.3213	1.0539	1.1270	1.1849
Poker	0.5548	0.9682	0.9492	0.9326
Olympic	0.8527	1.0002	1.0003	0.9999
Hyperplane	0.5822	0.9828	0.9768	0.9564
Wine	0.6033	0.9924	0.9970	0.9474
Connect4	0.5618	0.9835	0.9827	0.9930
Lymph	0.4900	1.0046	1.0038	1.0004
Avg	**0.6007**	**0.9918**	**0.9922**	**0.9801**
Std	0.1423	0.0265	0.0545	0.0818

while increasing throughput at the same time. To decide if the impact of such approach is meaningful, we would need to compare results with a single classifier that imply omit some instances.

Tab V: Computing time [s] for SAMR and **speed-up** when running HAMR with different numbers of partitions P.

Stream	SAMR	P=2	P=4	P=8
Electricity	7.093	2.2263	3.1595	4.7413
Covertype	24.453	1.7980	3.6410	3.7293
Weather	1.854	1.5593	1.7247	2.0108
Activity	57.101	1.7665	2.5766	7.2381
Poker	29.076	2.8361	5.0435	4.8452
Olympic	154.433	2.7139	5.8209	11.9373
Hyperplane	56.423	2.6296	8.2417	9.4369
Wine	26.612	2.7805	3.6236	4.0413
Connect4	7.37	1.7370	2.7801	2.9899
Lymph	37.565	2.6975	3.5055	4.1176
Avg	**40.198**	**2.2745**	**4.0117**	**5.5088**
Std	44.4792	0.5118	1.8947	3.1022

Conclusion. Finally, we can conclude that the horizontal parallelization provides better relation between accuracy relative to the sequential algorithm and obtained speed-up. In Fig. 2 we can see that HAMR benefits from increased parallelization more that VAMR. In the former case, it is possible that the rate of incoming updates is too slow compared with the rate of incoming instances. Increasing the number of partitions even more, may boost the performance of this method. On the other hand, HAMR keeps accuracy on a relatively stable level, while providing significant speed-up. It is possible that connecting both methods may lead to even more improvements [x].

However, one must keep in mind that all the presented results were obtained on a single machine with a limited number of cores. Further evaluation is required to analyze the observed trends while working with multiple physical machines (communication bottlenecks may be significant) providing more partitions to utilize.

V. Summary

In the presented work, we introduced our implementation of the distributed ARM classifier in Apache Flink, using

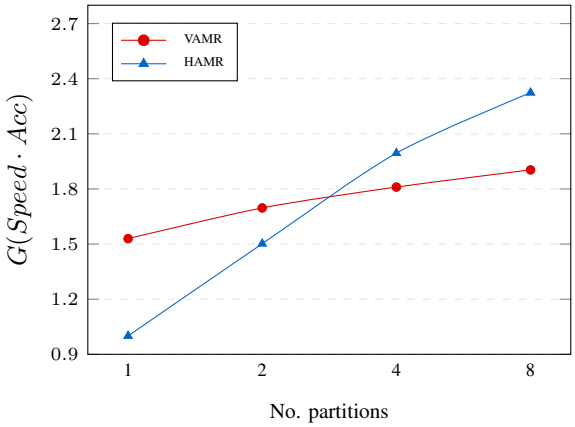

Fig. 2: Geometric mean of speed-up (Speed) and relative accuracy (Acc) for VAMR and HAMR (P=1 is SAMR).

two previously established general architectures – vertical and horizontal. We evaluated VAMR and HAMR on a set of well-known data streams, using different numbers of parallel partitions. The obtained results suggest that the horizontal version better exploits increased resources in terms of maintaining a sufficient accuracy and providing a speed-up.

We treat this work as a preliminary one. In the nearest future, we plan to improve the existing solutions, aiming for increasing both the quality of classification and throughput. An increased number of distributed partitions may give us an opportunity to perform additional optimization steps independently from the main flow. In addition, since rules can be easily interpreted, once we obtain reliable predictions in a reasonable time, we may attempt to analyze how rules are changing for some data streams – that may lead to some indirect observations about the dynamics of these streams and that knowledge could be utilized in subsequent projects.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.