ASML: Automatic Streaming Machine Learning

Ignacio Cano
Computer Science & Engineering
University of Washington
icano@cs.washington.edu

Muhammad Raza Khan
The Information School
University of Washington
mraza@uw.edu

ABSTRACT

Beyond the well-studied problem of scale in Big Data systems, the high velocity at which new data is generated and moved around introduces new challenges. It becomes critical to build systems that can process high speed data efficiently in order to extract useful insights, having access to Big Data is not good unless you can turn it into value.

As opposed to typical offline/batch machine learning scenarios, in streaming settings, data is not accessible beforehand, we need to learn as the data arrives, and we should make fast predictions in order to support real-time decisions, e.g. decide whether or not to show an ad to a user based on the probability he/she will click on it.

In this work, we propose *ASML*[1], an automatic streaming machine learning system that continuously learns predictive models from streaming data arriving at a fast pace. In contrast to the traditional streaming setting, where a single model is usually trained, our system trains several models in parallel and is capable of predicting with the right model at the right time by automatically switching between them.

Our preliminary empirical results on a public available dataset confirms the general validity of our approach and shows some improvements over traditional streaming settings.

1. INTRODUCTION

With the exponential increase of data on the web and the proliferation of sensors with the so called Internet of Things, it becomes necessary to build systems that can process continuous streams of data and are capable of extracting useful information out of it. The data is too big, moves too fast, and decisions need to be made on real-time. In order to cope with these challenges, we need new alternatives (systems) to efficiently analyze and process these never ending streams [14].

The Online Machine Learning framework fits well in this streaming scenario. Typical online learning algorithms operate on a single instance at a time[2], allowing for updates that are fast, simple and perform well in a wide range of practical settings. These algorithms operate in rounds. In round t the algorithm receives an instance $x^t \in R^d$ and applies its current prediction rule to make a prediction $\hat{y}^t \in Y$. It then receives the true label $y^t \in Y$ and suffers a loss $l(y^t, \hat{y}^t)$. Finally, the algorithm updates its model using (x^t, y^t) and proceeds to the next round [5]. Examples of online learning applications include anomaly detection in networking data, pattern recognition on stock exchange transactions, and online email categorization [15].

Online learning means training a model as the data arrives, as opposed to the offline mode, where a static dataset is available. Some challenges arise: a) in general, more data is available, but there are tighter time constraints (need real-time predictions), b) the concepts learned might change over time, and c) the ability to store examples is very limited compared to the size of the data set, i.e. you can only see an example once, and have to forget about it after you've used it to update your model. Storing the most "relevant" data points (either the most recent ones, the most diverse, etc.) might be possible, but usually load shedding takes place.

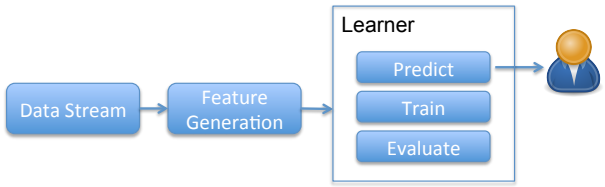

Figure 1: Traditional Streaming Pipeline. The data arrives in a streaming fashion, and is transformed by a Feature Generation module. This module forwards the example(s) to a Learner module, where predictions are made (e.g. the probability that the user will click on an ad). Once the truth label arrives (e.g. whether the user clicked on the ad or not), the model is trained. On regular intervals, the model is evaluated, and the process continues (ideally never ends).

Figure 1 illustrates a typical online or streaming pipeline. The raw data arrives at a fast pace, possibly from a network connection. Raw data is transformed into "formatted" examples that are fed into a classifier (or regressor). Prediction(s) are made, and sent to the final users. Once the truth labels arrive, the decision function is updated (i.e. the model learns), and the process repeats. On predefined intervals, the model is evaluated. By looking at the evaluation results, a data scientist might decide to *manually* change to a non-linear model as a linear classifier may not suffice anymore. We find this *manual* process tricky and error-prone. We therefore challenge the current practice and propose a system that uses a new streaming pipeline.

[1]https://github.com/nachocano/asml
[2]A mini-batch of examples can also be possible.

Contribution: Our system *ASML* enables *parallel training* of multiple models over streaming data and *automatic* model selection on a timely basis. To the best of our knowledge, no other open-source streaming system provides such characteristics out-of-the-box. Furthermore, we include the possibility of hot deployment of components, allowing users to roll out new Learners while the system is still running. We also handle nodes failure, and support different evaluation metrics and modes.

The system design favors flexibility towards seamless integration of new modules and components. Our initial experiments indicate that *ASML* can outperform the traditional streaming pipeline approach.

The remainder of this paper presents these findings as follows: §2 introduces our proposed system and describes its components, §3 describes some of the key design and implementation decisions. We explain the evaluation setup in §4 and show some preliminary results in §5. Finally, we discuss related work in §6, future work in §7, and conclude in §8.

2. ASML

Instead of training a single model in a streaming fashion, we propose a system that trains multiple models in parallel, and use the current best to make the final predictions. Our system continuously evaluates, compares models, and chooses the best one so far in order to output its predictions.

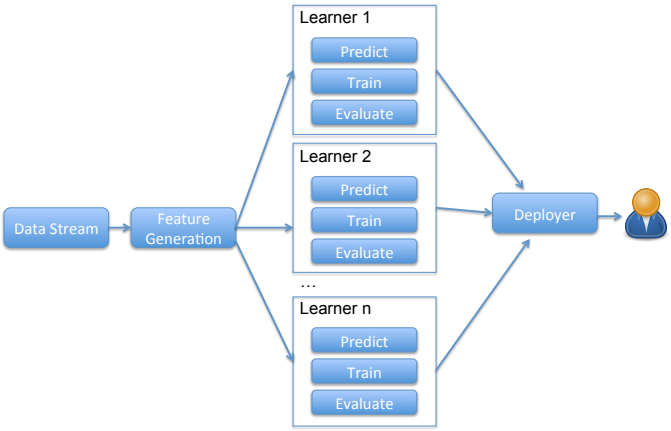

Figure 2: Our Streaming Pipeline instantiated by our system ASML. Data arrives at a fast pace and is forwarded to a Feature Generation component, which outputs example(s) for learning algorithms. Those examples are fed into several Learners. Every Learner predicts, and at regular intervals (or in a streaming fashion) they evaluate their performances. The predictions, together with each Learner's evaluation metric, are forwarded to a Deployer component, responsible for choosing which predictions to output (in general it will select the predictions done by the current best model). Once an example's label arrives, every Learner trains its model, and the flows continuous.

Similarly to the traditional pipeline shown in Figure 1, our pipeline illustrated in Figure 2 includes Data Stream and Feature Generation components. The main difference arises on the learning stage. Our system incorporates many Learners, trained in parallel, and a final Deployer component. The Deployer is in charge of deciding which predictions made by the different Learners should be sent to the final user. Different policies could be used, e.g. current best model, voting, weighted interpolation, etc. We support the current best model policy. The other alternatives could also help in boosting the overall performance, but they are left to future work.

2.1 Components

In this section, we briefly describe the components of our system.

2.1.1 *Data Stream*

Represents the stream of data. It augments the raw examples with unique identifiers (timestamps) and outputs these new time-aware examples to the Feature Generation module. We currently support a file-based implementation (File Stream), where we simulate data coming in a streaming fashion. We plan to support network-based streams in the near future.

2.1.2 *Feature Generation*

The Feature Generation module receives data from the incoming Data Stream and generates features on the fly. Amongst other things, it incrementally compute maxs, mins, reduces the features dimensionality using hashing kernels [17], etc. Similarity between different hashes [10] was also explored but not incorporated due to scalability issues. This module always outputs examples in the popular SVMLight format[3].

2.1.3 *Learner*

The Feature Generation module feeds the Learners, which perform the incremental training of the machine learning models. Each Learner trains a different model in parallel. We currently support the follow linear models[4]:

- L1 Logistic Regression.
- L2 Logistic Regression.
- Elastic Net Logistic Regression.
- Support Vector Machines (SVM).
- Perceptron.
- Passive Aggressive I (with Hinge Loss).
- Passive Aggressive II (with Squared Hinge Loss).

2.1.4 *Deployer*

The Deployer is the key component in our architecture. It receives the predictions from the different Learners together with their evaluation metrics. Based on its configured policy, it decides which predictions to output.

3. DESIGN AND IMPLEMENTATION

In this section we describe the design considerations of our system.

[3]http://svmlight.joachims.org/

[4]Non-linear models are left to future work.

3.1 Architecture

We propose a distributed dataflow system. Each of the components described in §2.1 runs its own server process. We use RPC-based communication between the components. The underlying implementation uses Apache Thrift services[5], which allows the system to work efficiently and seamlessly across different languages. The only requirement is to comply with the streaming interface contract described in §3.2.

3.2 Push instead of Pull

We evaluate two types of systems: a Pull-based system with a **getNext** method, similar to the Iterable interface used by query operators, and a Push-based system with an **emit** method, which keeps on pushing the data downstream. We decide to use the latter, similar to Aurora [1], as it better fits the streaming model and simplifies the implementation. We therefore define a streaming interface as a Thrift service. Every component in the system implements such interface, which includes the following method:

```
void emit(list<string> data)
```

The format of the data elements varies per component[6]. Each component knows the format of the data sent by its parent.

3.3 Fault tolerance

Our system offers model check-pointing. We currently support only Learners recovery. We allow the user to define the checkpoint interval (e.g. checkpoint the model every x batches of examples). We persist the model state into a PostgreSQL database. When a Learner restarts after some failure (due to a machine crash, an out of memory error, etc.), the system fetches its most recent check-pointed model in order to avoid training from scratch all over again.

3.4 Registry

We provide a Registry service to register/unregister components. It acts as a bookkeeper and keeps track of the available components in the system. Every new component needs to register with this service. It is mainly needed for hot deployment of Learners.

3.5 Historical Points

Similarly to Aurora [1], we allow for some historical storage. The user can specify the number of historical data points to persist into a PostgreSQL instance. We currently support persistence specs such as: keep the last 3 batches of data. The task in charge of the persistence is constantly deleting older data, which allows to keep the database in a manageable size.

The historical points play an important role when adding Learners to a running system. For example, if a user decides to start training a non-linear model (e.g. Gradient Boosted Trees) after a day of training linear classifiers, it will be beneficial to allow the Tree model to catch up with the models that have already seen one day worth of data. By being able to look at some previous data (plus the continuously incoming stream), the newly created classifier will be able to start producing meaningful results sooner.

[5]https://thrift.apache.org/

[6]They are not strictly tuples as in the relational model, instead, the data elements are more similar to NoSQL documents.

3.6 Evaluation Modes

We support two common evaluation modes in streaming settings: Holdout and Prequential.

In the holdout evaluation, the current model is applied to an independent test set at regular time intervals (or batch of examples). For a large enough test set, the loss estimated in the holdout is an unbiased estimator [9]. In practice, as the data stream evolves over time, the holdout set may become a non-representative portion of the streaming data, resulting in a poor performance indicator.

Instead, in Prequential (or Predictive Sequential), the error of a model is computed from the sequence of examples. The actual model makes a prediction for each example in the stream. In this scenario, the prequential error estimated over all the stream might be strongly influenced by the first part of the error sequence, when only few examples have been used to train the classifier. The model used to classify the first example is different from the one used to classify the hundred-th instance. This observation leads to the idea of computing the prequential error using a forgetting mechanism. We use the following fading factor approach proposed by Gama et al. [9], where the fading sum $S(i)$ of observations from a stream x is computed at time i, as:

$$S(i) = x_i + \alpha \times S(i-1) \tag{1}$$

where $S(1) = x_1$ and $0 \ll \alpha \leq 1$ is a constant determining the forgetting factor of the sum, which should be close to 1. The fading average at i is then computed as:

$$M(i) = \frac{S(i)}{N(i)} \tag{2}$$

where $N(i) = 1 + \alpha \times N(i-1)$ is the corresponding fading increment, with $N(1) = 1$.

3.7 Configuration

Our system is highly configurable. Each component reads a property file on startup, where the user can configure the parser used to read the data, checkpoint intervals, batch size, evaluation modes, evaluation metrics, log files, etc.

4. EVALUATION

To assess our contributions, we use a pre-processed subset of the click-through rate dataset from 2012 KDD Cup Track 2[7]. It consists of around 2.4M examples and 100K sparse features.

We run some preliminary experiments to show the validity of our approach. We use three Learners, each one trains a logistic regression model with different regularizer priors: L1, L2 and Elastic Net. We use the Prequential evaluation scheme described in §3.6, and set α to 0.995 (i.e. we give higher importance to newer data).

5. RESULTS AND DISCUSSION

The overall RMSE of the individual learners and the one provided by our system is shown in Figure 3. We see that our approach performs the best, followed by Elastic Net and L2. Although the difference is small, in this particular task (click-through rate prediction of ads), a tiny improvement may result in significant monetary benefits.

[7]http://www.kddcup2012.org/c/kddcup2012-track2

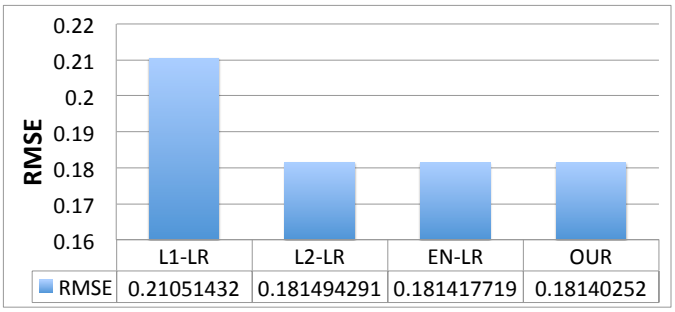

Figure 3: Overall RMSE after a whole pass of the streaming data. Our system performs the best as it switches between L2 and EN at the right time (i.e. when EN has lower RMSE than L2).

Figure 4 illustrates the RMSE evolution over time, and helps explaining the result in Figure 3. We see that the L1-regularized classifier performs the worst during the entire learning process. At the beginning of the optimization, the L2 classifier performs the best, therefore our system chooses it (their RMSE curves overlap for the first 1.25M examples while the Elastic Net curve is slightly worse). During the last part of the training (after 1.25M examples), the Elastic Net model dominates, hence, our system does an *automatic switch* and starts predicting with this "better" classifier (their RMSE curves overlap while the L2 curve is slightly worse).

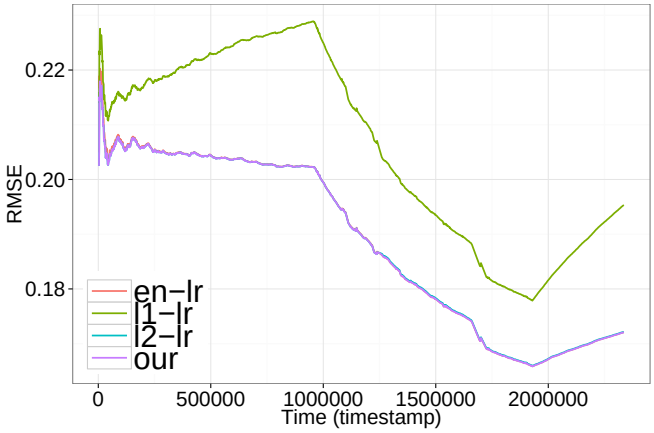

Figure 4: RMSE evolution over time. The Deployer decision is based on these curves, it chooses to output the predictions of the model with the minimum RMSE. During the first half of the training procedure, L2 performs the best. After 1.25M examples, Elastic Net takes over.

6. RELATED WORK

There are many open-source frameworks that provide high-level mechanisms for doing large-scale analytics. Spark Streaming [18] and Twitter Storm [16] are two popular ones in the streaming community, though they have limited support for machine learning algorithms. Instead, our system specifically targets ML workloads.

The design of online machine learning algorithms has been the topic of a broad research agenda [13, 8, 6]. To the best of our knowledge, there is no end-to-end streaming machine learning system that incorporates many of these algorithms out-of-the-box. Although we only support a few [3, 4], it is almost "trivial" to incorporate more with our flexible design.

The work by Dietterich et al. provides an overview of machine learning for streaming data [7]. They mention that many of the assumptions valid for offline/batch learning do not apply for streaming settings, such as evaluation metrics, ways of generating features and even model selection. In this project, we go a step further and implement a mechanism that deals with such problems.

The paper by Bifet on adaptive learning and mining over streaming data assumes the system can detect changes only if there aren't many [2]. This approach may not be suitable in cases where there can be many small changes over a period of time, as some of those changes can go undetected. In our system, as we are continuously evaluating the performance, even frequent small changes can be detected, and models switched.

Hosseini et al. use an ensemble of classifiers for semi-supervised classification of non stationary data streams [11, 12]. Their work is somewhat similar to ours in the sense that they train many learners instead of just one. They do predictions with the ensemble instead of the current best, which is something we left to future work. However, they mainly focus in the Learner component, whereas we provide an end-to-end solution, from the data source, feature generation, learning, model selection and prediction.

7. FUTURE WORK

A number of avenues exist to address further questions related to our Automatic Streaming Machine Learning system.

Feature generation for streaming data poses many research challenges as even the simple task of calculating similarity between tokens becomes an expensive task for streaming data that contains high dimensional vectors. Changing the feature dimensionality on the fly (either expand or shrink) and measuring how the change affects performance, can be an interesting path to follow. Doing efficient incremental generation of complex features (e.g. based on user profiles that change over time), will require incorporating approximation algorithms, and all the theory behind them (error bounds, etc).

Further experimental investigations are needed to explore the trade-off between latency and throughput with respect to the batch size. A great deal of improvement can be generated from the incorporation of non-linear models (Trees, Neural Nets), and testing with bigger datasets (and on other tasks). Being able to learn from multiple streams (Text, Image, Video) may enable interesting inter-disciplinary collaborations.

Adding fault-tolerance capabilities to other components beyond the Learners can be the right step to make the project more appealing to end-users. Another avenue of further exploration would be adding real-time visualization gadgets to easily debug the learning process.

Another interesting extension of this project can be the support of queries over data streams in real time, and target other types of learning problems, mainly unsupervised learning.

8. CONCLUSION

Data is being generated at a massive scale all around the globe. Besides the scaling factor, data's velocity introduces new fundamental challenges. Hence, there is an urgent need for new tools and techniques that can efficiently transform streaming data into useful information and knowledge.

In this work, we proposed *ASML*, a streaming machine learning system that adapts to changes in data by automatically switching between models on real-time. Our preliminary empirical results showed improvements over traditional streaming pipelines, where single models are usually trained, but many research challenges remain open.

Acknowledgments

Special thanks to Tianqi Chen and Rahul Kidambi for the helpful discussions about the project. We will like to thank the CSE544 teaching staff for their guidance and support during the quarter.

9. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] A. Bifet. Adaptive learning and mining for data streams and frequent patterns. *SIGKDD Explor. Newsl.*, 11(1):55–56, Nov. 2009.

[3] L. Bottou. Large-scale machine learning with stochastic gradient descent. In Y. Lechevallier and G. Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer.

[4] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *JOURNAL OF MACHINE LEARNING RESEARCH*, 7:551–585, 2006.

[5] K. Crammer, A. Kulesza, and M. Drezde. Adaptive regularization of weight vectors. In *Advances in Neural Information Processing System*. 2009.

[6] S. de Rooij, T. van Erven, P. D. Grünwald, and W. M. Koolen. Follow the leader if you can, hedge if you must. *CoRR*, abs/1301.0534, 2013.

[7] T. G. Dietterich. Machine learning for sequential data: A review. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, London, UK, UK, 2002. Springer-Verlag.

[8] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[9] J. Gama, R. Sebastião, and P. P. Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346, 2013.

[10] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[11] M. J. Hosseini, Z. Ahmadi, and H. Beigy. Pool and accuracy based stream classification: a new ensemble algorithm on data stream classification using recurring concepts detection. In *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on*, pages 588–595. IEEE, 2011.

[12] M. J. Hosseini, A. Gholipour, and H. Beigy. An ensemble of cluster-based classifiers for semi-supervised classification of non-stationary data streams. *Knowledge and Information Systems*, pages 1–31, 2015.

[13] H. B. McMahan. In *AISTATS*, pages 525–533. JMLR.org.

[14] O. Media. Volume, velocity, variety: What you need to know about big data, 2012. http://www.forbes.com/sites/oreillymedia/2012/01/19/volume-velocity-variety-what-you-need-to-know-about-big-data/.

[15] S. Shalev-Shwartz. Online learning: Theory, algorithms, and applications. Technical report, 2007.

[16] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[17] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA, 2009. ACM.

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.